

SQL Server 2014
SSL Visual Studio 2013 ORM
MVVM AJAX SEO WCF RWD IoC
Linq AGILE SCRUM TDD
JSON jQuery
CSS HTML5
JavaScript REST Service
ASP.NET
EF 6 MVC 5
C# 6.0
WEB API 2.0 Windows AZURE
Google Facebook HTML DOM
AngularJS AJAX MVP
MongoDB NoSQL

Krzysztof Żydzik, Tomasz Rak

C# 6.0 i MVC 5

Tworzenie nowoczesnych
portali internetowych

Helion

Autorzy:

Krzesztof Żydzik, Microsoft Certified Solutions Developer, Microsoft Certified Professional, Microsoft Specialist

Tomasz Rak, Politechnika Rzeszowska, Wydział Elektrotechniki i Informatyki, Katedra Informatyki i Automatyki, Rzeszów, Polska

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

Redaktor prowadzący: Michał Mrowiec

Projekt okładki: Studio Gravite / Olsztyń
Obarek, Pokoński, Pazdrijowski, Zaprucki

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

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
http://helion.pl/user/opinie/c6mvc5_ebook
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Kody źródłowe kolejnych kroków tworzenia przykładowej aplikacji dostępne są pod adresem:
<ftp://ftp.helion.pl/przyklady/c6mvc5.zip>

ISBN: 978-83-283-0864-0

Copyright © Helion 2015

- [Poleć książkę na Facebook.com](#)
- [Kup w wersji papierowej](#)
- [Oceń książkę](#)

- [Księgarnia internetowa](#)
- [Lubię to! » Nasza społeczność](#)

Spis treści

Wstęp	15
Rozdział 1. C# — teoria i praktyka	19
Wprowadzenie do języka C#	19
Kolejne wersje języka C#	20
C# 2.0 (.NET Framework 2.0, Visual Studio 2005)	20
C# 3.0 (.NET Framework 3.5, Visual Studio 2008)	20
C# 4.0 (.NET Framework 4.0, Visual Studio 2010)	21
C# 5.0 (.NET Framework 4.5, Visual Studio 2012 oraz 2013)	21
C# 6.0 (zapowiedź)	21
Konwencje	21
Cel stosowania konwencji	22
Pliki a klasy i interfejsy	22
Wcięcia	22
Komentarze	23
Deklaracje klas, interfejsów i metod	24
Puste linie	24
Nawiasy klamrowe	25
Konwencje nazewnicze	25
Pozostałe dobre praktyki	26
Typy	26
Deklaracja zmiennej	27
Inicjalizacja zmiennej	28
Słowa kluczowe	29
Stałe i zmienne tylko do odczytu	29
Literałы	30
Typ wyliczeniowy	31
Konwersje typów i rzutowanie	31
Opakowywanie (boxing) i rozpakowywanie (unboxing)	32
Wartości zerowe oraz typy dopuszczające wartości zerowe	33
Typy generyczne	34
Tablice,łańcuchy i kolekcje	34
Tablice	34
Łańcuchy	36
Kolekcje	37
Operatory	38
Operator trójargumentowy ?:	38
Operator ??	40

Instrukcje sterujące	40
Instrukcja if	40
Instrukcja switch	41
Instrukcje iteracyjne	43
Pętla while	43
Pętla do while	43
Pętla for	44
Pętla foreach	45
Instrukcje skoku	45
Klasy, obiekty, pola, metody i właściwości	46
Klasy	46
Obiekty	48
Pola	48
Metody	49
Właściwości	52
Podstawowe pojęcia związane z programowaniem obiektowym	53
Abstrakcja	53
Hermetyzacja	54
Dziedziczenie	54
Polimorfizm	55
Przeciążanie operatorów	56
Przeciążanie operatorów relacji	57
Metody Equals() i GetHashCode()	57
Przeciążanie operatorów konwersji	57
Przeciążanie operatorów logicznych	58
Przeciążanie operatorów arytmetycznych	60
Przeciążanie metod	61
Indeksatory	61
Klasa System.Object	63
Konstruktor i destruktor	64
Garbage Collector	66
Zasada działania GC	66
Podział na generacje (przechowywanie obiektów w pamięci)	66
Struktury	67
Interfejsy	68
Jawna implementacja interfejsu	69
Zwalnianie zasobów niezarządzanych	70
Interfejs IDisposable	70
Słowo kluczowe using	71
Delegaty, metody anonimowe, wyrażenia lambda i zdarzenia	72
Delegaty	72
Metody anonimowe	74
Wyrażenia lambda	74
Zdarzenia	75
Dyrektyny preprocesora	76
Wyjątki	77
Zgłaszanie wyjątków	77
Przepelnienia arytmetyczne	78
Instrukcje checked i unchecked	78
Przestrzenie nazw	79
Zagnieżdżanie przestrzeni nazw	79
Dyrektiva using	81
Aliases	82
Zewnętrzne aliasy	83

Typy, metody, klasy i kolekcje uogólnione (generyczne)	83
Metody generyczne	84
Klasy generyczne	84
Kolekcje generyczne i interfejsy	86
Interfejs IDictionary< TKey, TValue > — słownik	86
Interfejs IEnumerable< T >	87
Interfejs ICollection< >	87
Interfejs IList< >	87
Interfejs IQueryable< >	88
Wyrażenia regularne	89
Data i czas	90
Operacje wejścia, wyjścia, foldery i pliki	92
Pozostałe elementy języka i nowości w wersji C# 5.0	93
Mechanizm refleksji i atrybuty	93
Atrybuty	94
IEnumerable a IEnumerator	96
Iteratory i słowo kluczowe yield return	97
Inicjalizatory obiektów i kolekcji	100
Drzewa wyrażeń	101
Metody rozszerzające	102
Metody i klasy częściowe	103
Metody częściowe	104
Zmienne domniemane	105
Typy anonimowe	105
Słowa kluczowe this i base	106
Typy dynamiczne	107
Argumenty nazwane — Named Arguments	110
Parametry opcjonalne	111
Obsługa kontra- i kowariancji oraz słowa kluczowe in i out	111
Słowa kluczowe is, as i typeof	114
Leniwa inicjalizacja — Lazy Initialization	114
Metody asynchroniczne — async i await	118
Atrybuty Caller Info	119
Nowości w C# 6.0	120
Konstruktorzy pierwotne — Primary Constructors	120
Automatyczna inicjalizacja właściwości — Initializers for Auto-properties	120
Dyrektywa using dla składowych statycznych — Using Static Members	121
Inicjalizatory słownikowe — Dictionary Initializer	121
Deklaracje inline dla parametrów out — Inline Declarations for Out Params	122
Wyrażenia dla właściwości — Property Expressions	122
Wyrażenia dla metod — Method Expressions	122
Modyfikator private protected	123
Kolekcje IEnumerable jako parametr — Params for Enumerables	123
Jednoargumentowe sprawdzanie wartości null — Monadic Null Checking	123
Słowo kluczowe await w blokach catch i finally	124
Filtry wyjątków — Exception Filters	124
Literaly binarne i separatory cyfr — Binary Literals, Digit Separators	124
Rozdział 2. Wzorce architektoniczne	125
Architektura wielowarstwowa	125
Architektura jednowarstwowa	126
Architektura dwuwarstwowa	126
Architektura trójwarstwowa	126
Architektura n-warstwowa	126

MVC	127
View	128
Controller	128
Model	128
Domain Model, MVC Model i ViewModel — porównanie	128
Model pasywny a model aktywny	129
MVP	129
Model	130
View	130
Presenter	130
MVVM	131
MVC, MVP i MVVM	131
DDD	132
SOA	132
EDA	133
Rozdział 3. Microsoft .NET Framework	135
Struktura .NET	135
CLI	136
CIL	137
CLR	137
DLR	137
Elementy .NET wykorzystywane w ASP.NET MVC	138
Implementacje .NET	138
Projekt Mono	139
WPF	139
WCF	140
Service Contract	140
Operation Contract	140
Data Contract	140
Data Member	140
WCF Endpoint = adres + binding + contract	142
Silverlight	142
Microsoft Azure	143
Windows Azure Storage	143
BLOB Storage	143
Table Storage	143
Queue Storage	143
Hostowanie aplikacji w Azure	143
Worker Role	144
Web Role	144
Web Site	144
Virtual Machine	144
Azure Service Bus	144
Service Bus Relay	144
Service Bus Queue	145
Service Bus Topic	145
ASP.NET Web Forms	145
ASP.NET Web Pages	146
ADO.NET	146
Obiekt DataSet	147
Obiekty DataTable i DataRow	147
Obiekty DataRelation	147
Obiekty DataView	147
.NET Framework Data Provider	147

LINQ	148
LINQ to XML	148
LINQ to Objects	149
LINQ to SQL	149
LINQ to DataSet	149
LINQ to Entities	149
Przykłady zapytań LINQ	150
Składnia metod — Method Syntax	150
Składnia zapytań — Query Syntax	151
PLINQ	151
Narzędzia ORM w .NET	153
Entity Framework	153
NHibernate	153
NHibernate 3 a Entity Framework 6	154
Alternatywa dla Entity Framework i NHibernate	154
Rozdział 4. Entity Framework 6	157
Podejście do pracy z modelem danych	157
Porównanie różnych podejść	157
Model dla podejścia Model First	158
Model dla podejścia Code First	158
Nowości wprowadzane w kolejnych wersjach EF	159
Nowości wprowadzone w EF 5	159
Nowości wprowadzone w EF 6	160
Relacyjne bazy danych i EF	160
Krótki opis baz relacyjnych	160
Relacja „jeden do wielu”	161
Relacja „jeden do jednego”	161
Relacja „wiele do wielu”	162
Relacje opcjonalne	165
Obiekty DbContext i DbSet	165
DbContext i DbSet	165
Metody Attach i Detach	165
Relacje poprzez klucz FK a relacje niezależne (obiektowe)	166
Relacje poprzez klucz obcy — FK Association	166
Relacje niezależne — Independent Association	167
Odpisywanie bazy danych za pomocą EF i LINQ	168
Wczytywanie zachłanne — Eager Loading	168
Wczytywanie leniwe — Lazy Loading	169
Jawne ładowanie — Explicit Loading	170
Problem N+1	170
Metoda AsNoTracking()	171
Odroczone i natychmiastowe wykonanie	171
Entity SQL	172
Bezpośrednie zapytania SQL do bazy (Direct/RAW SQL) i procedury składowane w EF	173
Transakcje w EF	174
Śledzenie zmian	175
Migawkowe śledzenie zmian — Snapshot Change Tracking	175
Dynamiczne śledzenie zmian — Dynamic Change Tracking (proxy)	175
Zarządzanie operacjami współbieżnymi	176
Kaskadowe usuwanie — Cascade Delete	177

Strategie dziedziczenia w bazie danych — TPT, TPH i TPC	178
TPH	178
TPT	178
TPC	179
SQL Logging	179
Code First Fluent API i Data Annotations	180
Migracje	182
Metoda Seed	183
Rozdział 5. ASP.NET MVC 5	185
Kolejne wersje ASP.NET MVC	185
ASP.NET MVC 1	185
ASP.NET MVC 2	185
ASP.NET MVC 3	186
ASP.NET MVC 4	186
ASP.NET MVC 5	186
ASP.NET MVC 6 (zapowiedź)	187
Konwencje w MVC	187
Struktura projektu	187
Konwencje a ASP.NET MVC	188
MVC Pipeline — ścieżka wywołań, handlery i moduły	189
Ścieżka wywołań	189
Pierwsze żądanie do aplikacji ASP.NET	189
Podstawowe obiekty tworzone dla każdego żądania	189
HttpApplication	190
Uchwyty i moduły HTTP	193
Uchwyty HTTP	193
Moduły HTTP	193
HttpHandler a HttpModule	193
Kontroler	194
Typy rezultatu	194
Parametry akcji	196
Żądanie GET	196
Żądanie POST	196
Filtry akcji	197
Widok	200
Zasady odnajdywania widoków	200
Folder Shared	201
Widoki częściowe	201
Razor	202
Dodatkowe właściwości silnika Razor	203
ViewBag, ViewData i TempData	204
Widoki typowane — Strongly Typed Views	205
HTML helpers	208
Paczki skryptów i minimalizacja — Script/CSS Bundling and Minification	209
Sekcje	211
Routing	214
Kolejność w routingu	214
Ignorowanie ścieżek	214
Ograniczenia	215
Routing na podstawie atrybutów	215
Prefiksy	216
Ograniczenia	217
Nazwywanie ścieżek i generowanie linków po nazwie ścieżki	217
Obszary	217

Model	218
ViewModel	218
Walidacja	222
MVC Scaffolding	223
Generowanie kontrolerów	223
Generowanie widoków	227
Metody synchroniczne i asynchroniczne w MVC	228
Słowa kluczowe — Async, Await, Task	230
Cache	231
Cachowanie po stronie serwera — Server Side Caching	231
Atrybut OutputCache	232
Cachowanie częściowe	233
Cachowanie rozproszone	233
Cachowanie po stronie klienta — Client Side Caching	234
Cachowanie w HTML 5	234
HTML 5 Application Cache	234
HTML 5 WebStorage	234
Code First Data annotations	235
Bezpieczeństwo	235
SQL Injection	236
Cross-Site Request Forgery	236
Cross-Site Scripting	237
Over-Posting — parametr binding	237
Obsługa, śledzenie i logowanie wyjątków w MVC	238
Lokalne zarządzanie wyjątkami	238
Blok try-catch	238
Nadpisywanie metody OnException() w kontrolerze	238
Globalne zarządzanie wyjątkami	239
Klasa FilterConfig	239
HandleError na poziomie kontrolerów i akcji	239
Zwracanie widoków dostosowanych do konkretnych typów wyjątków	240
Logowanie globalne za pomocą osobnych narzędzi	240
Identyfikacja, uwierzytelnianie i autoryzacja w MVC 5	241
Identyfikacja	241
Uwierzytelnianie	241
Autoryzacja	242
Role w MVC	242
Stan aplikacji, sesje i ciasteczka	242
Stan aplikacji	242
Ciasteczka	243
Sesje	243
OWIN	244
ASP.NET Identity	244
WIF i uwierzytelnianie za pomocą claimów	245
Identity Provider, STS	246
Strona ufająca — Relying Party	246
Federated Authentication	247
Windows ACS	248
OpenId i OpenAuth	249
OpenId	249
OpenAuth	250

Rozdział 6. Web serwisy i ASP.NET Web API 2	251
Web API 2	251
Web API a ASP.NET MVC	252
Web serwis, REST, SOAP i OData	253
SOAP	253
REST	253
OData	254
CORS i JSONP	255
JSONP	255
CORS	255
Uruchamianie CORS w Web API	256
Routing w Web API	257
Mapowanie żądań na akcje bądź metody w kontrolerze Web API	257
Web API a Entity Framework i warstwa modelu	258
Typy rezultatu w Web API	258
Typ void	259
HttpResponseMessage	259
IHttpActionResult	260
Inny dowolny typ z aplikacji	260
Pobieranie danych z Web API	261
Pobieranie danych po stronie serwera (.NET, C#)	261
Pobieranie danych po stronie klienta (JavaScript, jQuery, AJAX)	261
Wersjonowanie w Web API	262
Rozdział 7. Narzędzia, licencje i ceny	263
Serwer IIS	263
Kategorie dla modułów dostępnych w IIS	263
Pule aplikacji w IIS	264
Przetwarzanie żądań w IIS	264
Microsoft SQL Server 2014	264
Licencjonowanie SQL Server 2014	265
Ceny licencji SQL Server 2014	265
Nowości w SQL Server 2014	266
Windows Server 2012	267
Wersje Windows Server 2012	267
Licencjonowanie Windows Server 2012	267
Ceny Windows Server 2012	268
Microsoft Visual Studio 2013 Ultimate	268
Snippety	269
Page Inspector	269
Nowości w Visual Studio 2013	269
Poprawiony pasek przewijania	270
Podgląd definicji	270
Browser Link	270
JSON Editor i JavaScript	271
Powiązanie z Microsoft Azure	272
Wsparcie dla GIT	272
Najważniejsze skróty klawiszowe	272
Rozdział 8. Aplikacja i wdrożenie	277
Wzorce projektowe i architektoniczne wykorzystywane w .NET	277
Repozytorium	277
Wzorzec IoC	277
Repozytorium generyczne	278
Wzorzec UnitOfWork	278

Przykładowa aplikacja	278
Etap 1. Krok 1. Tworzenie nowego projektu i aktualizacja pakietów	279
Etap 1. Krok 2. Utworzenie modelu danych	283
Klasa Kategoria	286
Klasa Ogloszenie_Kategoria	287
Klasa Uzytkownik	287
Etap 1. Krok 3. Tworzenie klasy kontekstu	290
Etap 1. Krok 4. Przenoszenie warstwy modelu do osobnego projektu	294
Dodawanie referencji pomiędzy projektami	296
Ustawienie projektu startowego	297
Instalacja bibliotek dla nowego projektu	298
Przenoszenie plików z modelem do osobnej warstwy (projektu)	299
Etap 1. Krok 5. Migracje	300
Instalacja migracji	300
Konfiguracja migracji	301
Tworzenie migracji początkowej	302
Uruchomienie pierwszej migracji	305
Metoda Seed()	306
Zmiany w modelu i kolejna migracja	309
Praca z błędami i niespójnością w migracjach	310
Etap 1. Podsumowanie (warstwa modelu i migracje)	311
Etap 2. Krok 1. Dodawanie kontrolerów i widoków — akcja Index	311
Dodawanie kontrolera z widokami	311
Pierwsze uruchomienie aplikacji i routing	316
Lista ogłoszeń (akcja Index) — aktualizacja widoku/wyglądu strony	317
Lista ogłoszeń a pobieranie danych	321
Optymalizacja listy ogłoszeń	322
Etap 2. Krok 2. Debugowanie oraz metody AsNoTracking() i ToList()	324
Sprawdzanie wartości zmiennych	325
Metoda ToList() i odroczone wykonanie (Deferred Execution)	325
Metoda AsNoTracking()	326
Etap 2. Krok 3. Poprawa wyglądu i optymalizacja pod kątem SEO	329
Poprawa wyglądu strony za pomocą Twitter Bootstrap	329
Podświetlanie wierszy za pomocą CSS	330
Optymalizacja pod kątem pozycjonowania — SEO	331
Etap 2. Podsumowanie	333
Etap 3. Krok 1. Poprawa architektury aplikacji	334
Przeniesienie zapytania LINQ do osobnej metody	334
Przeniesienie metody do repozytorium	334
Etap 3. Krok 2. Zastosowanie kontenera Unity — IoC	336
Wstrzykiwanie repozytorium poprzez konstruktor w kontrolerze	336
Tworzenie interfejsu dla repozytorium	337
Instalacja kontenera IoC Unity	338
Wstrzykiwanie kontekstu do repozytorium	340
Cykl życia obiektu a kontener IoC	341
Etap 3. Podsumowanie	341
Etap 4. Krok 1. Akcje Details, Create, Edit, Delete	342
Details	342
Metoda Details() w repozytorium	342
Aktualizacja i optymalizacja SEO dla widoku Details	343
Delete	345
Create	353
Edit	359

Etap 4. Krok 2. Aktualizacja szablonu _Layout.cshtml	365
Etap 4. Krok 3. Widoki częściowe — PartialView	366
Etap 4. Podsumowanie	369
Etap 5. Bezpieczeństwo, uwierzytelnianie i autoryzacja dostępu	369
Uwierzytelnianie i logowanie przez portale	369
Autoryzacja — role	372
Zabezpieczanie akcji	373
Etap 5. Podsumowanie	380
Etap 6. Stronicowanie i sortowanie	381
Stronicowanie	381
Sortowanie	388
Etap 6. Podsumowanie	392
Etap 7. Ogłoszenia użytkownika, kategorie, cache i ViewModel	393
Zakładka Moje ogłoszenia	393
Cache	394
Kategorie	395
Zastosowanie HTML helpera — Html.Action	402
Zastosowanie ViewModel	403
Etap 7. Podsumowanie	406
Etap 8. Dane w JSON, zarządzanie relacją „wiele do wielu” i attribute routing	407
PartialView a dane w formacie JSON lub XML	407
Użycie attribute routingu	407
Zarządzanie relacją „wiele do wielu” i autocomplete	409
Dodatek na AspNetMvc.pl	409
Etap 8. Podsumowanie	410
Etap 9. Dodatek — tworzenie modelu dla podejścia Model First	410
Publikacja systemu na zewnętrznym serwerze hostingowym	415
Dodawanie domeny	416
Konfiguracja witryny	418
Tworzenie bazy danych	421
Tworzenie konta FTP	422
Połączenie z bazą danych poprzez SQL Server Management Studio	422
Wdrażanie aplikacji na serwer za pomocą Microsoft Visual Studio	423
Dodatek A Zasady i metodologie w programowaniu	427
Zasady	427
SOLID	427
Zasada pojedynczej odpowiedzialności (SRP)	427
Zasada otwarte-zamknięte (OCP)	428
Zasada podstawienia Liskov (LSP)	428
Zasada separacji interfejsów (ISP)	429
Zasada odwrócenia zależności (DIP)	431
GRASP	432
Creator	433
Information Expert	433
Controller	433
Low Coupling	433
High Cohesion	434
Polymorphism	434
Pure Fabrication	434
Indirection	435
Protected Variations	435
DRY	435
KISS	436

Rule of Three	436
Separation of Concern	436
YAGNI	437
MoSCoW	437
Metodologie	437
Manifest Agile	437
Scrum	439
eXtreme Programming	439
TDD	440
Dodatek B HTTP i SSL/TLS	443
HTTP	443
SSL/TLS	447
Rodzaje certyfikatów	449
Zakup certyfikatu SSL	450
Aktywacja, walidacja i instalacja certyfikatu SSL	450
Certyfikat w praktyce	450
Dodatek C HTML 5 i CSS 3	453
HTML 5	453
Sekcje	457
Nowe typy pól formularza	458
Atrybuty dla formularza	459
Znaczniki	459
Web Storage	461
Server Side Events	461
WebSockets	462
Drag and Drop	463
Geolokalizacja	465
Walidacja	466
CSS 3	466
Nowe selektory	471
Nowe właściwości	472
Twitter Bootstrap	473
CSS 4	473
Dodatek D HTML DOM i JavaScript	477
HTML DOM	477
Metody dostępne w DOM	477
Właściwości dostępne w DOM	479
Poziomy DOM	479
JavaScript	480
Składnia języka	480
Możliwości JavaScriptu	483
JQuery	484
Instalacja jQuery	485
Selektory i filtry	485
Zdarzenia	487
Efekty w postaci animacji	488
Metody	488
Przechodzenie po elementach HTML	489
JQuery UI	490
JQuery Mobile	490

AJAX	491
JSON	492
XMLHttpRequest	493
AJAX w jQuery	494
Dodatek E Bazy nierelacyjne	497
MongoDB	498
RavenDB	498
Dodatek F Podstawy pozycjonowania w Google	499
Metatagi	499
Znacznik <title>	500
Opis strony	500
Słowa kluczowe	500
Wartości noindex inofollow	500
Znaczniki HTML	500
Linkowanie	501
Zaplecze, katalogi stron i prekle	502
Skrypty katalogów	502
Skrypty blogowe	503
Schematy linkowania	503
Schemat koła	503
Schemat piramidy	504
Gwiazda	504
Schematy mieszane	505
Linkowanie wewnętrzne	505
„Długi ogon”	505
Przyjazne adresy URL — Friendly URL	506
Pliki związane z pozycjonowaniem	506
robots.txt	506
sitemap.xml	507
.htaccess	508
Filtryle i kary	509
Ban	509
Sandbox	509
Zmiany algorytmu Google	510
Panda	510
Pingwin	510
EDM	511
Narzędzia związane z pozycjonowaniem	511
Google Analytics i Google Webmasters Tools	511
Narzędzia do pracy z tekstem	511
Systemy wymiany linków	512
Półautomaty „dodawarki” i automaty do postowania	512
Inne narzędzia	513
Skorowidz	515

Wstęp

Dziedzina związana z konstruowaniem aplikacji webowych rozwija się bardzo szybko. Ma to niewątpliwie związek z ciągłym rozwojem branży informatycznej oraz jednego z głównych kierunków, jakim jest internet. Ponieważ internet jest aktualnie jedną z najważniejszych dziedzin i stał się nieodłącznym elementem życia wielu ludzi, aplikacje internetowe stanowią obecnie najpopularniejszą część programowania. Wiele aspektów życia codziennego zostało częściowo lub całkowicie przeniesionych do internetu. Można śmiało powiedzieć, że duża część gospodarki bazuje na internecie. Za pośrednictwem sieci globalnej wyszukuje się informacje, dokonuje zakupów i płatności, pobiera i wysyła dokumenty, komunikuje z innymi osobami czy wręcz pracuje bez wychodzenia z domu. Internet pozwala na szybsze załatwianie wielu spraw życia codziennego i biznesowego. Prawdopodobnie najważniejszą zaletą internetu jest jego wpływ na bardzo szybki rozwój cywilizacji. Dawniej studenci (i nie tylko oni) musieli szukać informacji w książkach. Teraz są one dostępne od ręki poprzez wyszukiwarki internetowe w dużo lepszej i bogatszej formie.

W ostatnich latach zaobserwowano ogromny wzrost zapotrzebowania na aplikacje webowe. Każda firma potrzebuje własnego systemu, dzięki któremu pracownicy lub urządzenia będą się mogły komunikować. Internet wywarł znaczący wpływ na postrzeganie modelu aplikacji. Koncepcja aplikacji internetowej polega na komunikacji sieciowej z użyciem przeglądarki internetowej na komputerze użytkownika z aplikacją uruchomioną na serwerze. Nie jest zatem zależna od platformy systemowej. Dodatkowo wszechobecna dziś mobilność daje możliwość skorzystania z aplikacji w każdym miejscu na świecie. Użytkownik nie musi instalować dodatkowych programów ani sterowników. Przeglądarka internetowa stała się wspólnym, intuicyjnym interfejsem wielu aplikacji realizujących różne zadania. Jednak wraz ze wzrostem tych zjawisk rosną również oczekiwania internautów w stosunku do funkcjonalności i wygody użytkowania aplikacji internetowych. Nowoczesne aplikacje powinny być w pełni funkcjonalne, posiadać intuicyjny interfejs, a przede wszystkim być bezpieczne i przejrzyste.

Informacja w internecie jest przekazywana na wiele sposobów. Najpopularniejsze to strony i serwisy WWW zawierające tekst, grafikę, obraz i dźwięk. Internauta widzi stronę internetową, której wygląd jest generowany przez przeglądarkę internetową na podstawie kodu HTML będącego szkieletem każdej strony WWW. Różnice pojawiają się dopiero w sposobie generowania tego kodu. Istnieje wiele różnych technologii, jednak wynikiem działania każdej z nich jest kod HTML. W internecie znajdują się zarówno

proste, statyczne strony, jak i interaktywne aplikacje. Ponieważ ma miejsce ciągły rozwój internetu, powstają zatem również nowe technologie, które ułatwiają i przyspieszają tworzenie aplikacji WWW.

Współczesny programista powinien znać przynajmniej kilka języków programowania i kilkanaście lub nawet kilkadziesiąt narzędzi programistycznych. Do tworzenia aplikacji webowych niezbędna jest znajomość języków wykonujących kod po stronie klienta (np. HTML, CSS, JavaScript) oraz języków wykonywanych po stronie serwera (np. PHP, Java, C#, Python, Ruby). Do wielu języków powstały frameworki, czyli gotowe szkielety wspomagające i przyspieszające budowę aplikacji.

Prawdopodobnie jednym z najnowszych, a zarazem najszybciej rozwijających się aktualnie narzędzi jest framework ASP.NET MVC firmy Microsoft. Przez wiele lat większość stron bazowała na języku PHP i nadal jest to najpopularniejszy język do tworzenia stron oraz portali internetowych (szczególnie w Polsce). Spowodowane to było brakiem sensownej alternatywy. Od samego początku język PHP był udoskonalany i aktualnie jest już bardzo dojrzałym i dopracowanym językiem. W 2007 r. firma Microsoft udostępniła pierwszą wersję frameworku ASP.NET MVC, który jest oparty na platformie .NET. Platforma .NET wspiera wiele języków programowania, m.in. takie języki jak C# lub Visual Basic. Praktycznie co roku pojawia się nowa wersja frameworku ASP.NET MVC. Obecnie jest to wersja MVC 5, a od pewnego czasu trwają już prace nad kolejną wersją MVC 6.

ASP.NET MVC nie jest językiem programowania, lecz jedną z gałęzi całej platformy .NET służącej do tworzenia „dużych” portali internetowych. Framework MVC opiera się na wzorcu „model, widok, kontroler”, który bardzo dobrze wpisuje się w specyfikację sieci Web i żądań HTTP, przez co staje się prawdopodobnie najlepszą bazą do utworzenia nowoczesnego portalu internetowego. Aby strona WWW była nowoczesna, musi bazować na najnowszych trendach, być zoptymalizowana pod wyszukiwarki, powiązana z największymi portalami internetowymi oraz posiadać nowoczesny design.

Książka została napisana z myślą o przyszłości i bazuje wyłącznie na najnowszych wersjach narzędzi. Zarówno język C#, framework ASP.NET MVC, jak i Entity Framework zostały opisane z podziałem na kolejne wersje i z wyszczególnieniem wprowadzanych nowości. Takie podejście wpływa na szybsze przyswojenie kompletnej wiedzy. „Na początku było tak i tak, ale teraz mogę to zrobić łatwiej za pomocą tego i tego”. Dodatkowym atutem takiego podejścia jest możliwość odszukania w książce informacji na temat technologii w dowolnej wersji. Rozpoczęcie nauki od najnowszej wersji bez znajomości wcześniejszych kojarzy się z polskim mało kulturalnym powiedzeniem (w wersji „lajt”): „od d...(ruej) strony”, co oznacza „od końca”.

W obecnych czasach zmienia się podejście do wartości posiadanej wiedzy. Większą wartością jest to, czy ktoś wie, w jaki sposób działa pewien element systemu, jakie ma możliwości i jaki ma związek z pozostałymi elementami, niż to, że wie, w jaki sposób wykonać konkretną czynność. W dobie internetu w kilka sekund można odnaleźć informacje i przykładowy kod, jak wykonać dane zadanie, dlatego dużo ważniejsza jest wiedza, jakiego narzędzia użyć, aby w jak najprostszy i najtańszy sposób osiągnąć

zamierzony cel. Nie ma możliwości ani sensu zapamiętywania wszystkich metod do określonych klas, ponieważ za kilka lat większość z nich będzie już nieaktualna. Ta książka nie koncentruje się na tym, jak zmienić kolor z szarego na zielony, tylko jakie są możliwości i czego użyć, aby móc to zrobić, i to nie tylko z szarego na zielony, ale również dla innych konfiguracji kolorów.

Książka jest adresowana do czytelników mających pewną wiedzę ogólną na temat programowania i baz danych. Dużym udogodnieniem będzie również posiadanie podstawowej wiedzy na tematy związane z omawianymi językami, co pomoże w szybszym zrozumieniu zagadnień i przykładów. W książce starano się opisać wszystko od podstaw, jednak można się domyślać, że ze względu na rozległość tematu pewne zagadnienia pominięto. Nie należy tej pozycji traktować jako szczegółowego opisu z wieloma przykładami dla każdego zagadnienia. W książce zawarto odnośniki do dokumentacji, dzięki której można się szybko zapoznać z większą liczbą szczegółów na dany temat.

Lista zagadnień opisanych w książce, tak jak lista wymagań stawianych programistom webowym, jest bardzo długa. Najważniejsze z nich to: C# 6.0, ASP.NET MVC 5, Entity Framework 6, SEO — pozycjonowanie w Google, Web API 2, routing, ASP.NET Identity, OWIN, JavaScript, jQuery, JSON, JSONP, AJAX, HTML 5, CSS 3, HTML DOM, HTTP, SSL, REST, CORS, relacyjne bazy danych (RDBSM), bazy nierelacyjne (NoSQL), wzorce projektowe i architektoniczne z naciskiem na wzorzec MVC, metodyologie i zasady w programowaniu, Scrum, TDD, Visual Studio 2013, SQL Server 2014, Windows Server 2012, .NET Framework, chmura Windows Azure, LINQ, ADO.NET, cachowanie (użycie pamięci podręcznej), bezpieczeństwo, licencjonowanie czy ceny narzędzi.

Książka pisana jest z punktu widzenia startupowca¹, dlatego porusza bardzo dużo tematów i łączy je w jedną całość. Po zapoznaniu się z teorią na poszczególne tematy zamieszczoną w kolejnych rozdziałach na końcu opisano kolejne kroki tworzenia nowoczesnego portalu internetowego z ogłoszeniami. Nauczysz się korzystać z Visual Studio z wykorzystaniem skrótów klawiszowych i snippetów, dowiesz się, jak debugować aplikację, jak tworzyć projekty i dodawać pomiędzy nimi referencje, oraz zostaniesz prowadzony przez kompletny proces przebudowy aplikacji pod względem architektonicznym na system warstwowy i wymienny dzięki usunięciu zależności pomiędzy warstwami za pomocą kontenera IoC (odwrócenie zależności). Aplikacja wykorzystuje najnowsze narzędzia i łączy większość opisanych zagadnień i języków w jedną całość. Została ona zoptymalizowana pod wyszukiwarkę Google i powiązana z portalem Facebook. Ponadto utworzona została funkcjonalność Web serwisu, dzięki któremu będzie można pobierać dane w formatach JSON lub XML poprzez urządzenia mobilne bądź aplikacje desktopowe, które nie będą miały bezpośredniego dostępu do bazy danych.

Wszystkie kody etapów i kroków dostępne są w archiwum na serwerze FTP wydawnictwa Helion pod adresem <ftp://ftp.helion.pl/przyklady/c6mvc5.zip>. Działająca aplikacja została uruchomiona pod adresem [AspNetMvc.pl²](AspNetMvc.pl).

¹ Osoba lub firma, w tym przypadku w branży informatycznej, związana z nowymi technologiami, która przy niskich kosztach działalności, zwiększonym ryzyku oraz relatywnie wyższym zwrotcie z inwestycji realizuje pewne przedsięwzięcie programistyczne, np. w postaci e-usługi.

² Profil książki na Facebooku: <https://www.facebook.com/CSharp6MVC5>.

Rozdział 1.

C# — teoria i praktyka

W niniejszym rozdziale będą przedstawione najbardziej istotne elementy języka C#. Opisane zostaną także bardzo szczegółowe i potrzebne, ale zarazem zazwyczaj niebrane pod uwagę aspekty języka. Rozdział nie wyczerpuje informacji na temat języka C# — ma tylko zwrócić uwagę na najważniejsze zagadnienia.

Aby zrozumieć dalszą część książki, niezbędna jest podstawowa wiedza z zakresu języka C#.

Wprowadzenie do języka C#

Język C# to nowoczesny obiektowy język programowania. Jest to język ściśle typowany, co oznacza, że każda zmienna musi być konkretnego typu i nie może zmieniać swojego typu w trakcie działania programu (z małym wyjątkiem, jakim jest typ dynamic). Kod języka C# jest komplikowany do kodu pośredniego CIL (ang. *Common Intermediate Language*), który można uruchomić na różnych maszynach wirtualnych. C# jest częścią platformy .NET, która pozwala na użycie języka C# w praktycznie wszystkich możliwych zastosowaniach. Automatycznym zarządzaniem i zwalnianiem pamięci zajmuje się *Garbage Collector* („odśmiecacz”), który usuwa nieużywane zasoby z pamięci operacyjnej. Nie ma potrzeby pisania destruktów, jak w języku C lub C++, ale w określonych sytuacjach istnieje taka możliwość. Pierwsza wersja języka C# została wydana w 2000 r. na bazie języków, które powstały wcześniej, takich jak C++, Java, Delphi czy Smalltalk. Język C# jest ciągle rozwijany i regularnie wychodzą jego nowe wersje.

Kolejne wersje języka C#

C# 2.0 (.NET Framework 2.0, Visual Studio 2005)

Rozszerzenia zaimplementowane w C# 2.0:

- ◆ typy generyczne (ang. *Generics*) — polimorfizm parametryczny (nie precyzujemy typów, na jakich operujemy),
- ◆ typy częściowe (ang. *Partial Types*) — rozbicie klasy, interfejsu lub struktury do wielu plików,
- ◆ metody anonimowe (ang. *Anonymous Methods*),
- ◆ delegaty (ang. *Delegates*),
- ◆ iteratory i instrukcja `yield return`,
- ◆ typ `Nullable`,
- ◆ prywatne settery (np. `public int Id {get; private set;}`).

C# 3.0 (.NET Framework 3.5, Visual Studio 2008)

Rozszerzenia zaimplementowane w C# 3.0:

- ◆ słowo kluczowe `var`, zmienne domniemane — niejawnia deklaracja typów (ang. *Implicitly Typed Local Variables*),
- ◆ inicjalizatory kolekcji i obiektów (ang. *Object and Collection Initializers*),
- ◆ inicjalizatory tablic (ang. *Implicitly Typed Arrays*),
- ◆ automatycznie implementowane pola `get` i `set` (ang. *Auto-implemented Properties*),
- ◆ typy anonimowe (ang. *Anonymous Types*) — pozwalają tworzyć obiekty klas, których deklaracja nie istnieje,
- ◆ metody rozszerzające (ang. *Extension Methods*) — pozwalają rozszerzyć funkcjonalność typów już istniejących; mogą być zadeklarowane tylko w klasach statycznych,
- ◆ wyrażenia lambda (ang. *Lambda Expressions*),
- ◆ drzewa wyrażeń (ang. *Expression Trees*),
- ◆ metody częściowe — możliwość rozbicia metody do kilku plików (ang. *Partial Methods*),
- ◆ LINQ oraz nowe słowa kluczowe `select`, `from` i `where`.

C# 4.0 (.NET Framework 4.0, Visual Studio 2010)

Rozszerzenia zaimplementowane w C# 4.0:

- ♦ dynamiczne wiązania (ang. *Dyn3amic Binding*),
- ♦ argumenty nazwane i parametry opcjonalne (ang. *Named and Optional Arguments*),
- ♦ obsługa kontra- i kowariancji (ang. *Generic Co- and Contra Variance*),
- ♦ opóźniona inicjalizacja (ang. *Lazy Initializers*).

C# 5.0 (.NET Framework 4.5, Visual Studio 2012 oraz 2013)

Rozszerzenia zaimplementowane w C# 5.0:

- ♦ metody asynchroniczne (ang. *Asynchronous Methods*) — `async`, `await`,
- ♦ nowe atrybuty wywołania (ang. *Caller Info Attributes*) — pozwalają uzyskać informacje o wywołaniu metod.

C# 6.0 (zapowiedź)

Rozszerzenia, które prawdopodobnie będą lub, już w trakcie pisania tej książki, zostaną zaimplementowane w C# 6.0:

- ♦ konstruktory pierwotne (ang. *Primary Constructors*),
- ♦ automatyczna inicjalizacja właściwości (ang. *Initializers for Auto-properties*),
- ♦ `using` dla składowych statycznych (ang. *Using Static Members*),
- ♦ inicjalizatory słownikowe (ang. *Dictionary Initializers*),
- ♦ deklaracje `inline` dla parametrów `out` (ang. *Inline Declarations for Out Params*),
- ♦ wyrażenia dla właściwości (ang. *Property Expressions*),
- ♦ wyrażenia dla metod (ang. *Method Expressions*),
- ♦ modyfikator `private protected`,
- ♦ kolekcje `IEnumerable` jako parametr (ang. *Params for Enumerable*),
- ♦ jednoargumentowe sprawdzanie wartości `null` (ang. *Monadic Null Checking*),
- ♦ `await` w blokach `catch` i `finally`,
- ♦ filtry wyjątków (ang. *Exception Filters*),
- ♦ literały binarne i separatory cyfr (ang. *Binary Literals and Digit Separators*).

Konwencje

Styl kodowania (konwencja kodowania) to podstawy niezbędne, aby móc pisać dobry kod w danym języku lub środowisku. W różnych językach programowania oraz frameworkach korzysta się z różnych konwencji, dlatego już na samym początku należy

się zapoznać z konwencjami obowiązującymi w danym języku czy framework'u. Istnieją także konwencje kodowania niezwiązane z konkretnym językiem czy frameworkiem, jak np. notacja węgierska. Notacja węgierska to sposób zapisu nazw zmiennych oraz obiektów, polegający na poprzedzaniu właściwej nazwy małą literą (literami), określającą rodzaj tej zmiennej (obiektu).

Przykładowe zmienne zapisane w notacji węgierskiej:

- ◆ `i_Liczba` — liczba typu `int`,
- ◆ `by_Zmienna` — zmienna typu `byte`, `unsigned char`.

Notacja ta obecnie nie odgrywa już znaczącej roli, a nawet jest uważana za złą praktykę, ponieważ w nowoczesnych środowiskach programistycznych, takich jak np. Visual Studio, bardzo łatwo można sprawdzić, jakiego typu jest dana zmienna.

Cel stosowania konwencji

Konwencje mają na celu:

- ◆ zachowanie czytelności kodu — konwencje tworzą spójny wygląd kodu, czytelnicy mogą się skupić na treści, a nie układzie,
- ◆ utrudnienie popełnienia pomyłek,
- ◆ usuwanie niejednoznaczności, których nie widać na pierwszy rzut oka,
- ◆ informowanie o celu zastosowania kodu,
- ◆ szybsze zrozumienie kodu poprzez założenia oparte na wcześniejszych doświadczeniach,
- ◆ ułatwienie kopiowania, modyfikowania i utrzymywania kodu.

Pliki a klasy i interfejsy

Zalecenia w odniesieniu do plików:

- ◆ każda klasa lub interfejs w osobnym pliku,
- ◆ nazwa pliku powinna być taka sama jak nazwa klasy bądź interfejsu,
- ◆ nie należy przekraczać 2000 linii kodu w jednym pliku.

Wcięcia

Wcięcia w kodzie powinny być wstawiane za pomocą klawisza `Tab`, a nie spacji. Oszczędza to czas potrzebny na kilkakrotne naciśnięcie klawisza spacji i gwarantuje równe odstępy.

Kiedy wyrażenie nie kończy się w pojedynczej linii, stosuje się przełamanie (przejście do kolejnej linii) zgodnie z poniższymi zasadami:

- ♦ przełamanie po przecinku,
- ♦ przełamanie po operatorze,
- ♦ wyrównuj nową linię z początkiem wyrażenia przy tym samym poziomie w poprzedniej linii.

Prawidłowo sformatowany kod:

```
public class DbContextFactory : IDbContextFactory
{
    private readonly DbContext c;
    public DbContextFactory()
    {
        c = new RepoContext();
    }
    public DbContext GetContext()
    {
        return c;
    }
}
```

Nieprawidłowo sformatowany kod:

```
public class DbContextFactory : IDbContextFactory
{private readonly DbContext c;
    public DbContextFactory(){c = new RepoContext();}
    public DbContext GetContext(){return c; }}
```

Komentarze

Komentarz dotyczący tylko jednej linii wygląda następująco:

```
// Linia 1
```

Komentarz dotyczący wielu linii kodu wygląda następująco:

```
/*
Linia 1
Linia 2
Linia 3
*/
```

Komentarz do dokumentacji — jednoliniowy:

```
/// Komentarz do dokumentacji
```

Komentarz do dokumentacji — wieloliniowy:

```
/*
Komentarz do dokumentacji linia1
Komentarz do dokumentacji linia2
*/
```

Skróty klawiszowe służące do komentowania w Visual Studio:

- ♦ *Ctrl+K*, a następnie *Ctrl+C* — komentarz zaznaczonej części kodu,
- ♦ *Ctrl+K*, a następnie *Ctrl+U* — usunięcie komentarza zaznaczonej części kodu.

Deklaracje klas, interfejsów i metod

Przy pisaniu klas, interfejsów czy metod w języku C# należy stosować się do poniższych zasad:

- ◆ brak spacji między nazwą metody a nawiasami (zaczynającymi listę parametrów,
- ◆ nawiasy otwierające { pojawiają się w kolejnej linii po instrukcji deklaracji,
- ◆ nawias zamykający } zaczyna się w linii wcięcia pasującej do odpowiedniego nawiasu otwierającego.

Przykład prawidłowo sformatowanego kodu prezentuje listing 1.1.

Listing 1.1. Poprawne formatowanie kodu

```
namespace DataRepo
{
    public class DbContextFactory : IDbContextFactory
    {
        private readonly DbContext c;
        public DbContextFactory()
        {
            c = new RepoContext();
        }

        public DbContext GetContext()
        {
            return c;
        }
    }
}
```

Puste linie

Dwie puste linie stosuje się pomiędzy:

- ◆ logicznymi sekcjami pliku źródłowego,
- ◆ definicjami klasy i interfejsu (tylko w przypadku, gdy w jednym pliku znajdują się kilka klas czy interfejsów).

Jedną pustą linię stosuje się pomiędzy:

- ◆ metodami,
- ◆ właściwościami,
- ◆ lokalnymi zmiennymi w metodzie i jej pierwszą instrukcją,
- ◆ logicznymi sekcjami wewnętrz metody dla poprawy czytelności.

Nawiąsy klamrowe

Nawiąsy klamrowe powinny być umieszczone w nowej linii po:

- ♦ deklaracjach przestrzeni nazw,
- ♦ deklaracjach klasy, interfejsu bądź struktury,
- ♦ deklaracjach metod,
- ♦ instrukcjach warunkowych i sterujących.

W pozostałych przypadkach nawias powinien się znajdować w tej samej linii.

Konwencje nazewnicze

Style pisania wielkimi literami:

- ♦ styl Pascal'a — każde słowo w nazwie zaczyna się wielką literą, np. TestController,
- ♦ styl wielbłędzi (ang. *Camel Case*) — każde słowo nazwy za wyjątkiem pierwszego zaczyna się wielką literą, np. testController,
- ♦ wielkie litery — używa się tylko dużych liter dla nazw identyfikatorów składających się z maksimum dwóch liter, np. public const PI.

Nazewnictwo klas:

- ♦ nazwa klasy musi być rzeczownikiem lub grupą rzeczownikową,
- ♦ należy używać stylu Pascal'a,
- ♦ nie należy używać żadnego prefiksu klasy (notacja wegierska).

Nazewnictwo interfejsów:

- ♦ nazwa interfejsu musi być rzeczownikiem, grupą rzeczownikową lub przymiotnikiem — opisuje zachowanie,
- ♦ należy używać stylu Pascal'a,
- ♦ należy używać prefiksu I dla nazw, po których następuje duża litera (pierwszy znak nazwy interfejsu to I, np. IEnumerable).

Nazwy parametrów:

- ♦ należy używać nazw opisowych, które powinny być wystarczające do określenia znaczenia zmiennej i jej typu,
- ♦ należy używać stylu wielbłędziego.

Nazwy zmiennych:

- ♦ zmienne zliczające najczęściej nazywane są literami (i, j, k, l, m, n), kiedy używane są w pętlach zliczających,
- ♦ należy używać stylu wielbłędziego.

Nazwy metod:

- ◆ nazwy metod muszą być czasownikami lub grupą czasownikową,
- ◆ należy używać stylu Pascala.

Nazwy właściwości:

- ◆ nazwy właściwości muszą być rzeczownikami lub grupą rzeczownikową,
- ◆ należy używać stylu Pascala,
- ◆ w niektórych wypadkach zasadne jest nazwanie właściwości tą samą nazwą co jej typ.

Nazwy zdarzeń:

- ◆ nazwy obsługi zdarzeń powinno się podawać z przyrostkiem EventHandler,
- ◆ należy używać dwóch parametrów nazwanych sender i e,
- ◆ powinno się używać stylu Pascala,
- ◆ nazwy argumentu zdarzenia klas należy podawać z przyrostkiem EventArgs,
- ◆ do określenia nazw zdarzeń używa się czasu teraźniejszego lub przeszłego,
- ◆ do określenia nazw zdarzeń można użyć czasowników.

Pozostałe dobre praktyki

Dobre praktyki to również:

- ◆ nazwy metod powinny opisywać akcję,
- ◆ nazwy pól prywatnych klas można zaczynać od znaku _, np. _polePrywatne,
- ◆ klasy zgodnie z regułą pojedynczej odpowiedzialności robią tylko jedną rzecz,
- ◆ należy łączyć deklarację zmiennej z jej inicjalizacją,
- ◆ powinno się szczegółowo komentować publiczne interfejsy, a w przypadku prywatnych ograniczyć się do kilku słów wyjaśnienia, jeżeli znaczenie nie wynika wprost z nazwy klasy lub metody,
- ◆ należy zmniejszać liczbę poziomów zagnieźdżeń instrukcji sterujących w metodach,
- ◆ powinno się unikać zmiennych globalnych i obiektów typu *singleton*, które utrudniają testowalność, sprawiają problemy synchronizacyjne przy kilku wątkach oraz wprowadzają zbędne zależności wewnętrz kodu.

Typy

Typy, które zostały omówione w tabeli 1.1, można podzielić na dwie kategorie: typy wartości (skalarne) i typy referencyjne. Typy referencyjne dziedziczą bezpośrednio po typie object, natomiast typy wartości dziedziczą po klasie ValueType, która z kolei

dziedziczy po object. Podstawową różnicą pomiędzy tymi dwoma rodzajami typów jest sposób przechowywania wartości. Typy skalarne przechowują wartość na stosie, natomiast typy referencyjne przechowują na stosie wskaźnik (adres) do wartości. Na jedną wartość może pokazywać kilka zmiennych typu referencyjnego. Zmieniając wartość typu referencyjnego, zmienia się również wartość dla wszystkich zmiennych pokazujących na tę zmienną¹.

Do typów wartości zalicza się:

- ♦ typy proste,
- ♦ enum — typ wyliczeniowy,
- ♦ struktury.

Do typów referencyjnych zalicza się:

- ♦ typ object,
- ♦ typ string,
- ♦ tablice,
- ♦ delegaty,
- ♦ klasy,
- ♦ interfejsy,
- ♦ zdarzenia.

Deklaracja zmiennej

Każda zmienna używana w programie ma swój typ danych, który określa, jakie wartości mogą być w niej przechowywane. Aby móc skorzystać ze zmiennej danego typu, należy ją zadeklarować. Składnia deklaracji wygląda następująco:

typ identyfikator [=wartosc];

gdzie:

- ♦ typ — określa typ zmiennej (wymagane),
- ♦ identyfikator — nazwa zmiennej (wymagane),
- ♦ wartosc — wartość początkowa zmiennej (opcjonalne).

Deklarując zmienną, należy przede wszystkim pamiętać, że identyfikator (nazwa zmiennej) nie może być identyczny jak słowa kluczowe języka.

¹ <http://msdn.microsoft.com/pl-pl/library/ms173104.aspx>

Tabela 1.1. Tablica typów

Typ	Bajty	Opis
byte	1	8-bitowa liczba całkowita bez znaku
sbyte	1	8-bitowa liczba całkowita ze znakiem
short	2	16-bitowa liczba całkowita ze znakiem
ushort	2	16-bitowa liczba całkowita bez znaku
int	4	32-bitowa liczba całkowita ze znakiem
uint	4	32-bitowa liczba całkowita bez znaku
long	8	64-bitowa liczba całkowita ze znakiem
ulong	8	64-bitowa liczba całkowita bez znaku
float	4	Liczba zmiennoprzecinkowa
double	8	Liczba zmiennoprzecinkowa podwójnej precyzji
decimal	8	Liczba zmiennoprzecinkowa stałej precyzji
string	-	Ciąg znaków Unicode
char	2	Znak Unicode
bool	-	Wartość logiczna
intptr	-	Liczba całkowita ze znakiem Rozmiar zależny od platformy: 32 bity na platformie 32-bitowej 64 bity na platformie 64-bitowej
uintptr	-	Liczba całkowita bez znaku Rozmiar zależny od platformy: 32 bity na platformie 32-bitowej 64 bity na platformie 64-bitowej
enum	-	Typ wyliczeniowy
object	-	Obiekt, z którego pochodzą wszystkie typy danych
struct	-	Typ wartości — struktura
class	-	Klasa
interface	-	Interfejs
delegate	-	Delegaty
event	-	Zdarzenie

Inicjalizacja zmiennej

Wymagana jest inicjalizacja zmiennej określona wartością przed jej użyciem. Inicjalizacja zmiennej może przebiegać na dwa sposoby:

- ◆ przypisanie wartości w czasie deklaracji: `int wartosc = 123;`
- ◆ przypisanie wartości do zadeklarowanej wcześniej zmiennej:

```
int wartosc;           wartosc = 123;
```

Słowa kluczowe

Słowa kluczowe to słowa używane w składni języka i nie powinny być wykorzystywane w inny sposób. W zależności od kontekstu możliwe staje się użycie wybranych słów. Jeśli koniecznie trzeba użyć wybranego słowa kluczowego (jako napis — typ string), należy poprzedzić je znakiem @, np. @int.

Lista zastrzeżonych słów kluczowych języka C# znajduje się w tabeli 1.2.

Tabela 1.2. Zastrzeżone słowa kluczowe

abstract	as	base	bool	break
byte	case	catch	char	checked
class	const	continue	decimal	default
delegat	do	double	else	enum
event	explicit	extern	false	finally
fixed	float	for	foreach	goto
if	implicit	In	int	interface
internal	is	lock	long	namespace
new	null	object	operator	out
override	params	private	protected	public
readonly	ref	return	sbyte	sealed
short	sizeof	stackalloc	static	string
struct	switch	this	throw	true
try	typeof	uint	ulong	unchecked
unsafe	ushort	using	virtual	void
volatile	while			

W tabeli 1.3 znajduje się lista słów kluczowych zależnych od kontekstu. Nie są to słowa zastrzeżone w przypadku użycia w innym kontekście.

Tabela 1.3. Zastrzeżone słowa kluczowe zależne od kontekstu

add	alias	ascending	async	await
descending	dynamic	from	get	global
group	into	join	let	orderby
partial	remove	select	set	value
var	where		yield	

Stałe i zmienne tylko do odczytu

Stałe to dane, które nie mogą być modyfikowane. Aby zdefiniować stałą, używa się słowa kluczowego const. Stała nie może być deklarowana z modyfikatorem static, musi zostać koniecznie zainicjowana podczas deklaracji, ponieważ wartość do zmiennej jest przypisywana w czasie komplikacji programu. Późniejsze zainicjowanie zmiennej, czyli zmiana jej wartości, jest niemożliwe. Stała może być wykorzystana w warunku case instrukcji warunkowej switch. Przykład deklaracji stałej:

```
const int Stala = 10;
```

Zmienne tylko do odczytu (ang. *Read-Only*), podobnie jak stałe, nie mogą być modyfikowane. Inicjalizacja zmiennych tylko do odczytu może mieć miejsce zarówno w deklaracji, jak i w konstruktorze, ponieważ wartość do zmiennej jest przypisywana w trakcie działania (startu) programu, a nie podczas komplikacji. Aby zdefiniować zmienną tylko do odczytu, używa się słowa kluczowego `readonly`. Zmienne `readonly` mogą być poprzedzone modyfikatorem `static`.

Przykład deklaracji zmiennej `readonly`:

```
readonly int TylkoDoOdczytu = 10;
```

Literały

Literał to tekstowa reprezentacja wartości danego typu. Służy do przypisywania wartości dla zmiennych. Podział literałów:

- ◆ Literały logiczne — przeznaczone dla typu `bool`. Zmienna `bool` może mieć wartość `true` (prawda) lub `false` (fałsz): `bool zmienna = false;`.
- ◆ Literały dla liczb całkowitych i zmiennoprzecinkowych:

```
var d = 1.0d; //double
var f = 1.0f; //float
var m = 1.0m; //decimal
var i = 1; //int
var ui = 1U; //uint
var ul = 1UL; //ulong
var l = 1L; //long
```

- ◆ Literały znakowe — są przeznaczone dla typu `char`, a ich wartość podaje się w cudzysłowach. Wykorzystywane są dla znaków szczególnych, aby odróżnić te znaki od znaków odnoszących się do struktury języka programowania. Wybrane znaki szczególne przedstawiono w tabeli 1.4.

Tabela 1.4. Tabela znaków szczególnych

Sekwencja uproszczona	Kodowanie Unicode	Nazwa
\0	0x0000	Null
\a	0x0007	Sygnal dźwiękowy
\b	0x0008	Backspace
\t	0x0009	Tabulacja
\n	0x000A	Nowa linia
\r	0x000C	Powrót na początek linii
\`	0x0022	Cudzysłów
\'	0x0027	Pojedynczy cudzysłów
\\\	0x005C	Backslash

- ♦ Literały łańcuchowe — podstawowym literałem jest @. Pozwala wyłączyć specjalne znaczenie sekwencji znaków i traktuje napis jako zwykły tekst. Ten literal jest przeznaczony tylko dla typu string:

```
string tekst = @"\"typ\""; // Wyświetli tekst: \typ
```

- ♦ Literal null — niedozwolony dla typów wartości, oznacza „pusty”.

Typ wyliczeniowy

enum to typ wyliczeniowy (stałe nazwane), który pozwala korzystać z nazw zamiast z liczb. Inaczej mówiąc, zamienia liczbę na nazwę, co ułatwia korzystanie z określonej funkcjonalności programu². Typ enum nadaje się np. do oznaczania stanów (takich jak dodane, usunięte, edytowane) lub dni tygodnia czy miesięcy. Bez użycia enum konieczna byłaby deklaracja zmiennej np. typu int i dla każdej jej wartości ustalenie znaczenia. Przykładowo jeśli zmienna ma wartość 1, to znaczy, że jest to stan „dodane”, jeśli ma wartość 2 — stan „usunięte” itd. Dzięki enum nie zmienia się wartości, tylko posługuję się nazwami stanów, które są tłumaczone na wartości. Typy dostępne do użycia z enum to: byte, sbyte, short, ushort, int, uint, long lub ulong. Domyślnie odliczanie zaczyna się od 0, jednak w niektórych przypadkach powoduje to problemy. W przedstawionym przykładzie odliczanie rozpoczyna się od 1.

Deklaracja enum:

```
enum Stan {Dodane=1, Usuniete, Edytowane };
```

gdzie:

```
int x = (int)Stan.Usuniete;  
//x = 2  
Stan stan = Stan.Dodane;  
//stan = Dodane
```

Konwersje typów i rzutowanie

Konwersja typów to zamiana wartości jednego typu na wartość drugiego typu³. Rzutowanie typu to określenie, jak ma być traktowana zmienna innego typu. Aby wykonać rzutowanie, podaje się w nawiasie przed zmienną typ, na który ma być zamieniona:

```
int liczbaInt = (int)liczbaDouble; // Rzutowanie na typ int
```

Podział konwersji ze względu na ilość przechowywanych informacji:

- ♦ poszerzające — występuje w przypadku, gdy zmienia się podstawowy typ na typ, który pozwala przechować więcej informacji, np. int na long;
- ♦ przybliżające — występuje w przypadku, gdy zmienia się podstawowy typ na typ, który pozwala przechować mniej informacji, np. double na int.

² <http://msdn.microsoft.com/pl-pl/library/sbbt4032.aspx>

³ <http://msdn.microsoft.com/pl-pl/library/ms173105.aspx>

Typy konwersji dostępne w C#:

- ◆ konwersje niejawne (ang. *Implicit*) — nie wymagają rzutowania (poszerzające);

```
int liczbaInt = 645;
long liczbaLong = liczbaInt;
```

- ◆ konwersje jawne (ang. *Explicit*) — wymagane rzutowanie (przybliżające);

```
double liczbaDouble = 645.8;
int liczbaInt;
liczbaInt = (int)liczbaDouble; // Rzutowanie na typ int
// liczbaInt = 645
```

- ◆ konwersje zdefiniowane przez użytkownika;

- ◆ konwersje z klasy pomocy — konwersje za pomocą metod do konwersji dostępnych w bibliotekach.

Dostępne konwersje niejawne przedstawiono w tabeli 1.5.

Tabela 1.5. Tabela konwersji niejawnych

Z	Na
sbyte	short, int, long, float, double lub decimal
byte	short, ushort, int, uint, long, ulong, float, double lub decimal
short	int, long, float, double lub decimal
ushort	int, uint, long, ulong, float, double lub decimal
int	long, float, double lub decimal
uint	long, ulong, float, double lub decimal
long	float, double lub decimal
char	ushort, int, uint, long, ulong, float, double lub decimal
float	double
ulong	float, double lub decimal

Dostępne konwersje jawne zostały wymienione w tabeli 1.6.

Opakowywanie (boxing) i rozpakowywanie (unboxing)

Mechanizm opakowywania i rozpakowywania polega na zamianie typu wartości na typ referencyjny (`object`), czyli obiekt, i odwrotnie — z obiektu na typ wartości⁴.

Opakowywanie, jak już wspomniano, polega na zamianie typu wartości na typ `object`. Zostaje utworzona nowa instancja obiektu, a następnie do tego obiektu kopiwana jest wartość zmiennej:

```
int iLiczba = 456;
object oLiczba = iLiczba;
```

⁴ <http://msdn.microsoft.com/pl-pl/library/yz2be5wk.aspx>

Tabela 1.6. Tabela konwersji jawnych

Z	Na
sbyte	byte, ushort, uint, ulong lub char
byte	sbyte lub char
short	sbyte, byte, ushort, uint, ulong lub char
ushort	sbyte, byte, short lub char
int	sbyte, byte, short, ushort, uint, ulong lub char
uint	sbyte, byte, short, ushort, int lub char
long	sbyte, byte, short, ushort, int, uint, ulong lub char
ulong	sbyte, byte, short, ushort, int, uint, long lub char
char	sbyte, byte lub short
float	sbyte, byte, short, ushort, int, uint, long, ulong, char lub decimal
double	sbyte, byte, short, ushort, int, uint, long, ulong, char, float lub decimal
Decimal	sbyte, byte, short, ushort, int, uint, long, ulong, char, float lub double

Rozpakowywanie jest operacją odwrotną do operacji opakowywania. Zmienna z obiektu zostaje przypisana do nowej zmiennej. Należy pamiętać, że zmienna musi mieć właściwy typ wartości, taki sam jak przed przypisaniem do obiektu. W przeciwnym wypadku zostanie zwrócony wyjątek (`InvalidOperationException`):

```
int iLiczba = 456;
object oLiczba = iLiczba;
int iNowaLiczba = (int)oLiczba;
```

Operacje opakowywania i rozpakowywania są wykonywane w standardowych kolekcjach (nietypowanych), jednak jest to kosztowny proces, przez co kolekcje nietypowane są dużo wolniejsze niż kolekcje typowane (generyczne). Należy unikać używania tych operacji, jeśli nie ma wyraźnej potrzeby.

Wartości zerowe oraz typy dopuszczające wartości zerowe

Wartość `null` oznacza nieokreśloną wartość, czyli zmienną nieposiadającą przypisanej żadnej wartości. Typy `Nullable` to instancje struktury `System.Nullable<T>`, gdzie `T` to typ wartości (np. `int`, `char`, `bool`) i `T` nie może być typem referencyjnym (listing 1.2). Typ `string` jest strukturą, dlatego domyślnie może przyjmować wartość `null`. Dla pozostałych typów należy wskazać, że mogą przyjmować wartość `null`.

Listing 1.2. Oznaczanie zmiennych jako `Nullable`

```
Nullable<int> liczba = null;
//Lub
int? liczba = null;
//Nullable dla różnych typów
double? d = 2.25;
bool? flaga = null; //Zmienna typu bool? posiada 3 dopuszczalne wartości: true, false i null
char? znak = 'z';
int?[] tablica = new int?[5];
```

Każdy typ oznaczony jako `Nullable` posiada dodatkowe właściwości:

- ◆ `HasValue` — jeśli zmienna ma wartość `null`, to zwraca `false`; w przeciwnym wypadku zwraca `true` (listing 1.3);
- ◆ `Value` — ta właściwość jest tego samego typu jak zmienna, jednak nie `Nullable`; jeśli zmienna ma przypisaną wartość, to jest ona zwracana, a jeśli zmienna jest pusta, czyli ma wartość `null`, to zostaje wywołany wyjątek `InvalidOperationException`.

Listing 1.3. Przykład użycia właściwości dla typów `Nullable`

```
int? liczba = 10;
if (liczba.HasValue)      // Właściwość HasValue
{
    Console.WriteLine(liczba.Value); // Właściwość Value
}
```

Jeśli w argumentach do operatora (unarnego lub binarnego) znajduje się choć jedna zmienna, która ma wartość `null`, to wynik działania będzie miał wartość `null` (nawet jeśli jest to dodawanie). Jeśli jeden element do porównania ma wartość, a drugi nie posiada wartości, czyli ma wartość `null`, to wynikiem porównania zawsze będzie `false`. Jedynie operator `!=` zwróci wartość `true`⁵.

Typy generyczne

Typy generyczne (ang. *Generics*) zostały wprowadzone w wersji C# 2.0. W zależności od preferencji nazywane są również typami ogólnymi lub typami szablonowymi. Typy generyczne pozwalają umieścić w kolekcji dane tego samego, jednak nieznanego typu. Typ danych określa się dopiero przy ich tworzeniu. Dzięki wykorzystaniu typów generycznych nie trzeba za każdym razem wykonywać opakowywania, a następnie wypakowywania, co pozwala zaoszczędzić moc obliczeniową.

Tablice, łańcuchy i kolekcje

Tablice

Tablice to zbiory wielu zmiennych tego samego typu. W tablicach oprócz typów wbudowanych można przechowywać również klasy i struktury. Każda tablica dziedziczy po klasie `System.Array`, która dostarcza podstawowych mechanizmów do manipulacji na elementach tablicy. Do deklaracji tablicy służy operator `[]`⁶.

Tablice jednowymiarowe, jak sama nazwa wskazuje, to tablice, które mają tylko jeden wymiar. Deklaracja tablicy jednowymiarowej wygląda następująco:

```
string[] nazwa;
int[] liczba;
```

⁵ <http://msdn.microsoft.com/pl-pl/library/1t3y8s4s.aspx>

⁶ <http://msdn.microsoft.com/pl-pl/library/system.array.aspx>

Poniżej przedstawiono inicjalizację tablicy jednowymiarowej:

```
nazwa = new string[2] { "nazwa1", "nazwa2" };
liczba = new int[3] {1, 2, 3};
```

Deklarację z inicjalizacją tablicy jednowymiarowej pokazano poniżej:

```
string[] nazwa = new string[2] { "nazwa1", "nazwa2" };
int[] liczba = new int[3] {1, 2, 3};
```

Tablice wielowymiarowe, jak sama nazwa wskazuje, to tablice, które mają więcej niż jeden wymiar. Deklaracja tablicy wielowymiarowej wygląda następująco:

```
string[,] nazwa;
int[,] liczba;
```

Poniżej zaprezentowano inicjalizację tablicy wielowymiarowej:

```
nazwa = new string[2,2] { { "nazwa1", "nazwa2" }, { "nazwa1", "nazwa2" } };
liczba = new int[3, 3] { { 1, 2, 3 }, { 1, 2, 3 }, { 1, 2, 3 } };
```

Deklaracja z inicjalizacją tablicy wielowymiarowej wygląda jak poniżej:

```
string[,] nazwa = new string[2,2] { { "nazwa1", "nazwa2" }, { "nazwa1",
    "nazwa2" } };
int[,] liczba = new int[3, 3] { { 1, 2, 3 }, { 1, 2, 3 }, { 1, 2, 3 } };
```

Właściwości dostępne w klasie `System.Array` zostały wymienione w tabeli 1.7.

Tabela 1.7. Właściwości klasy `System.Array`

Nazwa	Opis
<code>IsFixedSize</code>	Wartość wskazująca, czy tablica ma stały rozmiar
<code>IsReadOnly</code>	Wartość wskazująca, czy tablica jest tylko do odczytu
<code>IsSynchronized</code>	Wartość wskazująca, czy dostęp do tablicy jest synchronizowany (ang. <i>Thread Safe</i>)
<code>Length</code>	Wartość (32-bitowa liczba całkowita) reprezentująca liczbę elementów we wszystkich wymiarach tablicy
<code>LongLength</code>	Wartość (64-bitowa liczba całkowita) reprezentująca liczbę elementów we wszystkich wymiarach tablicy
<code>Rank</code>	Liczba wymiarów tablicy
<code>SyncRoot</code>	Pobiera obiekt, który może być użyty do synchronicznego dostępu do tablicy

Najważniejsze metody dostępne w klasie `System.Array`:

- ◆ `BinarySearch()` — przeszukuje tablice,
- ◆ `Clear()` — ustawia zakres elementów w tablicy na zero, na `false` lub `null`, w zależności od typu,
- ◆ `Clone()` — tworzy płytka kopię tablicy,
- ◆ `ConvertAll<TInput, TOutput>` — konwertuje tablicę jednego typu na tablicę innego typu,

- ◆ `Find<T>` — pobiera element odpowiadający warunkom określonym poprzez wskazany predykat,
- ◆ `GetLength()` — pobiera liczbę elementów w określonym wymiarze tablicy,
- ◆ `GetValue()` — pobiera wartość z określonego położenia w tablicy jednowymiarowej,
- ◆ `IndexOf(Array, Object)` — wyszukuje obiekt i zwraca indeks pierwszego wystąpienia w tablicy,
- ◆ `Resize<T>` — zmienia liczbę elementów tablicy do nowo podanego rozmiaru,
- ◆ `Reverse(Array)` — odwraca kolejność elementów w tablicy jednowymiarowej,
- ◆ `Sort()` — sortuje elementy w tablicy.

Łańcuchy

Łańcuchy, a więc ciągi znaków w języku C#, są typu `string`. Ciągi znaków są przechowywane jako sekwencyjne, tylko do odczytu kolekcje obiektów typu `char` w standardzie Unicode. W języku C# słowo kluczowe `string` jest aliasem do `System.String`, a więc obydwa zapisy są równoważne⁷. Możliwe jest odwołanie się do konkretnego znaku w łańcuchu za pomocą operatora indeksowania:

```
string napis = "Tekst";
if (napis[1] == 'k')
{
    ZrobCos();
}
```

Ponieważ obiekt typu `string` jest niezmienny, podczas operacji na łańcuchach za każdym razem tworzony jest nowy obiekt typu `string`. Poniżej wymieniono właściwości dostępne w klasie `System.String`:

- ◆ `Chars` — pobiera znak z określonego miejsca w łańcuchu,
- ◆ `Length` — zwraca liczbę znaków.

Najważniejsze metody dostępne w klasie `System.String` to:

- ◆ `Compare()` — porównuje łańcuchy znaków;
- ◆ `Concat()` — łączy łańcuchy znaków;
- ◆ `Copy()` — kopiuje łańcuch znaków;
- ◆ `Equals()` — określa, czy łańcuchy są takie same;
- ◆ `IndexOf()` — zwraca pozycję w łańcuchu dla konkretnego znaku;
- ◆ `IsNullOrEmpty()` — sprawdza, czy podany ciąg znaków jest pusty;

⁷ <http://msdn.microsoft.com/pl-pl/library/ms228362.aspx>, <http://msdn.microsoft.com/pl-pl/library/system.string.aspx>

- ◆ PadLeft() — tworzy nowy ciąg, który wyrównuje znaki do prawej, a po lewej stronie dopełnia łańcuch określonymi znakami;
- ◆ PadRight() — tworzy nowy ciąg, który wyrównuje znaki do lewej, a po prawej stronie dopełnia łańcuch określonymi znakami;
- ◆ Remove() — zwraca nowy pusty ciąg;
- ◆ Split() — dzieli łańcuch na tablice ciągów określonej długości;
- ◆ ToLower() — konwertuje ciąg znaków na małe litery;
- ◆ ToUpper() — konwertuje ciąg znaków na wielkie litery;
- ◆ Trim() — usuwa znaki spacji z początku i końca łańcucha;
- ◆ ToCharArray() — kopiuje znaki z łańcucha do tablicy znaków.

Kolekcje

Kolekcje można porównać do tablic, jednak kolekcje zamiast przetrzymywać typy wartościowe, przetrzymują elementy kolekcji jako obiekty. Tablica przechowuje dane określonego typu, natomiast kolekcja przetrzymuje wszystko jako obiekty. Nie trzeba podawać typu, więc za każdym razem musi zostać wykonane rzutowanie. Kolekcje, w przeciwieństwie do tablic, nie mają stałego rozmiaru — w zależności od potrzeb zmieniają go dynamicznie (po przekroczeniu rozmiaru kolekcji jest on dwukrotnie zwiększany)⁸.

Kolekcja `ArrayList` jest bardzo podobna do tablic i można ją traktować jako ich ulepszenie. W przeciwieństwie do tablic pozwala usuwać lub dodawać elementy w środku zbioru, przy czym pozostałe elementy zostają przesunięte⁹. Wybrane metody dostępne w `ArrayList` to:

- ◆ Add() — dodaje obiekt na końcu kolekcji `ArrayList`;
- ◆ Remove() — usuwa wybrany element z kolekcji;
- ◆ Insert() — dodaje element do kolekcji pod określonym indeksem;
- ◆ Clear() — czyści zawartość kolekcji;
- ◆ ToArray() — kopiuje zawartość kolekcji do tablicy obiektów;
- ◆ IndexOf() — wyszukuje indeks wybranego elementu;
- ◆ RemoveRange() — kasuje elementy z określonego przedziału;
- ◆ Sort() — sortuje elementy;
- ◆ Reverse() — odwraca kolejność elementów kolekcji.

Kolekcja `HashTable` pozwala na użycie innych indeksów niż liczby całkowite. Składa się z dwóch tablic, z których jedna tablica przechowuje klucze, a druga wartości przypisane

⁸ <http://msdn.microsoft.com/pl-pl/library/System.Collections.aspx>

⁹ <http://msdn.microsoft.com/pl-pl/library/system.collections.arraylist.aspx>

tem kluczom. Zarówno klucz, jak i wartość są obiektami. Klucze nie mogą się powtarzać, dlatego została zaimplementowana metoda `ContainsKey()`, która sprawdza, czy dany klucz już istnieje. W kolekcji `HashTable` nie ma typowania, dlatego można używać różnych typów w obrębie jednej kolekcji. Kolekcja `HashTable` jest bezpieczna wątkowo (ang. *Thread Safety*), a więc można z niej korzystać w aplikacji wielowątkowej. Wiele wątków może w tym samym czasie odczytywać dane z kolekcji, jednak tylko jeden wątek może do niej zapisywać¹⁰.

Kolekcja `SortedList` działa podobnie do `HashTable`, jednak wartości w kolekcji są sortowane po kluczach. Podczas umieszczania elementów w kolekcji automatycznie zostaje on wstawiony w odpowiednim miejscu¹¹.

Kolekcja `Queue` działa według mechanizmu FIFO (ang. *First In, First Out* — pierwszy na wejściu, pierwszy na wyjściu). Elementy dodawane do kolejki zawsze zostają umieszczone na końcu, natomiast elementy pobierane znajdują się na początku. Za pomocą metody `Enqueue()` dodaje się elementy do kolejki, natomiast za pomocą metody `Dequeue()` pobiera się elementy z kolejki¹².

Kolekcja `Stack` (stos) działa według mechanizmu LIFO (ang. *Last In, First Out* — ostatni na wejściu, pierwszy na wyjściu). Elementy są zarówno dodawane, jak i pobierane ze spodu stosu. Ostatnio dodany element będzie pierwszym elementem do pobrania ze stosu. Za pomocą metody `Push()` dodaje się elementy do stosu, natomiast za pomocą metody `Pop()` wyciąga się elementy ze stosu¹³.

Operatorы

Operatorы делятся на две группы¹⁴:

- ◆ unarne — оперируют только на одной переменной, например `-8`, `liczba++`,
- ◆ binarne — оперируют на двух переменных, например `2*3`, `liczba+=6`.

Подział operatorów ze względu na ich przeznaczenie prezentuje tabela 1.8.

Подział operatorów ze względu na priorytet prezentuje tabela 1.9.

Operator trójargumentowy ?:

Operator trójargumentowy `?:` to instrukcja działająca prawie identycznie jak konstrukcja `if else`, z tą różnicą, że jest to operator, a nie instrukcja sterująca, co powoduje, że kompilator stosuje inne reguły walidacji. Skracając znacznie zapis, jednak jest mniej czytelna niż konstrukcja `if else`.

¹⁰ <http://msdn.microsoft.com/pl-pl/library/system.collections.hashtable.aspx>

¹¹ <http://msdn.microsoft.com/pl-pl/library/system.collections.sortedlist.aspx>

¹² <http://msdn.microsoft.com/pl-pl/library/system.collections.queue.aspx>

¹³ <http://msdn.microsoft.com/pl-pl/library/system.collections.stack.aspx>

¹⁴ <http://msdn.microsoft.com/en-us/library/6a71f45d.aspx>

Tabela 1.8. Przeznaczenie operatorów

Kategoria operatora	Operatory
Arytmetyczne	+, -, *, /, %
Logiczne bitowe	&, , ^, ~
Logiczne warunkowe	&&, , !
Konkatenacja łańcuchów	+
Jednostkowego zwiększenia i zmniejszania	++, --
Przesunięcia	<<, >>
Relacji	==, !=, <, >, <=, >=
Przypisania	=, +=, -=, *=, /=, %=, &=, =, ^=, <<=, >>=
Dostępu do składnika klasy	.
Indeksowania	[]
Rzutowania i grupowania	()
Informacje o typie	as, is, sizeof, typeof
Tworzenia obiektu	New
Wyrażenie warunkowe	?:

Tabela 1.9. Priorytety operatorów

Kategoria	Operator	Wiązanie
Nadrzędne	., (), [], x++, x--, new, typeof	Od lewej do prawej (., (), [])
		Od prawej do lewej (x++, x--, new, typeof)
Unarne	(typ), +, -, !, ~, ++x, --x	Od prawej do lewej
Dzielenie, mnożenie, reszta z dzielenia	/, *, %	Od lewej do prawej
Dodawanie, odejmowanie	+, -	Od lewej do prawej
Relacje	<, >, <=, >=, is, as	Od lewej do prawej
Równość, różnica	==, !=	Od lewej do prawej
Iloczyn bitowy	&	Od lewej do prawej
Suma bitowa		Od lewej do prawej
Różnica symetryczna	^	Od lewej do prawej
Iloczyn warunkowy	&&	Od lewej do prawej
Suma warunków		Od lewej do prawej
Wyrażenie warunkowe	?:	Od prawej do lewej

Składnia wyrażenia wygląda następująco:

[wyr_logiczne] ? [wykonaj to, gdy wyrażenie jest prawdziwe] : [wykonaj to, gdy nieprawdziwe]

Przykład:

```
int a = 1;
int b = 8;
int min = a < b ? a : b;
// Wynik: min = 1;
```

Operator ??

Operator ?? zwraca wartość znajdująca się po lewej stronie operatora, gdy jest ona różna od null; w przeciwnym wypadku zwraca wartość znajdująca się po prawej stronie operatora. Przykład:

```
int? a = null; // Znak ? oznacza, że wartość typu int może być null
int b = 8;
int min = a ?? b; // Wynik: min = 8, ponieważ a == null
```

Gdyby zmienna a była różna od null (czyli byłaby liczbą), to zmienna min miałyby przypisaną wartość zmiennej a. Jeśli zmienna a ma wartość null, to zmienna min będzie miała przypisaną wartość zmiennej b.

Instrukcje sterujące

Do instrukcji warunkowych¹⁵ należą instrukcje:

- ◆ if,
- ◆ switch.

Instrukcja if

Budowa instrukcji if jest bardzo prosta, dlatego nie będzie szczegółowo opisywana (listing 1.4).

Listing 1.4. Kod przykładowej instrukcji if

```
int a = 0;
if (a == 0)
{
    // Kod
}
else
{
    // Kod2
}
```

W wyrażeniu kaskadowym (listing 1.5) najpierw sprawdzana jest pierwsza instrukcja if. Gdy jest fałszywa, wykonywane jest kolejne wyrażenie else if w kolejności, w jakiej zostały podane. Gdy żaden z warunków sprawdzanych przez kolejne wyrażenie else if nie będzie prawdziwy, zostanie wykonany ostatni zapis else (jeśli takowy istnieje).

¹⁵ <http://msdn.microsoft.com/pl-pl/library/676s4xab.aspx>

Bardzo ważne, aby instrukcje były podane w kolejności od najbardziej prawdopodobnych do najmniej prawdopodobnych. Taka kolejność pozwala bowiem na zaoszczędzenie mocy obliczeniowej procesora.

Listing 1.5. Kaskadowe wyrażenie if

```
int a = 2;
if (a == 0)
{
    // Kod
}
else if (a == 1)
{
    // Kod2
}
else if (a == 2)
{
    // Kod, który się wykona
}
else
{
    // Kod wykonywany, gdy żaden warunek nie jest spełniony
}
```

Konstrukcję `if` można tworzyć za pomocą snippetu `if`. Należy wpisać słowo `if`, a następnie kliknąć dwukrotnie klawisz `Tab`, natomiast konstrukcję `else` można tworzyć przy użyciu snippetu `else`. Wpisz słowo `else`, a następnie kliknij dwukrotnie klawisz `Tab`.

Instrukcja switch

Instrukcja `switch` pozwala na sprawdzenie wielu warunków w zależności od wartości zmiennej. Składnia instrukcji `switch` jest następująca:

```
switch (wyrażenie_sterujące)
{
    case wartość_sekcji_1:
        instrukcje_1;
        break;

    case wartość_sekcji_2:
        instrukcje_2;
        break;

    ...
}

default:
    instrukcja_domyslna;
```

gdzie:

- ♦ `wyrażenie_sterujące` — pole wymagane,
- ♦ `wartość_sekcji_1, wartość_sekcji_2` — wartość zwracana przez wyrażenie sterujące,

- ◆ instrukcje_1, instrukcje_2 — instrukcje przypisane do sekcji (opcjonalne),
- ◆ default — sekcja jest wykonywana w przypadku braku pasujących sekcji (opcjonalne).

Słowo kluczowe `break` informuje o zakończeniu instrukcji (listing 1.6). W przypadku gdy po danej instrukcji trzeba wykonać jeszcze jedną instrukcję, stosuje się instrukcję `goto case`. Jeśli żaden z bloków `case` nie pasuje, można użyć opcjonalnego bloku `default`. W przypadku braku bloku `default` i nieznalezienia pasującego bloku `case` instrukcja `switch` zakończy się bez wykonywania instrukcji.

Listing 1.6. Kod przykładowej instrukcji `switch`

```
int Liczba = 20;

switch (Liczba)
{
    case 10:
        Console.WriteLine("Liczba = 10");
        break;

    case 20:
        Console.WriteLine("Po wykonaniu przejdzie do case 30");
        goto case 30;

    case 30:
        Console.WriteLine("Było 20, jest 30");
        break;

    default:
        Console.WriteLine("Wartość domyślna");
        break;
}
```

Zasady zastosowania instrukcji `switch`:

- ◆ działa na typach: `int`, `uint`, `sbyte`, `byte`, `short`, `ushort`, `long`, `ulong`, `char`, `string` i `enum` (typy takie jak `double` i `float` nie są obsługiwane);
- ◆ nie mogą występować dwa takie same przypadki `case`;
- ◆ wartość zadeklarowana w `case` musi być stała;
- ◆ nie można określić tego samego bloku kodu dla różnych wartości, jak to jest w przypadku instrukcji `if`.

Konstrukcje `switch` można tworzyć za pomocą snippetu `switch`. Należy wpisać słowo `switch`, a następnie kliknąć dwukrotnie klawisz `Tab`.

Instrukcje iteracyjne

W przypadku gdy zachodzi potrzeba ponownego wykonania tej samej instrukcji lub grupy instrukcji, należy skorzystać z pętli, czyli instrukcji iteracyjnych. Pętla to nic innego jak wielokrotne wykonywanie tych samych instrukcji¹⁶. Wyróżnia się następujące typy instrukcji iteracyjnych:

- ◆ while,
- ◆ do while,
- ◆ for,
- ◆ foreach.

Pętla while

Instrukcja `while` wykonuje kod aż do momentu spełnienia warunku zakończenia. Ważne jest, aby warunek zakończenia został osiągnięty; w przeciwnym wypadku pętla będzie nieskończona, co spowoduje maksymalne obciążenie komputera i zablokuje program. Warto tutaj zwrócić uwagę na przypadek, gdy warunek zakończenia jest np. parzystą liczbą, a iteracja odbywa się tylko po nieparzystych liczbach. Aby zabezpieczyć się przed sytuacją, w której nie zostanie osiągnięty warunek zakończenia, należy zastosować operator `<=` zamiast `==`. Warunek zakończenia, czyli wyrażenie sterujące w pętli `while`, jest wymagany. Składnia instrukcji `while` jest następująca:

```
while (wyrażenie)
    instrukcja1
```

gdzie:

- ◆ wyrażenie — wyrażenie kończące działanie pętli, wartość `bool` (wymagane);
- ◆ instrukcja1 — instrukcja lub grupa instrukcji wykonywanych w pętli.

Poniżej zaprezentowano przykładowy kod instrukcji `while`:

```
while (warunek_zakończenia)
{
    // Kod do wykonania, inkrementacja
}
```

Pętlę `while` można tworzyć za pomocą snippetu `while`. Wpisz słowo `while`, a następnie kliknij dwukrotnie klawisz `Tab`.

Pętla do while

Pętla `do while` działa identycznie jak pętla `while`, z tą różnicą, że bez względu na warunek kod instrukcji zostanie wykonany przynajmniej jeden raz. Należy tutaj zauważyć, że inkrementacja zmiennej następuje jeszcze przed pierwszym sprawdzeniem warunku.

¹⁶ <http://msdn.microsoft.com/pl-pl/library/32dbf1by.aspx>

Składnia instrukcji do `while` jest następująca:

```
do
    instrukcja1
    while (wyrażenie)
```

gdzie:

- ◆ wyrażenie — wyrażenie kończące działanie pętli, wartość `bool` (wymagane);
- ◆ instrukcja1 — instrukcja lub grupa instrukcji wykonywanych w pętli.

Poniżej zaprezentowano przykładowy kod instrukcji `do while`:

```
do
{
    // Kod do wykonania, inkrementacja
}
while (warunek zakończenia);
```

Pętlę `do while` można tworzyć za pomocą snippetu `do`. Wpisz słowo `do`, a następnie kliknij dwukrotnie klawisz `Tab`.

Pętla `for`

W pętli `for` decyduje się, ile iteracji ma zostać wykonanych. Wyrażenia sterujące nie są wymagane. W przypadku pętli `for` bez instrukcji sterujących pętla taka zachowuje się jak nieskończona pętla `while`. Aby z niej wyjść, należy użyć instrukcji `break`.

W pętli `for` można zadeklarować kilka zmiennych iteracyjnych widzianych tylko w obrębie pętli, należy jednak pamiętać, że muszą to być zmienne tego samego typu. Zmienne są iterowane pierwszy raz po pierwszym wykonaniu instrukcji. Istnieje również możliwość pominięcia iteracji w nagłówku pętli i umieszczenia go razem z instrukcjami w ciele pętli, co nie przeszkodzi w poprawnym działaniu pętli. Składnia instrukcji `for` jest następująca:

```
for ([inicjatory]:[wyrażenie]:[iteratory])
    instrukcja1
```

gdzie:

- ◆ wyrażenie — wyrażenie kończące działanie pętli, wartość `bool` (opcjonalne), w przypadku braku wyrażenia kończącego domyślnie przyjmowana jest wartość `true`;
- ◆ instrukcja1 — instrukcja lub grupa instrukcji wykonywanych w pętli;
- ◆ inicjatory — lista instrukcji lub wyrażeń inicjujących oddzielonych przecinkami (opcjonalne);
- ◆ iteratory — lista wyrażeń operujących na licznikach (opcjonalne).

Poniżej zaprezentowano przykładowy kod rozbudowanej instrukcji for:

```
for (int i = 1, j = 20; i <= 10 || j<=40 : i++, j+=2)
{
    Console.WriteLine(„Wypisz i: {0} oraz j: {1}”, i, j);
}
```

Pętlę for można tworzyć za pomocą snippetu for. Wpisz słowo for, a następnie kliknij dwukrotnie klawisz Tab.

Pętla foreach

Pętla foreach służy do operowania na elementach tablicy, łańcuchach lub kolekcji. Najważniejszą różnicą w porównaniu do pętli for jest to, że nie można zmieniać liczbę elementów w kolekcji, po której się iteruje, a więc nie da się dodawać i usuwać elementów ani zmieniać wartości zmiennych, po których się iteruje. Pętla ta ma jedną wielką zaletę — zapobiega przekroczeniu indeksu tabeli lub bufora. Pętla foreach jest bardzo przydatna w pracy z kolekcjami zwracanymi z zapytań LINQ. Składnia instrukcji foreach jest następująca:

```
foreach (typ identyfikator in wyrażenie)
    instrukcja1
```

gdzie:

- ◆ wyrażenie — kolekcja lub tablica wartości pozwalających na konwertowanie na typ danych identyfikatora (wymagane);
- ◆ instrukcja1 — instrukcja lub grupa instrukcji wykonywanych w pętli;
- ◆ typ — typ danych elementu zbioru (wymagane);
- ◆ identyfikator — nazwa zmiennej dla pojedynczego elementu zbioru (wymagane).

Poniżej zaprezentowano przykładowy kod instrukcji foreach:

```
string strTekst = "Jakiś tekst";
foreach (char znak in strTekst)
{
    // Wypisz znak
}
```

Pętlę foreach można tworzyć za pomocą snippetu foreach. Wpisz słowo foreach, a następnie kliknij dwukrotnie klawisz Tab.

Instrukcje skoku

Do podstawowych instrukcji skoku¹⁷ należą:

- ◆ instrukcja break — wychodzi z pętli;
- ◆ instrukcja continue — pozwala kontynuować pętlę np. bez wykonania wszystkich instrukcji w niej zawartych; przechodzi do następnego obiegu pętli;

¹⁷ <http://msdn.microsoft.com/pl-pl/library/d96yfwee.aspx>

- ◆ instrukcja goto — przenosi wykonywanie programu do określonego miejsca w kodzie oznaczonego etykietą; instrukcja goto jest bardzo rzadko używana, ponieważ bardzo komplikuje analizę kodu;
- ◆ instrukcja return — oprócz zakończenia pętli kończy również działanie metody;
- ◆ instrukcja throw — rzuca wyjątek zatrzymującą pętlę.

Klasy, obiekty, pola, metody i właściwości

Programowanie obiektowe to koncepcja programowania bazująca na pojęciach klasy i obiektu. Programy definiuje się za pomocą obiektów, czyli elementów łączących stan (dane) i zachowanie (metody). Program składa się z wielu obiektów komunikujących się między sobą w celu wykonania zadania. Obiekty nie są ze sobą bezpośrednio powiązane.

Klasy

Klasy są podstawowymi typami referencyjnymi w C#. Klasa w programowaniu to rodzaj klasyfikacji, czyli próby sformułowania częściowej lub całkowitej definicji dla pewnych obiektów. Definicja klasy obejmuje dopuszczalny stan obiektów oraz ich zachowania. Ogólna składnia deklaracji klasy wygląda następująco:

```
[modyfikatory] class Identyfikator [ :lista_bazowa]
{
    składniki_klasy
}
```

gdzie:

- ◆ modyfikatory — dostępne modyfikatory to: new, abstract, sealed, public, private, protected, internal (opcjonalne) — domyślnie elementy danej klasy są prywatne;
- ◆ identyfikator — nazwa klasy (wymagane);
- ◆ lista_bazowa — lista może zawierać nazwę tylko jednej klasy, po której dziedziczy, oraz nazwy interfejsów oddzielone przecinkami (opcjonalne);
- ◆ składniki_klasy — lista pól i metod danej klasy.

Poniżej zaprezentowano przykładowy kod klasy:

```
public class HomeController : Controller
{
    public ActionResult Index()
    {
        ViewBag.Message = "Wiadomość";
        return View();
    }
}
```

Deklarację klasy można tworzyć za pomocą snippetu `class`. Wpisz słowo `class`, a następnie kliknij dwukrotnie klawisz *Tab*¹⁸.

Dostępne są modyfikatory dostępu dla klas¹⁹:

- ◆ `public` — publiczne składniki klasy są dostępne dla wszystkich metod wszystkich klas;
- ◆ `private` — składowe prywatne są dostępne tylko dla metod klasy, w której się znajdują;
- ◆ `protected` — dostępne są dla klasy, w której się znajdują, oraz dla klas dziedziczących po niej w danej bibliotece;
- ◆ `internal` — składowe wewnętrzne są dostępne dla klasy znajdującej się w danym podzespołe (biblioteka, plik `*.dll`);
- ◆ `protected internal` — jest to połączenie modyfikatorów `protected` i `internal`, jednak klasa nie może być jednocześnie `protected` i `internal`, tylko `protected` lub `internal` — w zależności od sytuacji, w jakiej jest użyta; dla klasy dziedziczącej będzie to modyfikator `protected` (klasa ta nie musi być z tego samego podzespołu), natomiast w przypadku innej niedziedziczącej klasy, która jednak znajduje się w tym samym podzespołe, będzie to modyfikator `internal`.

Pozostałe modyfikatory to:

- ◆ `abstract` — klasa abstrakcyjna może być tylko klasą bazową dla innych klas; nie można utworzyć instancji takiej klasy;
- ◆ `async` — wskazuje, że metoda, wyrażenie lambda lub metoda anonimowa są asynchroniczne;
- ◆ `const` — oznacza, że wartość składowej lub zmiennej nie może być modyfikowana;
- ◆ `event` — deklaruje zdarzenie;
- ◆ `extern` — wskazuje, że metoda została zaimplementowana zewnętrznie;
- ◆ `new` — ukrywa składniki klasy, które mają taką samą nazwę w klasie podstawowej; operator niweluje ostrzeżenie o ukrywaniu składnika klasy;
- ◆ `override` — służy do nadpisywania metod wirtualnych odziedziczonych po klasie bazowej;
- ◆ `partial` — pozwala rozbić klasy, metody i struktury na wiele kawałków kodu przechowywanych w osobnych plikach;
- ◆ `readonly` — określa, że polu można nadać wartość jedynie w konstruktorze lub w deklaracji pola;

¹⁸ <http://msdn.microsoft.com/pl-pl/library/x9afc042.aspx>

¹⁹ <http://msdn.microsoft.com/pl-pl/library/ms173121.aspx>

- ◆ `sealed` — określa, że klasa jest klasą ostateczną i nie może być klasą bazową dla innych klas;
- ◆ `static` — określa, że składnik nie należy do obiektu, tylko do typu;
- ◆ `unsafe` — deklaruje niebezpieczny kontekst, czyli daje dostęp do operatorów: `sizeof`, `&`, `->` oraz `*`;
- ◆ `virtual` — określa metody wirtualne, które mogą być nadpisywane w klasach dziedziczących po klasie, w której się znajdują;
- ◆ `volatile` — określa pole, które może być modyfikowane w programie przez system operacyjny, sprzęt lub równoległy działający wątek.

Obiekty

Obiekt jest instancją klasy, czyli tworem o określonym zachowaniu i cechach zdefiniowanych w klasie. Nie można działać na klasie, tylko na obiekcie, czyli instancji klasy²⁰. Tworzenie obiektu przebiega następująco:

```
Pojazd samochód = new Pojazd();
```

Każdy obiekt charakteryzuje:

- ◆ niepowtarzalność — każdy jest osobnym tworem, można jednak skopiować obiekt, a dokładniej jego zawartość;
- ◆ zachowanie — każdy obiekt może wykonywać określone operacje;
- ◆ stan, w jakim się znajduje.

Pola

Pola to zmienne dowolnego typu dostępne w obrębie klasy. W zależności od modyfikatora dostępu dostępne są one również dla innych klas²¹. Domyślnie pola są prywatne. Ogólna składnia deklaracji pola klasy wygląda następująco:

```
[modyfikator] typ Identyfikator [=wart_początkowa];
```

gdzie:

- ◆ modyfikator — dozwolone są modyfikatory: `static`, `readonly`, `new`, `volatile`, `private`, `public`, `internal`, `protected` (opcjonalne);
- ◆ identyfikator — nazwa pola (wymagane);
- ◆ typ — typ danych pola (wymagane);
- ◆ wart_początkowa — wartość początkowa pola (opcjonalne).

²⁰ <http://msdn.microsoft.com/pl-pl/library/ms173110.aspx>

²¹ <http://msdn.microsoft.com/pl-pl/library/ms173118.aspx>

Poniżej zaprezentowano przykładowy kod:

```
public class HomeController : Controller
{
    int Liczba = 1;
}
```

Metody

Metody to funkcje, czyli bloki kodu zawierające serie instrukcji. Metoda jest składnią klasy implementującym obliczenia albo działania wykonywane przez obiekt²².

Metoda musi zwracać wartość zgodną z typem umieszczonym w definicji. Aby zwrócić wartość i zakończyć działanie metody, używa się słowa kluczowego return. Dalsza część kodu znajdująca się za słowem return nie zostaje wykonana. W przypadku metod niezwracających żadnych wartości słowo return nie jest wymagane. Należy pamiętać, aby dla każdego dostępnego w metodzie scenariusza była umieszczona instrukcja return. W metodzie może się znajdować kilka lub nawet kilkanaście instrukcji return, jednak wykonana zostanie tylko jedna. Ogólna składnia definicji metody wygląda następująco:

```
[modyfikator] typ Identyfikator [typ_parametru
↳ identyfikator_parametru];
```

gdzie:

- ◆ modyfikator — domyślnie private (opcjonalne);
- ◆ identyfikator — nazwa metody (wymagane);
- ◆ typ — typ zwracanego wyniku (wymagane);
- ◆ typ_parametru — typ parametru przekazywanego do metody;
- ◆ identyfikator_parametru — nazwa parametru.

Poniżej pokazano przykładowy kod metody publicznej:

```
public int LiczbaDoKwadratu(int i)
{
    // Zwraca kwadrat liczby
    return i * i;
}
```

Argumenty metod

Argumenty metod to informacje przekazywane do lub z metody²³. Sposoby przekazywania argumentów:

- ◆ Przez wartość — domyślna metoda przekazywania argumentów, do metody przekazywana jest wartość zmiennej (listing 1.7).

²² <http://msdn.microsoft.com/pl-pl/library/ms173114.aspx>

²³ <http://msdn.microsoft.com/pl-pl/library/0f66670z.aspx>

Listing 1.7. Przekazywanie argumentów przez wartość

```
class Przyklad_Wartosc
{
    static void Metoda(int i)
    {
        i = i + 44;
        // Tutaj w metodzie liczba = 45
    }

    static void Main()
    {
        int liczba = 1;
        Metoda(liczba);
        Console.WriteLine(liczba);
        // Tutaj liczba = 1
    }
}
```

- ◆ Przez referencje (*ref*) — w przypadku przekazywania argumentu przez wartość operuje się na kopii przekazanych danych, natomiast w przypadku przekazywania przez referencje pracuje się na oryginalnych danych (listing 1.8).

Listing 1.8. Przekazywanie argumentów przez referencje

```
class Przyklad_Ref
{
    static void Metoda(ref int i)
    {
        i = i + 44;
    }

    static void Main()
    {
        int liczba = 1;
        // Tutaj liczba = 1
        Metoda(ref liczba);
        Console.WriteLine(liczba);
        // Tutaj liczba = 45
    }
}
```

- ◆ Przez wyjście (*out*) — służy do przekazywania danych z metody, a nie do metody; operuje na kopii danych, które muszą zostać zainicjalizowane wartością, która zostanie przypisana danym oryginalnym (listing 1.9).

Listing 1.9. Przekazywanie argumentów przez wyjście

```
class Przyklad_Out
{
    static void Metoda(out int i)
    {
        i = 44;
    }
}
```

```
static void Main()
{
    int liczba = 1;
    // Tutaj liczba = 1
    Metoda(out liczba);
    // Tutaj liczba = 44
}
```

Możliwe jest przekazanie zmiennej liczby argumentów do metody np. w postaci tablicy. Służy do tego słowo `params`.

Poniżej zaprezentowano przykładowy kod metody ze zmienną liczbą parametrów:

```
public int Dodawaj(params int[] Tablica)
{
    // Ciało metody
}
```

Wywoływanie metod

Aby wywołać metodę, podaje się jej nazwę i przekazuje listę argumentów. Gdy metoda zwraca wartość, można wykonać wywołanie razem z przypisaniem. Przykładowe wywołanie metody razem z przypisaniem wygląda następująco:

```
int liczba = ZwrocLiczbe(int a);
```

W przypadku gdy chcesz wywołać metodę na rzecz obiektu (listing 1.10), użyj operatora dostępu do składowej: „.” (kropka).

Listing 1.10. Wywołanie metody na rzecz obiektu

```
class Liczba
{
    public void WypiszLiczbe()
    {
    }
}

Liczba liczba = new Liczba();
liczba.WypiszLiczbe();
```

W przypadku wywoływania metod w klasie, w której zostały zdefiniowane, nie stosuje się operatora dostępu. Metoda nie zostaje wywołana na rzecz obiektu, zatem nie musi zawierać operatora dostępu. Podobnie metody statyczne nie wymagają operatora dostępu, ponieważ nie należą do obiektów (instancji klasy), tylko do klasy.

Rekurencyjne wywołanie metod polega na wywoływaniu metody z wnętrza tej samej lub innej metody. Metoda rekurencyjna wywołuje samą siebie, dlatego wymagane jest zapewnienie wyjścia z metody (instrukcja `return`), gdyż w przeciwnym wypadku metody będą się wywoływać do momentu przepelenienia stosu. Rekurencja jest zasobnierzym procesem, dlatego powinno się stosować postać iteracyjną, która powoduje

duże mniejsze obciążenie. W większości przypadków możliwa jest zamiana postaci rekurencyjnej na iteracyjną. Podczas komplikacji, jeśli jest taka możliwość, kompilator automatycznie zamienia postać rekurencyjną na iteracyjną.

Przykładowa metoda rekurencyjna wygląda następująco:

```
public void WypiszLiczbe()
{
    WypiszLiczbe();
}
```

Właściwości

Właściwości wspierają hermetyzację danych wewnętrz klasy. Są to pola, które pozwalają określić rodzaj dostępu jako: tylko do odczytu, tylko do zapisu lub do odczytu i zapisu. Właściwości łączą zachowanie pól i metod. Wyglądają jak pola, a więc można wyświetlić ich zawartość. Zachowują się jak metody, co daje możliwość zastosowania w interfejsach²⁴.

Składnia deklaracji właściwości jest następująca:

```
[modyfikatory] typ Identyfikator
{
    deklaracje_dostępu
}
```

gdzie:

- ◆ modyfikatory — dostępne są modyfikatory: new, static, virtual, abstract, override oraz wszystkie modyfikatory dostępu (opcjonalne);
- ◆ typ — typ danych (wymagane);
- ◆ Identyfikator — nazwa właściwości;
- ◆ deklaracje_dostępu — blok get i/lub set, wymagany jest przynajmniej jeden z bloków get lub set.

Poniżej pokazano przykładowy kod w pełnej wersji:

```
public class Entity
{
    public int Id
    {
        get
        {
            return Id;
        }
        set
        {
            Id = value;
        }
    }
}
```

²⁴ <http://msdn.microsoft.com/pl-pl/library/x9fsa0sw.aspx>

Można go jednak zapisać też w skróconej postaci:

```
public class Entity  
{  
    public int Id { get; set; }  
}
```

Obie wersje działają tak samo.

Właściwości:

- ♦ nie przyjmują argumentów;
- ♦ nie mogą być typu `void` — muszą zwracać wartość;
- ♦ nazwy właściwości zaczyna się dużą literą, natomiast pola prywatne mają;
- ♦ mogą zostać przekazane do metod tylko poprzez wartość; niedozwolone jest przekazywanie właściwości poprzez referencje (`ref`) oraz wyjście (`out`);
- ♦ odwołanie do właściwości wygląda tak samo jak odwołanie do pól;
- ♦ mogą być deklarowane w interfejsach;
- ♦ od C# 2.0 dostępne są prywatne settery.

Podstawowe pojęcia związane z programowaniem obiektowym

Do podstawowych pojęć bezpośrednio związanych z programowaniem obiektowym należą:

- ♦ abstrakcja,
- ♦ hermetyzacja,
- ♦ dziedziczenie,
- ♦ polimorfizm.

Abstrakcja

Abstrakcja polega na ukrywaniu lub pomijaniu mało istotnych informacji, a skupieniu się na wydobyciu informacji, które są niezmienne i wspólne dla pewnej grupy obiektów. Pojęciem abstrakcyjnym jest np. środek transportu. Środkiem transportu może być zarówno samolot, jak i samochód. Oba te środki transportu poruszają się z pewną prędkością, a więc prędkość jest tu wspólnym parametrem. Idąc dalej, samochód może być osobowy lub ciężarowy. Każdy z nich posiada cechy wspólne, takie jak np. spalanie czy waga. Każdy z tych pojęć jest abstrakcyjne, ponieważ posiada pewne cechy wspólne dla grupy oraz cechy ukryte, czyli te, którymi się różnią. Kontynuując, samochody ciężarowe można przykładowo podzielić ze względu na markę lub liczbę osi. Idąc tym tokiem rozumowania, należy się zastanowić, gdzie kończy się abstrakcja. Prawdopodobnie nigdzie, gdyż zawsze znajdują się cechy wspólne i cechy, którymi dane elementy się różnią. To z kolei pozwala definiować kolejne poziomy abstrakcji.

Klasa abstrakcyjna to taka klasa, dla której nie można utworzyć obiektu. Jest bardzo podobna do interfejsu, z tą różnicą, że klasy mogą dziedziczyć po wielu interfejsach, a tylko po jednej klasie. Klasa abstrakcyjna w odróżnieniu od interfejsu może zawierać ciało metody oraz pola bądź zmienne. Wyjątkiem są metody abstrakcyjne, które nie mogą zawierać swojego ciała²⁵. Najpopularniejszym przykładem obrazującym zastosowanie klasy abstrakcyjnej jest obliczanie pola powierzchni figur geometrycznych, np. wielokątów. Przykładowo tworzy się klasę abstrakcyjną `wielokokat`, a w niej metodę abstrakcyjną `ObliczPole()` i dwa pola dla długości boków. Dla każdego rodzaju figury tworzy się klasę dziedziczącą po klasie abstrakcyjnej, dodaje pola, których brakuje w klasie bazowej, czyli np. gdy wielokąt ma pięć boków, trzeba dodać kolejne trzy pola i zaimplementować metodę `ObliczPole()` dla tego konkretnego przypadku, a więc dla pięciokąta. Podobnie postępuje się dla innych wielokątów i w każdym przypadku implementuje się inne ciało metody `ObliczPole()`.

Hermetyzacja

Hermetyzacja (enkapsulacja) polega na ukrywaniu nieistotnych informacji na temat obiektu w celu zminimalizowania efektów jego modyfikacji. Polega na oddzieleniu tego, co zawiera i co może zrobić obiekt, od tego, jak jest zbudowany i jak to robi. Przykładowo jeden obiekt potrafi obliczyć pewne zadanie, a inny potrzebuje tylko wyniku. Dla tego drugiego obiektu nieważne jest, w jaki sposób pierwszy obiekt to oblicza — ważny jest tylko wynik.

Dziedziczenie

Dziedziczenie pozwala rozszerzać możliwości klas poprzez implementacje osobnych klas rozszerzających²⁶. Klasa, po której dziedziczą inne klasy, jest klasą bazową. Klasy, które dziedziczą po klasie bazowej, noszą nazwę klas pochodnych. Dzięki dziedziczeniu można tworzyć nowe klasy w oparciu o już istniejące, bez potrzeby implementowania tych funkcjonalności, które zostały już zaimplementowane w klasach bazowych. Klasy potomne nie mogą być bardziej dostępne od klasy bazowej. Dziedziczyć można tylko z jednej klasy bazowej oraz z wielu interfejsów. Nazwę klasy i interfejsów, po których się dziedziczy, zapisuje się po znaku dwukropka w miejscu definicji nowej klasy:

```
public class KlasaDziedzicząca : KlasaBazowa, IBazoweInterfejsy
{
}
```

Dostęp do składowych klas bazowych:

- ◆ składowe `public` (publiczne) są także publiczne dla klasy potomnej;
- ◆ składowe `protected` (chronione) są dostępne dla klasy potomnej;
- ◆ składowe `private` (prywatne) nie są bezpośrednio dostępne dla klasy potomnej, można jednak zmieniać ich wartość za pomocą publicznych metod lub właściwości.

²⁵ <http://msdn.microsoft.com/pl-pl/library/ms173150.aspx>

²⁶ <http://msdn.microsoft.com/pl-pl/library/ms173149.aspx>

Polimorfizm

Przesłanianie metod (polimorfizm) pozwala na zamianę definicji metody z klasy bazowej na tę znajdującą się w klasie pochodnej. Dzieje się to w czasie wywołania metody dla klasy pochodnej.

Metody wirtualne:

- ◆ metody przesłaniane (wirtualne) poprzedza się modyfikatorem `virtual`;
- ◆ metody wirtualne muszą zawierać ciało (może być to ciało puste, czyli metoda nic nie robi, ale zawiera nawiasy klamrowe — pusty blok instrukcji);
- ◆ nie mogą być statyczne;
- ◆ nie mogą być prywatne.

Metody przesłaniające:

- ◆ poprzedza się modyfikatorem `override`;
- ◆ stają się automatycznie metodami wirtualnymi, przez co mogą być przesłonięte w klasie potomnej;
- ◆ muszą zawierać ciało (może być to ciało puste, czyli metoda nic nie robi, ale zawiera nawiasy klamrowe — pusty blok instrukcji);
- ◆ muszą mieć składnię identyczną jak składnia metody wirtualnej z klasy bazowej (nie można np. dodawać dodatkowych parametrów);
- ◆ nie mogą być statyczne;
- ◆ nie mogą być prywatne.

Istnieje możliwość ukrywania metod. Ukrycie metody polega na zastąpieniu metody o tej samej nazwie z klasą bazową zupełnie inną metodą w klasie pochodnej. Aby ukryć metodę, stosuje się słowo kluczowe `new`. Metoda ukrywająca może zwracać inny typ wartości niż metoda ukryta, ponieważ ten parametr nie jest brany pod uwagę w szacowaniu identyczności metody. Ukrywanie metod można stosować zarówno do metod wirtualnych, jak i niewirtualnych. W przypadku stosowania mechanizmu ukrywania dla metod wirtualnych stają się one metodami numer jeden w łańcuchu dziedziczenia.

Domyślnym konstruktorem uruchamianym w czasie wywołania konstruktora dla klasy potomnej jest bezparametryowy konstruktor klasy bazowej. Aby zamiast bezparametryowego konstruktora wywołać inny konstruktor z parametrami, używa się słowa kluczowego `base`. Dopiero po wywołaniu konstruktora klasy bazowej następuje wywołanie konstruktora z klasy potomnej. Słowa kluczowego `base` używa się również do odwoływania się do składowych klasy bazowej (w przypadku gdy pokrywają się nazwy pól). Poniżej zaprezentowano przykładowy kod korzystający ze słowa kluczowego `base`:

```
public class FullContext : DbContext
{
    public FullContext()
        : base("DefaultConnection")
    {
    }
```

```

        }
    public DbSet<Kurs> Kursy { get; set; }
}

```

Przeciążanie operatorów

Język C# pozwala zdefiniować funkcjonalność operatorów dla własnych typów, czyli np. tworzonych obiektów. Dzięki takim zabiegom zamiast pisać i wywoływać za każdym razem metodę odejmowania, można nadpisać operator, czyli utworzyć metodę, która coś zwraca, przyjmuje parametry oraz posiada w deklaracji słowo kluczowe operator wraz z symbolem operatora, który nadpisuje się, a więc w tym wypadku będzie to operator-. Każdy operator ma określzoną liczbę argumentów, dla operatora - muszą to być dwa argumenty²⁷.

Składnie definicji przeciążonego operatora można podzielić na typy:

- ◆ public static zwracany_typ operator op_unarny(typ_arg argument),
- ◆ public static zwracany_typ operator op_binarny(typ_arg1 argument1, typ_arg2 argument2),
- ◆ public static implicit operator typ_wyj(typ_wej argument),
- ◆ public static explicit operator typ_wyj(typ_wyj argument),

gdzie:

- ◆ zwracany_typ — typ zwracany przez operator (wymagane);
- ◆ op_unarny — operator unarny dostępny do przeciążania: +, -, !, ~, ++, --, true, false (wymagane);
- ◆ op_binarny — operator binarny dostępny do przeciążania: +, -, *, /, %, <<, >>, <, >, <=, >=, ==, !=, &, |, ^ (wymagane);
- ◆ typ_arg — typy argumentów wejściowych (wymagane);
- ◆ argument — nazwa argumentu wejściowego (wymagane);
- ◆ typ_wyj — typ wyjściowy operatora konwersji (wymagane);
- ◆ typ_wej — typ wejściowy operatora konwersji (wymagane).

Operatory, których nie wolno przeciążać:

- ◆ logiczne warunkowe: &&, ||;
- ◆ konwersji: () — służą do tego słowa kluczowe explicit i implicit;
- ◆ indeksacji: [] — stosuje się w tym celu indeksatory;
- ◆ przypisań: +=, -=, *=, %=, <<=, >>, |=, ^=, &=, /=;
- ◆ pozostałe: =, ., new, is, sizeof, typeof, ?::.

²⁷ <http://msdn.microsoft.com/pl-pl/library/8edha89s.aspx>

Przeciążanie operatorów relacji

Operatory relacji, czyli operatory takie jak `==`, `!=`, `<`, `>`, `<=`, `>=`, muszą być przeciążane parami. Nie można przeciążyć tylko jednego operatora z pary.

Metody Equals() i GetHashCode()

Przeciążając operatory `==` i `!=`, należy również nadpisać metody `Equals()` i `GetHashCode()`, które są dziedziczone po klasie `System.Object`. Metoda `Equals()` działa identycznie jak operator `==`, natomiast metoda `GetHashCode()` to wartość liczbową służąca do identyfikacji obiektu podczas testowania równości. Jeśli wynik metody `Equals()` daje wartość `true`, to metoda `GetHashCode()` musi zwracać te same wartości dla obydwu obiektów. Jeśli natomiast metoda `Equals()` zwraca wartość `false`, to metoda `GetHashCode()` może zwracać takie same lub różne wartości dla porównywanych obiektów.

Metoda `GetHashCode()` musi zwracać tę samą wartość tak długo, jak długo zmiany obiektu nie wpływają na zmianę wyniku metody `Equals()`.

Przeciążanie operatorów konwersji

Można przeciążać operator konwersji w postaci jawniej (ang. *Explicit*) oraz w postaci niejawniej (ang. *Implicit*). Aby przeciążyć operator niejawnego, używa się słowa kluczowego `implicit` (listing 1.11), natomiast aby przeciążyć operator jawnego, używa się słowa kluczowego `explicit` (listing 1.12) w definicji operatora. Aby uruchomić przykładowe programy w Visual Studio, wybierz *File/New Project/Windows Desktop/Console Application* i wklej kod zastępując domyślną metodę `Main()`.

Listing 1.11. Przykładowy kod — konwersja niejawnna

```
class Obwod
{
    private int _bok1;
    private int _bok2;
    private double _bok3;

    public Obwod(int bok1, int bok2, double bok3)
    {
        _bok1 = bok1;
        _bok2 = bok2;
        _bok3 = bok3;
    }

    public static implicit operator double(Obwod f)
    {
        return f._bok1 + f._bok2 + f._bok3;
    }
}

class Wyświetl
{
    public static void Main()
    {
        Obwod fig1 = new Obwod(3, 4, 4.34);

        double obwod = fig1;
```

```

        Console.WriteLine("Obwód wynosi: " + obwod);
        Console.ReadLine();
    }
}

```

Listing 1.12. Przykładowy kod — konwersja jawną

```

class Obwod
{
    private int _bok1;
    private int _bok2;
    private double _bok3;

    public Obwod(int bok1, int bok2, double bok3)
    {
        _bok1 = bok1;
        _bok2 = bok2;
        _bok3 = bok3;
    }

    public static explicit operator double(Obwod f)
    {
        return f._bok1 + f._bok2 + f._bok3;
    }
}

class Wyświetl
{
    public static void Main()
    {
        Obwod fig1 = new Obwod(3, 4, 4.34);

        double obwod = (double)fig1;

        Console.WriteLine("Obwód wynosi: " + obwod);
        Console.ReadLine();
    }
}

```

W przykładach zamieniamy typ obiektu obwod na wartość double. Jak łatwo zauważyć, jedyna różnica pomiędzy tymi dwoma rozwiązaniami sprowadza się do tego, że przy korzystaniu z konwersji jawną trzeba wykonać rzutowanie na typ double.

Przeciążanie operatorów logicznych

Operatory logiczne są rozwijane za pomocą &, |, true i false, dlatego nie ma możliwości rozwinęcia operatorów logicznych wprost. Aby rozwinąć wyrażenie z operatorem && lub ||, trzeba przeciążyć operatory &, |, true oraz false.

Rozwinięcie wygląda następująco:

- ◆ dla operatora && jest to T.false(x) ? x : T.&(x,y),
- ◆ dla operatora || jest to T.true(x) ? x : T.|(x,y),

gdzie T to typ zmiennych x i y.

Przykładowy kod przedstawiono na listingu 1.13.

Listing 1.13. Przykład przeciążania operatorów logicznych

```
class Obwod
{
    private int _bok1;
    private int _bok2;

    public Obwod(int bok1, int bok2)
    {
        _bok1 = bok1;
        _bok2 = bok2;
    }

    public static Obwod operator &(Obwod ob1, Obwod ob2)
    {
        return new Obwod(ob1._bok1 & ob2._bok1, ob1._bok2 &
                           ↪ob2._bok2);
    }

    public static Obwod operator |(Obwod ob1, Obwod ob2)
    {
        return new Obwod(ob1._bok1 | ob2._bok1, ob1._bok2 | 
                           ↪ob2._bok2);
    }

    public static bool operator true(Obwod ob1)
    {
        return ob1._bok1 != 0 && ob1._bok2 != 0;
    }

    public static bool operator false(Obwod ob1)
    {
        return ob1._bok1 == 0 || ob1._bok2 == 0;
    }
}

class Wyświetl
{
    public static void Main()
    {
        Obwod fig1 = new Obwod(3, 4);
        Obwod fig2 = new Obwod(0, 5);
        Obwod fig3 = new Obwod(5, 6);

        if (fig1 || fig2)
            Console.WriteLine("Długości boków w fig1 lub fig2
                           ↪różne od zera");

        if (fig1 && fig3)
            Console.WriteLine("Długości boków w fig1 i fig3
                           ↪różne od zera");

        Console.ReadLine();
    }
}
```

Dzięki przeciążeniu operatorów można użyć operatora `&&` lub `||` dla obiektu typu `obwod`. Bez przeciążenia nie byłoby możliwości zastosowania tych operatorów do działań na obiektach z klasy `Obwod`. W powyższym przykładzie zostaną wyświetlane obydwie linijki, ponieważ tylko `fig2` posiada bok o długości 0. Aby przetestować działanie programu, należy zmieniać długości boków dla `fig1`, `fig2` i `fig3`.

Przeciążanie operatorów arytmetycznych

W języku C# można przeciążać następujące operatory arytmetyczne: `+`, `-`, `/`, `*`. Jedy- nym ograniczeniem, jakie występuje w przypadku przeciążania operatorów arytmetycznych, jest to, że przynajmniej jeden argument definiowanego operatora musi być obiektem klasy, dla której został przeciążony.

W przykładzie z listingu 1.14 przeciążono operatory `+` i `-`. W przypadku operatora dodawania dodaje się do siebie długości boków, natomiast w przypadku operatora odejmowania odejmuje się od każdego boku określona liczbę — w tym przypadku jest to 2.

Listing 1.14. Przykład przeciążania operatorów arytmetycznych

```
class Obwod
{
    private int _bok1;
    private int _bok2;
    private double _bok3;

    public Obwod(int bok1, int bok2, double bok3)
    {
        _bok1 = bok1;
        _bok2 = bok2;
        _bok3 = bok3;
    }

    public static Obwod operator +(Obwod ob1, Obwod ob2)
    {
        return new Obwod(ob1._bok1 + ob2._bok1, ob1._bok2 +
            ↪ob2._bok2, ob1._bok3 + ob2._bok3);
    }

    public static Obwod operator -(Obwod ob1, int liczba)
    {
        return new Obwod(ob1._bok1 - liczba, ob1._bok2 -
            ↪liczba, ob1._bok3 - liczba);
    }

    public double ZwrocObwod()
    {
        return _bok1 + _bok2 + _bok3;
    }
}

class Wyświetl
{
    public static void Main()
    {
        Obwod fig1 = new Obwod(3, 4, 4.34);
```

```
Obwod fig2 = new Obwod(4, 5, 4.34);
Obwod fig3 = new Obwod(5, 6, 4.34);

Obwod obwod = fig1 + fig2 - 2;

Console.WriteLine("Obwód wynosi: " + obwod.ZwrocObwod());
Console.ReadLine();
}
```

Przeciążanie metod

W jednej klasie może być zdefiniowanych kilka metod o tej samej nazwie, ale muszą się one różnić listą przyjmowanych parametrów. Zdefiniowanie kilku metod o takiej samej nazwie i różnych parametrach nazywa się przeciążaniem metod. Podczas przeciążania brane są pod uwagę tylko przyjmowane parametry. Zwracana wartość nie jest uwzględniana (listing 1.15).

Listing 1.15. Przykład przeciążania metody

```
class Liczba
{
    public int WypiszLiczbe()
    {
        return 1;
    }

    public int WypiszLiczbe(int a)
    {
        return a;
    }
}
```

Indeksatory

Indeksatory pozwalają się odwoływać do tablicy zadeklarowanej w klasie. Składnia jest bardzo podobna do właściwości. W klasie zawierającej indeksator można mieć zadeklarowaną tylko jedną tablicę, do której odwołuje się poprzez nazwę obiektu i indeks. Podobnie jak właściwości, indeksator musi zawierać co najmniej jedną deklarację doługet lub set²⁸. Składnia indeksatorów wygląda następująco:

```
[modyfikatory] typ this[argumenty_indeksów]
{
    deklaracje_dostępu
}
```

²⁸ <http://msdn.microsoft.com/pl-pl/library/6x16t2tx.aspx>

gdzie:

- ◆ modyfikatory — dostępne są modyfikatory: new, virtual, abstract, sealed, extern, override oraz wszystkie modyfikatory dostępu (opcjonalne);
- ◆ typ — typ danych zwracanych przez indeksator (wymagane);
- ◆ argumenty_indeksu — lista argumentów w formacie: *typ nazwa*;
- ◆ deklaracje_dostępu — blok get i/lub set, wymagany jest przynajmniej jeden z bloków get lub set.

Poniżej zaprezentowano przykładowy kod indeksatora:

```
private int[] _lista = new int[100];
public int this[int indeks]
{
    get
    {
        return this._lista[indeks - 1];
    }

    set
    {
        this._lista[indeks-1] = value;
    }
}
```

Pozostałe informacje:

- ◆ indeksatory korzystają z takiej samej notacji jak tablice, jednak mogą używać różnych typów jako indeksów (nie tylko liczb całkowitych, jak ma to miejsce przy tablicach);
- ◆ indeksatory można przeciągać, czyli można utworzyć kilka indeksatorów o różnych typach indeksów;
- ◆ indeksator musi mieć przynajmniej jeden argument indeksujący, który jest przekazywany przez wartość (może posiadać ich więcej);
- ◆ nie można tworzyć statycznych indeksatorów, ponieważ są one powiązane z obiektami, czyli instancjami klasy;
- ◆ indeksatory mogą być przekazywane jako argumenty do metod tylko przez wartość, nie mogą być przekazywane przez referencje (*ref*) i wyjście (*out*).

Dozwolone jest deklarowanie indeksatorów w interfejsach. Wymusza się tym samym implementację indeksatora w klasie dziedziczącej po tym interfejsie (listing 1.16).

Listing 1.16. Przykładowy kod zastosowania indeksatora

```
public interface IJakisInterfejs
{
    //Deklaracja
    int this[int index]
    {
```

```
        get;
        set;
    }
}

// Implementacja interfejsu (klasa)
class KlasaIndeksatora : IJakisInterfejs
{
    private int[] _lista = new int[100];
    public int this[int index]
    {
        get
        {
            return _lista[index];
        }
        set
        {
            _lista[index] = value;
        }
    }
}
```

Indeksatory w interfejsach:

- ♦ nie mogą posiadać modyfikatorów dostępu, ponieważ muszą być publiczne;
- ♦ nie mogą zawierać ciała metod get i set²⁹.

Klasa System.Object

Klasa System.Object to podstawowa klasa, która jest klasą bazową dla wszystkich innych klas. W przypadku gdy konkretna klasa nie dziedziczy po innej klasie, domyślnie zostaje przyjęte, że dziedziczy po System.Object³⁰.

Klasa System.Object zawiera następujące metody:

- ♦ Equals(Object) — określa, czy obiekty są równe;
- ♦ Finalize — pozwala obiektowi na zwolnienie zasobów;
- ♦ GetHashCode — dostarcza mechanizm haszowania;
- ♦ GetType — zwraca typ instancji klasy;
- ♦ MemberwiseClone — tworzy kopię obiektu;
- ♦ ReferenceEquals — określa, czy instancje klas odnoszą się do tego samego obiektu;
- ♦ ToString — konwersja obiektu na typ tekstowy.

²⁹ <http://msdn.microsoft.com/pl-pl/library/tkyhsw31.aspx>

³⁰ <http://msdn.microsoft.com/pl-pl/library/system.object.aspx>

Konstruktor i destruktor

Konstruktor to specjalna metoda klasy, która jest wywoływana jako pierwsza podczas tworzenia obiektu. Konstruktor ma taką samą nazwę jak nazwa klasy i nie zwraca żadnych wartości. Klasa może posiadać wiele konstruktorów, jednak muszą się one różnić typem lub liczbą argumentów. Konstruktor jest wywoływany za pomocą wyrażenia `new` podczas tworzenia obiektu. Konstruktor jest bardzo często wykorzystywany do inicjowania elementów obiektu. Oto przykładowy konstruktor:

```
public class Pojazd
{
    public Pojazd()
    {
        // Ciało konstruktora
    }
}
```

Konstruktor można tworzyć za pomocą snippetu `ctor`. Wpisz słowo `ctor`, a następnie kliknij dwukrotnie klawisz `Tab`.

Konstruktor domyślny to konstruktor, który zostaje utworzony w przypadku, kiedy nie zdefiniowano żadnego konstruktora. Poniżej zostały wymienione właściwości konstruktora domyślnego:

- ◆ posiada modyfikator dostępu `public`,
- ◆ nie posiada argumentów,
- ◆ inicjuje wartości pól klasy zerem, `false` lub `null` — w zależności od typu pól,
- ◆ ma taką samą nazwę jak klasa,
- ◆ nie zwraca wartości.

Podobnie jak metody, konstruktory mogą być przeciążane. Aby przeciążyć konstruktor, wystarczy utworzyć kilka konstruktorów różniących się liczbą parametrów lub typem przyjmowanych parametrów.

Konstruktor zajmuje się inicjalizacją pól składowych klasy. W przypadku gdy pola nie zostaną zainicjowane żadnymi wartościami w konstruktorze, domyślnie zostanie im przypisana domyślna wartość, czyli np. 0. Istnieją dwie metody inicjalizacji pól: przez parametry (listing 1.17) oraz przez wartości domyślne (listing 1.18). Słowo `this` wskazuje na zmienną zadeklarowaną w klasie, a bez `this` na zmienną przekazaną jako parametr do konstruktora (obie mają taką samą nazwę, dlatego konieczne jest użycie `this`; za dobrą praktykę uznaje się rozpoczęwanie nazwy znakiem „_nazwaZmiennej”, np. `_a`).

Więcej informacji na temat słowa kluczowego `this` znajduje się w dalszej części rozdziału.

Listing 1.17. Przykład inicjalizacji pól przez parametry

```
class Liczba
{
    private int a, b, c;

    public Liczba(int a, int b, int c)
```

```
{  
    this.a = a;  
    this.b = b;  
    this.c = c;  
}  
}
```

Listing 1.18. Przykład inicjalizacji pól przez wartości domyślne

```
class Liczba  
{  
    private int a, b, c;  
  
    public Liczba()  
    {  
        a = 1;  
        b = 2;  
        c = 3;  
    }  
}
```

Możliwe jest wywołanie innego konstruktora w miejsce konstruktora domyślnego, przy pomocy listy inicjalizacyjnej. Konstruktor bezparametrowy, który jest wywoływany podczas tworzenia obiektu, sam wywołuje inny konstruktor z domyślnymi parametrami (listing 1.19).

Listing 1.19. Inicjalizacja poprzez listę inicjalizacyjną

```
class Liczba  
{  
    private int a, b, c;  
  
    public Liczba() : this(1,2,3)  
    {  
    }  
  
    public Liczba(int a, int b, int c)  
    {  
        _a = a;  
        _b = b;  
        _c = c;  
    }  
}
```

Kolejność wywoływania konstruktorów:

- ♦ konstruktory klas bazowych w kolejności, w jakiej są dziedziczone (na początku konstruktor klasy bazowej dla wszystkich pozostałych dziedziczących klas, na końcu konstruktor aktualnej klasy);
- ♦ konstruktory dla obiektów zadeklarowanych w ciele klasy;
- ♦ konstruktor klasy³¹.

³¹ <http://msdn.microsoft.com/pl-pl/library/ace5hbzh.aspx>

Destruktor natomiast jest specjalną metodą wywoływaną podczas niszczenia obiektu zawierającej ją klasy. Posiada taką samą nazwę jak klasa, jednak poprzedzoną znakiem ~. Destruktor nie posiada modyfikatora dostępu, nie przyjmuje argumentów i nie zwraca wartości (listing 1.20). W języku C# w zdecydowanej większości przypadków nie ma potrzeby używania destruktora, ponieważ zwalnianiem zasobów zajmuje się *Garbage Collector* („kolekcjoner nieużytków” lub inaczej „odśmiecacz”)³².

Listing 1.20. Przykładowy destruktor

```
public class Pojazd
{
    public ~Pojazd()
    {
        // Ciało destruktora
    }
}
```

Garbage Collector

GC (ang. *Garbage Collector*) to mechanizm, którego zadaniem jest szukanie oraz usuwanie martwych obiektów. Pozwala utrzymać porządek w pamięci oraz zabezpiecza aplikacje przed przepełnieniem lub wyciekiem pamięci.

Zasada działania GC

GC działa automatycznie (niedeterministycznie). Oznacza to brak wpływu użytkownika na to, kiedy zostaną usunięte konkretnie obiekty (GC działa jako wątek w tle). Aby rozpoznać obiekty, które nie są już używane, sprawdzane są referencje do obiektów. GC działa według algorytmu *Mark & Sweep*. Algorytm dzieli się na dwie fazy: *Mark* (oznaczanie) i *Sweep* (usuwanie).

Schemat działania (faza *Mark*): jeśli klasa A posiada referencje do klasy B, to obiekty zostają oznaczone i algorytm przechodzi do kolejnej klasy, czyli B. Algorytm zaczyna działanie od korzeni, którymi są np.: pola statyczne, parametry metod, zmienne lokalne i rejestry CPU. Po przejściu wszystkich elementów rozpoczyna się faza *Sweep*. W tej fazie wszystkie nieoznaczone (nieposiadające referencji) obiekty zostają usunięte.

Podział na generacje (przechowywanie obiektów w pamięci)

Obiekty, które nie zostają usunięte, trafiają do jednej z generacji (GEN0, GEN1, GEN2)³³. Zwolnienie obiektów z GEN0 jest najmniej kosztowne (najszybsze), natomiast zwolnienie obiektów z GEN2 jest najbardziej kosztowne (najwolniejsze).

³² <http://msdn.microsoft.com/pl-pl/library/66x5fx1b.aspx>

³³ <http://msdn.microsoft.com/pl-pl/library/garbage-collector-cz-1.aspx>

Podział na generacje powstał na bazie założeń:

- ♦ najnowsze obiekty mają największą szansę na to, że zostaną najwcześniej zwolnione;
- ♦ ponieważ starsze obiekty nie zostały wcześniej zwolnione, to istnieje mniejsze prawdopodobieństwo, że zostaną zwolnione w aktualnym wywołaniu algorytmu *Mark & Sweep*;
- ♦ wykonywanie algorytmu na całości grafu jest zbyt czasochłonne, dlatego należy podzielić obiekty na grupy.

Obiekty używane przez krótki czas znajdują się w GEN0 (duża szansa na zwolnienie), obiekty, które przetrwały wcześniejsze wywołania algorytmu, awansują do grupy GEN1, natomiast obiekty, które przetrwały dużą liczbę wywołań algorytmu, trafiają do grupy GEN2 (bardzo małe prawdopodobieństwo zwolnienia). Algorytm dla obiektów z GEN0 będzie wywoływany najczęściej, natomiast dla GEN2 najrzadziej. Częstotliwość wywoływania algorytmu oraz liczba oznakowań, aby awansować do kolejnej grupy, są ustalane automatycznie i nie są stałe³⁴.

Duże obiekty (powyżej 85 kB) są przechowywane na osobnej stercie (ang. *Large Objects Heap*), w której obiekty nie zmieniają położenia w pamięci (kosztowna operacja dla dużych obiektów). Duże obiekty należą do GEN2, dlatego należy projektować aplikacje w taki sposób, aby były potrzebne przez dłuższy czas.

Struktury

Struktury są bardzo podobne do klas, podobnie jak klasy mogą zawierać: stałe, pola, metody, właściwości, indeksatory, zdarzenia i konstruktory. Struktury są typem wartości (bezpośrednim), natomiast klasy typem referencyjnym. W strukturze, w przeciwieństwie do klas, nie ma możliwości zdefiniowania konstruktora bezparametrowego, nie można również tworzyć destruktorów. Konstruktor musi mieć przynajmniej jeden parametr, a w ciele konstruktora muszą zostać zainicjowane wszystkie pola struktury. W strukturach nie można także inicjować pól w chwili ich deklaracji. Struktury nie obsługują dziedziczenia, ale mogą implementować interfejsy. Główną przewagą struktur nad klasami jest w większości przypadków dużo większa prędkość działania³⁵. Wartości struktury w przypadku przypisania do nowej zmiennej są kopowane (istnieją dwie odrębne kopie tych samych danych), podczas gdy klasy przechowują wskaźnik do wartości (pojedyncze dane, na które pokazują dwie zmienne). Struktury należy wykorzystywać dla prostych, małych (wielkość instancji do 16 bajtów), krótkotrwałych danych reprezentujących logicznie pojedynczą wartość³⁶. Ogólna składnia deklaracji struktury jest następująca:

³⁴ <http://msdn.microsoft.com/pl-pl/library/0xy59wtx.aspx>

³⁵ <http://msdn.microsoft.com/pl-pl/library/saxz13w4.aspx>

³⁶ <http://msdn.microsoft.com/en-us/library/ms173109.aspx>

```
[modyfikatory] struct Identyfikator [ :lista_bazowa]
{
    składniki_struktury
}
```

gdzie:

- ◆ modyfikatory — dostępne modyfikatory to: new, public, private, protected, internal (opcjonalne);
- ◆ identyfikator — nazwa struktury (wymagane);
- ◆ lista_bazowa — nazwy interfejsów oddzielone przecinkami (opcjonalne);
- ◆ składniki_struktury — lista pól i metod danej struktury.

Przykładowy kod struktury prezentuje listing 1.21.

Listing 1.21. Przykład struktury

```
struct Punkty
{
    public Punkty(double x, double y)
    {
        X = x; Y = y;
    }
    public double X;
    public double Y;
}
```

Interfejsy

Interfejs to jest podobny do klasy, z tą różnicą, że posiada jedynie deklaracje składowych metod, a nie implementuje ich. Nie można utworzyć obiektu, który będzie instancją interfejsu. Klasa natomiast może implementować interfejs. Klasa implementująca interfejs musi posiadać implementacje wszystkich składników interfejsu. Oznacza to, że interfejs mówi klasie, co ma zawierać (jakie metody i właściwości). Co bardzo ważne, klasa może dziedziczyć tylko po jednej klasie, ale może implementować wiele interfejsów jednocześnie. Interfejs może być interfejsem rozszerzającym dla wielu innych interfejsów, jednak żaden interfejs rozszerzający nie może być bardziej dostępny od interfejsu bazowego. Klasa implementująca interfejs może być bardziej dostępna niż interfejs (listing 1.22)³⁷. Ogólna składnia deklaracji interfejsu wygląda następująco:

```
[modyfikatory] interface Identyfikator [ :lista_bazowa]
{
    składniki_interfejsu
}
```

³⁷ <http://msdn.microsoft.com/pl-pl/library/ms173156.aspx>

gdzie:

- ♦ modyfikatory — dostępne modyfikatory to: new, public, private, protected, internal (opcjonalne);
- ♦ identyfikator — nazwa interfejsu zaczynająca się dużą literą I (wymagane);
- ♦ lista_bazowa — lista nazw interfejsów bazowych oddzielonych przecinkami (opcjonalne);
- ♦ składniki_interfejsu — deklaracje metod, właściwości, zdarzeń i indekserów — domyślnie metody są publiczne (opcjonalne).

Listing 1.22. Przykładowy kod interfejsu

```
interface IDziedziczący : IBazowy, IBazowy1
{
    string ZwrocNazwe();
}
```

Interfejs nie może zawierać:

- ♦ stałych,
- ♦ pól,
- ♦ operatorów,
- ♦ konstruktorów i destruktörów,
- ♦ zagnieżdżonych typów.

Jawna implementacja interfejsu

Jawną implementacją interfejsu stosuje się, gdy w interfejsach bazowych znajdują się metody o takich samych nazwach. Wymagane jest wskazanie interfejsu i metody, które się implementuje (listing 1.23).

Listing 1.23. Przykładowy kod jawnej implementacji interfejsu

```
interface IBazowy1
{
    void Metoda1();
}

interface IBazowy2
{
    void Metoda1();
}

class Wykonaj: IBazowy1, IBazowy2
{
    void IBazowy1.Metoda1()
    {
        // Implementacja metody
    }
}
```

Ograniczenia wynikające z jawnej implementacji interfejsu:

- ◆ metody nie mogą być implementowane jako wirtualne;
- ◆ nie można określić modyfikatora dostępu dla metody;
- ◆ wywołanie implementacji może się odbyć tylko poprzez odwołanie do interfejsu;
- ◆ aby odwołać się do metody, trzeba odwoływać się poprzez wybrany interfejs.

Zwalnianie zasobów niezarządzanych

Interfejs IDisposable

Głównym zastosowaniem interfejsu `IDisposable` jest zwalnianie niezarządzanych przez GC zasobów. Niezarządzane zasoby to takie elementy jak np. pliki, strumienie i uchwyty do okien przechowywane przez obiekt. Aby zwolnić zasoby, wywołuje się metodę `Dispose()`. Nie powoduje to zwolnienia pamięci, tylko zwolnienie zasobów, a więc zamknięcie pliku lub połączenia i usunięcie referencji, przez co obiekt nadaje się do natychmiastowego usunięcia. Fizycznym zwalnianiem pamięci zajmuje się GC. W przypadku użycia Entity Framework zalecane jest używanie metody `Dispose()` dla obiektu kontekstu (ang. *context*).

Chodzi o to, aby zasoby były używane jak najkrócej. Każde połączenie z bazą danych po zakończonym odczytce staje się bezużyteczne, ponieważ przy kolejnym zapytaniu jest tworzone osobne połączenie z bazą. Dlatego metoda `Dispose()` pozwala na kontrolowane zwolnienie zasobów. Zasoby te mogą posiadać referencje do innych klas, przez co GC czekałby, aż obiekt nie będzie posiadał żadnych referencji. Referencje zostałyby usunięte dopiero przy kasowaniu obiektu rodzica, w którym znajdują się niezarządzane zasoby, a więc bardzo późno, co powodowałoby niepotrzebne zużycie pamięci. Ponieważ przez dłuższy czas zasoby nie mogły być usunięte, mogą przejść do ostatniej generacji (GEN2), czyli najbardziej kosztownej podczas usuwania. Podobnie wygląda sytuacja w przypadku odczytu plików czy połączenia z internetem.

W przypadku działania GC na zasobach zarządzalnych nie ma wpływu na długość istnienia obiektu, ponieważ GC działa niedeterministycznie. Obiekty są trzymane jak najdłużej i kasowane dopiero w momencie braku zasobów lub przekroczenia limitu zasobów przyjętego przez GC. Przyjęta została zasada, że każda klasa, która działa na zewnętrznych zasobach oraz musi zwalniać zasoby niezarządzane, powinna dziedziczyć po interfejsie `IDisposable`. W przypadku korzystania z kontenera IoC (ang. *Inversion of Control*) nie ma potrzeby używać metody `Dispose()`, ponieważ dla każdego typu obiektu deklaruje się jego cykl życia. Przykładowo dla połączenia z bazą danych jest to `PerWebRequest`. Przy każdym żądaniu HTTP jest tworzony nowy obiekt, a kontener IoC automatycznie wywoła metodę `Dispose()` po zakończonym odczytce danych (listing 1.24).

Listing 1.24. Przykładowy kod prawidłowego użycia `Dispose()`

```
public class MojaKlasa : IDisposable
{
    private bool IsDisposed=false;
```

```
public void Dispose()
{
    Dispose(true);
    GC.SuppressFinalize(this);
}

protected void Dispose(bool Disposing)
{
    if(!IsDisposed)
    {
        if(Disposing)
        {
            // Zwalniaj zasoby zarządzane
        }
        // Zwalniaj zasoby niezarządzane
    }
    IsDisposed=true;
}
~MojaKlasa()
{
    Dispose(false);
}
```

Taka implementacja metody `Dispose()` gwarantuje, że zasoby niezarządzane zostaną wyczyszczone bez względu na to, czy obiekt będzie automatycznie czyszczony przez GC, czy poprzez metodę `Dispose()` wywołaną przez użytkownika. Gdy użytkownik chce zwolnić obiekt, informuje GC, że przejmuje odpowiedzialność za zwolnienie obiektu, za pomocą metody `SuppressFinalize()`. Dzięki temu destruktor nie zostaje wywołany, a użyta zostaje metoda `Dispose()` z parametrem `bool`, który zabezpiecza przed podwójnym wykonaniem tej samej operacji. W przypadku gdy GC będzie chciał zwolnić zasoby jako pierwszy, zostaje wywołany destruktor wywołujący metodę `Dispose()` na zasobach niezarządzanych³⁸.

Słowo kluczowe `using`

Słowo kluczowe `using` posiada różne znaczenie w zależności od miejsca, w którym zostaje użyte. Jeśli jest użyte w przestrzeni nazw lub na początku pliku, pozwala na importowanie bibliotek. Jeśli słowo `using` zostaje użyte we wnętrzu klasy, oznacza blok kodu, dla którego mają zostać zwolnione zasoby po wykonaniu lub pobraniu danych. `using` gwarantuje, że obiekt zostanie prawidłowo zwolniony, poprzez wywołanie metody `Dispose()` i jest skrótem od bloku `try-finally`. Aby było możliwe korzystanie z bloku `using`, musi zostać zaimplementowana metoda `Dispose()`. `using` używa się do zasobów niezarządzanych, pobieranych z bazy danych lub z innych źródeł czy aplikacji. Podczas pobierania danych mogą wystąpić problemy, które spowodują przerwanie zapisu danych (tylko w przypadku `Dispose()`), przez co w pamięci mogą pozostać nieprawidłowe czy też niepotrzebne dane. Użycie `using` zabezpiecza przed pozostawieniem niepotrzebnych danych i zwalnia zasoby najwcześniej, jak to tylko możliwe.

³⁸ <http://msdn.microsoft.com/pl-pl/library/system.idisposable.aspx>

Na listingu 1.25 zaprezentowano przykład użycia `using`, a na listingu 1.26 przedstawiono przetłumaczony kod.

Listing 1.25. Przykład użycia `using`

```
using (MojZasob mojZasob = new MojZasob())
{
    mojZasob.PobierzDane();
}
```

Listing 1.26. Tłumaczone przez CLR na kod

```
{ // Nawiasy oznaczają zasięg zmiennej mojZasob
    MojZasob mojZasob = new MojZasob();
    try
    {
        mojZasob.PobierzDane();
    }
    finally
    {
        // Sprawdzenie, czy zasób nie jest pusty
        if (mojZasob != null)
        {
            // Wywołanie metody Dispose()
            ((IDisposable)mojZasob).Dispose();
        }
    }
}
```

Delegaty, metody anonimowe, wyrażenia lambda i zdarzenia

Delegaty

Delegaty to obiekty, w których wywołuje się inne funkcje (przechowuje referencje do funkcji). Delegat można porównać do wskaźników do funkcji w języku C++. Delegat jest funkcją, która nie ma ciała, ale może wywoływać ciała innych funkcji. Przewagą nad wskaźnikami jest to, że nie można odwoływać się do metody, której sygnatura nie odpowiada sygnaturze delegata. Metoda musi mieć zgodną sygnaturę, co oznacza, że musi przyjmować takie same argumenty i zwracać taki sam typ, jaki podano w deklaracji *delegata* (listing 1.27)³⁹. Ogólna składnia deklaracji delegata:

[modyfikatory] delegate typ Identyfikator ([lista_argumentów]);

gdzie:

- ◆ modyfikatory — dostępne modyfikatory to: new, public, private, protected, internal (opcjonalne);

³⁹ <http://msdn.microsoft.com/pl-pl/library/ms173171.aspx>

- ◆ typ — typ danych zwracanych przez metodę (wymagane, nie może zwracać void);
- ◆ identyfikator — nazwa delegata — musi być różna od nazwy klasy (wymagane);
- ◆ lista_argumentów — argumenty przekazywane przez delegat do metody (opcjonalne).

Listing 1.27. Przykładowy kod korzystający z delegata

```
class Delegat
{
    public delegate int Delegat1 (int arg1, int arg2);

    public int Dodawanie(int arg1, int arg2)
    {
        return arg1 + arg2;
    }

    public int Odejmowanie(int arg1, int arg2)
    {
        return arg1 - arg2;
    }
}

class Program
{
    static void Main()
    {
        Delegat d = new Delegat();

        Delegat.Delegat1 dodawanie = new
            ↪Delegat.Delegat1(d.Dodawanie);
        int wynikDodawania = dodawanie(2, 2);
        System.Console.WriteLine("Wynik dodawania: {0}.",
            ↪wynikDodawania.ToString());

        Delegat.Delegat1 odejmowanie = new
            ↪Delegat.Delegat1(d.Odejmowanie);
        int wynikOdejmowania = odejmowanie(2, 2);
        System.Console.WriteLine("Wynik odejmowania: {0}.",
            ↪wynikOdejmowania.ToString());

        Console.ReadLine();
    }
}
```

Do delegata można przypisać kilka funkcji — wówczas podczas wywołania delegata wywołane zostaną wszystkie te funkcje, które zostały do niego przypisane. Do przypisywania funkcji służy operator `+=`, zaś do usuwania — operator `-=` (listing 1.28).

Listing 1.28. Przykład przypisania funkcji do delegata

```
odejmowanie += d.Dodawanie;
odejmowanie -= d.Dodawanie;
```

Metody anonimowe

Metoda anonimowa to metoda, która nie posiada swojej nazwy. Poza tą różnicą, że nie ma nazwy, metoda anonimowa zachowuje się jak zwykła metoda, czyli przyjmuje argumenty, ma swoje ciało oraz zwraca wartość⁴⁰.

W przykładzie na listingu 1.29 nie dodano metody do delegata poprzez operator +=, tylko od razu przypisano do delegata metodę anonimową, czyli metodę bez nazwy. Metoda anonimowa zostanie wywołana tylko w przypadku korzystania z delegata⁴¹.

Listing 1.29. Przykładowy kod korzystający z metod anonimowych

```

class Delegat
{
    public delegate int Delegat1 (int arg1, int arg2);
}

class Program
{
    static void Main()
    {
        Delegat.Delegat1 dodawanie1 = delegate(int arg1, int arg2)
        {
            return arg1 + arg2;
        };
        System.Console.WriteLine("Wynik dodawania: {0}.",
            → dodawanie1(1,2).ToString());

        Delegat.Delegat1 odejmowanie1 = delegate(int arg1, int arg2)
        {
            return arg1 - arg2;
        };
        System.Console.WriteLine("Wynik odejmowania: {0}.",
            →odejmowanie1(3,4).ToString());

        Console.ReadLine();
    }
}

```

Wyrażenia lambda

Wyrażenia lambda to maksymalnie uproszczone metody anonimowe. Wyrażenia lambda zawierają jedynie listę parametrów oraz ciało metody. Nie mają nazwy oraz nie definiują typu zwracanego⁴².

⁴⁰ <http://msdn.microsoft.com/pl-pl/library/0yw3tz5k.aspx>

⁴¹ W specyfikacji C# 3.0 razem z LINQ zostały wprowadzone wyrażenia lambda, które są jeszcze bardziej uproszczenymi metodami anonimowymi, co spowodowało, że od C# 3.0 metod anonimowych się już praktycznie nie używa.

⁴² <http://msdn.microsoft.com/pl-pl/library/bb397687.aspx>

Metody dodawanie i odejmowanie różnią się nieznacznie od siebie, jednak obie są wyrażeniami lambda. Nie ma potrzeby używania słowa kluczowego return w ciele metody (listing 1.30).

Listing 1.30. Przykładowy kod korzystający z wyrażeń lambda

```
class Delegat
{
    public delegate int Delegat1 (int arg1, int arg2);
}
class Program
{
    static void Main()
    {
        Delegat.Delegat1 dodawanie = (int arg1, int arg2) =>
        {
            return arg1 + arg2;
        };
        System.Console.WriteLine("Wynik dodawania: {0}.",
            ↪dodawanie(1,2).ToString());

        Delegat.Delegat1 odejmowanie = (int arg1, int arg2) =>
            ↪arg1 - arg2;
        System.Console.WriteLine("Wynik odejmowania: {0}.",
            ↪odejmowanie(3,4).ToString());

        Console.ReadLine();
    }
}
```

Wyrażenia lambda:

- ♦ gdy wyrażenie przyjmuje jeden argument, nie trzeba otaczać go nawiasami;
- ♦ gdy ciało funkcji zawiera tylko jedną instrukcję, nie trzeba używać klamerek ani słowa kluczowego return (widoczne w metodzie odejmowanie);
- ♦ słowo delegate w porównaniu z metodą anonimową jest zastąpione znakiem =>;
- ♦ zmienne zdefiniowane w ciele wyrażenia lambda znikają po zakończeniu działania metody;
- ♦ można przekazywać do wyrażeń lambda parametry poprzez referencje i wyjście;
- ♦ nie trzeba podawać typów parametrów dla zmiennych przekazanych przez wartość, natomiast wymagane jest podanie typu parametru dla zmiennych przekazanych przez referencje i wyjście;
- ♦ zwracane wartości muszą pasować do typu danego delegata.

Zdarzenia

Zdarzenia bazują na wywoływaniu metod przez delegaty. Zdarzenie to odpowiedź programu na pewne zmiany w obiektach. Zdarzeniem może być np. kliknięcie przycisku lub poruszenie myszką. Zdarzenia wykorzystują model wydawcy i czytelnika. Oznacza

to, że czytelnikami są obiekty zainteresowane zmianami zachodzącymi u danego wydawcy, czyli innego obiektu. Gdy u wydawcy zajdzie zmiana, to zainteresowani czytelnicy zostaną o tym powiadomieni właśnie poprzez zdarzenie. Za pomocą delegatów zdarzenia wywołują odpowiednie metody u czytelników. Zmiana u wydawcy może wywołać wiele delegatów⁴³. Ogólna składnia deklaracji zdarzenia:

[modyfikatory] event typ Identyfikator;

gdzie:

- ◆ modyfikatory — dostępne modyfikatory to: abstract, new, static, override, virtual, extern, public, private, protected, internal (opcjonalne)
- ◆ typ — nazwa delegata, z którym ma być powiązane zdarzenie (wymagane);
- ◆ identyfikator — nazwa zdarzenia (wymagane).

Na listingu 1.31 po kliknięciu przycisku zostaje wywołany delegat dla danego zdarzenia.

Listing 1.31. Przykładowy kod zdarzenia i delegat

```
class Zdarzenia
{
    public delegate void DelegatZdarzenia();
    public event DelegatZdarzenia Kliknienie;
}
```

Dyrektywy preprocessora

Dyrektywy preprocessora to instrukcje zaczynające się od znaku #, które nie są zakończone średnikiem. Dyrektywy są analizowane przed właściwą komplikacją i dostarczają kompilatorowi dodatkowych informacji, jak należy kompilować dany plik⁴⁴. Poniżej wymieniono dostępne dyrektywy preprocessora w C#:

- ◆ #if,
- ◆ #else,
- ◆ #elif,
- ◆ #endif,
- ◆ #define,
- ◆ #undef,
- ◆ #warning,
- ◆ #error,
- ◆ #line,

⁴³ <http://msdn.microsoft.com/pl-pl/library/awbfdfjh.aspx>

⁴⁴ <http://msdn.microsoft.com/en-us/library/ed8yd1ha.aspx>

- ◆ #region,
- ◆ #endregion,
- ◆ #pragma,
- ◆ #pragma warning,
- ◆ #pragma checksum.

Wyjątki

Wyjątki to obiekty informujące o wystąpieniu pewnej wyjątkowej sytuacji (takiej jak przekroczenie indeksu tablicy). Wyjątek może oznaczać błąd lub nieoczekiwane zachowanie występujące w trakcie wykonywania pewnej części programu. Wyjątki są obiektami, które dziedziczą z klasy wyjątku `System.Exception`. W .NET Framework mechanizm obsługi błędów jest realizowany za pomocą bloku `try-catch`. W przypadku wystąpienia wyjątku w czasie wykonywania kodu z bloku `try` wykonywanie funkcji z tego bloku zostaje przerwane i następuje skok do bloku `catch`. Jako argument w bloku `catch` podaje się typ przechwytywanego wyjątku. Możliwe jest umieszczenie wielu bloków `catch` w kolejności od najbardziej szczegółowych do tych najbardziej ogólnych. Taka kolejność pozwala na najlepsze dopasowanie typu wyjątku, ponieważ zostaje rzucony pierwszy najlepiej pasujący wyjątek.

Blok `finally` to blok zawierający kod, który jest wykonywany zawsze na końcu — bez względu na to, czy wyjątek zostanie rzucony, czy nie. Należy tutaj zamieszczać instrukcje wspólne dla bloku `try` i `catch`, jak np. zamknięcie połączenia z bazą danych. Ponadto blok `finally` jest wykonywany nawet w przypadku użycia instrukcji skoku (`return`, `break`, `continue` itp.). Konstrukcja obsługi wyjątków jest następująca:

```
try
{
    JakiśKod();
}
catch(Exception e)
{
    // Kod wykonywany po wystąpieniu wyjątku
}
finally
{
    // Kod wykonywany na końcu
}
```

Zgłaszanie wyjątków

Tworząc własne typy wyjątków, nie należy dziedziczyć po klasie `System.Exception`. Wyjątki zgłaszane przez klasy z biblioteki klas bazowych .NET pochodzą z `System`.
→`SystemException`, natomiast wyjątki na poziomie aplikacji powinny być dziedziczone po `System.ApplicationException`. Służy to oddzieleniu prywatnych wyjątków aplikacji od wyjątków systemowych⁴⁵.

⁴⁵ <http://msdn.microsoft.com/pl-pl/library/system.exception.aspx>

Lista najczęściej rzucanych wyjątków:

- ◆ NullReferenceException — nie znaleziono obiektu lub brak wartości (null);
- ◆ ArrayTypeMismatchException — typ wartości, jaki chce się przypisać, jest niezgodny z typem docelowym;
- ◆ DivideByZeroException — dzielenie przez zero;
- ◆ IndexOutOfRangeException — przekroczenie indeksu;
- ◆ InvalidCastException — niepoprawne rzutowanie w czasie rzeczywistym;
- ◆ OutOfMemoryException — brak wolnej pamięci;
- ◆ OverflowException — przepelnienie arytmetyczne.

Przepelnienia arytmetyczne

Operatory arytmetyczne takie jak: +, -, *, / mogą zwracać wyniki, które są poza zakresem dostępnym dla danego typu zmiennej. Dlatego operacje nie powinny być wykonywane bez wcześniejszego sprawdzenia argumentów w celu upewnienia się, czy wynik mieści się w zakresie wartości dla uwzględnionych typów danych. W przypadku gdy istnieje prawdopodobieństwo, że może nastąpić przepelnienie, należy użyć bloku try-catch.

Instrukcje checked i unchecked

Instrukcje checked i unchecked służą do kontroli przepelnień arytmetycznych. Instrukcja checked podczas wystapienia przepelnienia rzuca wyjątek (listing 1.32), natomiast instrukcja unchecked nie zwraca wyjątku i ignoruje przepelnienie⁴⁶.

Listing 1.32. Przykładowy kod instrukcji checked

```
class Przepelnienie
{
    public static void Main()
    {
        byte x = 150;
        byte y = 12;
        byte z;

        try
        {
            z = checked((byte)(x * y));
        }
        catch (OverflowException)
        {
            Console.WriteLine("Przepelnienie");
            Console.ReadLine();
        }
    }
}
```

⁴⁶ <http://msdn.microsoft.com/en-gb/library/khy08726.aspx>

Przestrzenie nazw

Przestrzenie nazw służą do uporządkowania kodu i organizacji typów (klas, struktur, interfejsów itp.) w pewne grupy tworzące logiczne całości. Przestrzeń nazw deklaruje się, używając słowa kluczowego namespace. Podstawową przestrzenią nazw w .NET jest System. Każdy — nawet najprostszy — program odwołuje się do tej przestrzeni nazw⁴⁷.

Składnia deklaracji przestrzeni nazw jest następująca:

```
namespace Identyfikator1[.Identyfikator2[...]] {definicje_typów}
```

gdzie:

- ◆ Identyfikator1, Identyfikator2 — nazwy przestrzeni nazw;
- ◆ definicje_typów — klasy, interfejs itp., należące do danej przestrzeni nazw.

Przestrzenie nazw:

- ◆ mogą być wielokrotnie deklarowane w różnych plikach;
- ◆ nie mogą zawierać modyfikatorów dostępu — wszystkie są publiczne;
- ◆ pozwalają definiować zmienne o takiej samej nazwie w różnych przestrzeniach.

Poniżej zaprezentowano kod przykładowej przestrzeni nazw:

```
namespace przestrzen1
{
    class klasa1
    {
        public void Metoda1
        {
        }
    }
}
```

Przestrzeń nazw może zawierać:

- ◆ inne przestrzenie nazw (zagłędzanie),
- ◆ klasy,
- ◆ interfejsy,
- ◆ struktury,
- ◆ delegaty,
- ◆ typy wyliczeniowe.

Zagłędzanie przestrzeni nazw

Przestrzenie można zagłędzać, a więc umieszczać jedną przestrzeń wewnątrz drugiej przestrzeni:

⁴⁷ <http://msdn.microsoft.com/pl-pl/library/0d941h9d.aspx>

```
namespace przestrzen1
{
    namespace przestrzen2
    {
        // Klasa itp.
    }
}
```

Wersja skrócona dla zagnieżdżania:

```
namespace przestrzen1.przestrzen2
{
    // Klasa itp.
}
```

Nazwy niekwalifikowane

W przypadku używania klasy wewnętrznej danej przestrzeni nazw odwołanie do zdefiniowanego w przestrzeni typu odbywa się poprzez skróconą wersję zapisu, bez nazwy przestrzeni, w której się znajduje. Takie odwołanie określa się jako nazwę niekwalifikowaną:

```
namespace przestrzen1
{
    public class Klasa1
    {
        // Ciało klasy
    }

    public class Klasa2
    {
        static void Main()
        {
            Klasa1 NazwaObiektu = new Klasa1();
        }
    }
}
```

Nazwy kwalifikowane

W przypadku używania klasy z innej przestrzeni nazw odwołanie do zdefiniowanego w przestrzeni typu odbywa się poprzez pełną wersję zapisu. Należy zatem poprzedzić nazwę typu nazwą przestrzeni nazw, w której dany typ się znajduje. Takie odwołanie określa się jako nazwę kwalifikowaną:

```
namespace przestrzen1
{
    public class Klasa1
    {
        // Ciało klasy
    }
}

namespace program
{
    static void Main()
```

```
{  
    przestrzen1.Klasa1 NazwaObiektu = new przestrzen1.Klasa1();  
}  
}
```

Dyrektywa using

Dyrektywa `using` jest bardzo przydatnym narzędziem, pozwala bowiem uniknąć konieczności poprzedzania nazwą przestrzeni typów i metod znajdujących się w danym pliku lub przestrzeni nazw. Dyrektywy `using` muszą zostać zadeklarowane na samym początku przestrzeni nazw lub pliku. Najbardziej zewnętrzna przestrzeń nazw, w której znajdują się wszystkie inne przestrzenie nazw, nazywana jest globalną przestrzenią nazw. W przypadku gdy nie zadeklaruje się typu jawnie w przestrzeni nazw, zostanie on przypisany do przestrzeni globalnej — będzie się można do niego odwoływać bezpośrednio z każdej innej przestrzeni nazw. Należy tego unikać, ponieważ nie jest to poprawny styl programowania i kłoci się z ideą przestrzeni nazw⁴⁸.

W zależności od położenia dyrektywy `using` odnosi się ona do innej części kodu (listingi 1.33 – 1.36).

Listing 1.33. Przykład przed użyciem dyrektywy `using`

```
namespace przestrzen1  
{  
    public class Klasa1  
    {  
        System.Collections.Generic.IEnumerable<int> lista;  
        System.Collections.Generic.IEnumerable<int> listal;  
    }  
}
```

Listing 1.34. Przykład po użyciu dyrektywy `using`

```
namespace przestrzen1  
{  
    using System.Collections.Generic;  
    public class Klasa1  
    {  
        IEnumerable<int> lista;  
        IEnumerable<int> listal;  
    }  
}
```

Listing 1.35. Dyrektywa `using` odnosząca się do wszystkich przestrzeni nazw znajdujących się w pliku

```
using System;  
  
namespace przestrzen1.przestrzen2  
{  
    // Klasa itp.  
}
```

⁴⁸ <http://msdn.microsoft.com/pl-pl/library/zhdeawt.aspx>

```
namespace przestrzen2
{
    // Klasa itp.
}
```

Listing 1.36. Dyrektywa using odnosząca się tylko do jednej przestrzeni nazw, w której została zadeklarowana

```
namespace przestrzen1.przestrzen2
{
    using System;
    // Klasa itp.
}
```

W przypadku gdy w dwóch różnych przestrzeniach nazw zadeklarowanych poprzez using znajdują się typy o takich samych nazwach, należy używać nazwy kwalifikowanej.

Dyrektyna using nie jest rekurencyjna, a więc przykładowo dyrektywa using.przestrzen1 nie zawiera w sobie przestrzeni zagnieźdzonych w przestrzeni przestrzen1, czyli np.: using.przestrzen1.przestrzen2. Chcąc korzystać z obu tych przestrzeni, należy zamieścić obie dyrektywy.

Aliases

Za pomocą dyrektywy using można tworzyć również alias, czyli skrót do wybranej przestrzeni nazw⁴⁹. Alias może się odnosić do typu (listing 1.37).

Listing 1.37. Alias odnoszący się do typu

```
namespace przestrzen1.przestrzen2.przestrzen3
{
    public class Klasa1
    {
        // Ciało klasy
    }
}

namespace program
{
    using Alias1 = przestrzen1.przestrzen2.przestrzen3.Klasa1;
    static void Main()
    {
        Alias1 NazwaObiektu = new Alias1();
    }
}
```

Alias może się również odnosić do zagnieżdzonej przestrzeni nazw (listing 1.38).

⁴⁹ <http://msdn.microsoft.com/pl-pl/library/sf0df423.aspx>

Listing 1.38. Alias odnoszący się do przestrzeni nazw

```
namespace przestrzen1.przestrzen2.przestrzen3
{
    public class Klasa1
    {
        // Ciało klasy
    }
}

namespace program
{
    using Alias1 = przestrzen1.przestrzen2.przestrzen3;
    static void Main()
    {
        Alias1.Klasa1 NazwaObiektu = new Alias1.Klasa1();
    }
}
```

Zewnętrzne aliasy

Aby użyć dwóch wersji tej samej biblioteki, należy skorzystać z aliasu zewnętrznego⁵⁰. Te same biblioteki mają takie same nazwy przestrzeni nazw oraz metod, co powoduje dwuznaczność:

```
extern alias Obecna;
extern alias Stara;

class Test
{
    Obecna::Przestrzen.MojaKlasa a; // MojaKlasa pochodząca z nowszej biblioteki
    Stara::Przestrzen.MojaKlasa b; // i klasa pochodząca ze starszej biblioteki.
}
```

Aby móc korzystać z zewnętrznych aliasów, należy w czasie komplikacji powiązać biblioteki z ich aliasami. Pozwoli to rozróżnić biblioteki i powiązać je z konkretnymi aliasami.

Typy, metody, klasy i kolekcje uogólnione (generyczne)

Typy generyczne (lub inaczej nazywając, ogólne, uogólnione, szablonowe) zostały wprowadzone w C# 2.0. Nie ma prawdopodobnie dobrego tłumaczenia na język polski dla typów generics, dlatego korzysta się z określenia generyczne. Wszystkie typy dziedziczą po klasie System.Object i w zwykłych kolekcjach dane były przechowywane jako typ object. Wymagało to za każdym razem wykonywania pakowania i wypakowywania danych w kolekcji. Typy generyczne pozwalają na określenie typu przechowywanych danych w czasie deklaracji. Kolekcje standardowe (niegeneryczne) są bardzo rzadko używane, ponieważ mają zastosowanie tylko wtedy, gdy nie jest znany typ danych podczas komplikacji lub istnieje konieczność zapisywania danych różnego typu w jednej kolekcji.

⁵⁰ <http://msdn.microsoft.com/pl-pl/library/ms173212.aspx>

Metody generyczne

Istnieje możliwość deklarowania metod generycznych. Metoda generyczna może przyjmować parametry oraz zwracać wartość różnego typu — w zależności od wywołania. Metoda przedstawiona na listingu 1.39 zwraca typ generyczny i pobiera parametr generyczny. Jeśli poda się jako parametr typ `string`, zostanie również zwrócony typ `string`, natomiast jeśli jako parametr poda się typ `int`, metoda zwróci wartość typu `int`.

Listing 1.39. Przykładowy kod metody generycznej

```
class Program
{
    static void Main(string[] args)
    {
        var inty = GenerycznaMetoda<int>(12);
        Console.WriteLine("Metoda z parametrem int: " + inty);

        var stringi = GenerycznaMetoda<string>("napis");
        Console.WriteLine("Metoda z parametrem string: " +
                         stringi);

        Console.ReadKey();
    }

    public static T GenerycznaMetoda<T>(T parametr)
    {
        return parametr;
    }
}
```

W przykładzie na listingu 1.39 podano, jaki typ jest przekazywany do metody, jednak nie trzeba tego robić. Kompilator na podstawie parametru sam potrafi rozpoznać typ przekazywanego parametru. Efekt działania metod będzie taki sam. Przykład na listingu 1.40 prezentuje wywołanie tej samej metody z podaniem i bez podania typu danych.

Listing 1.40. Rozwinięcie przykładu z podaniem i bez podania typu danych

```
var stringi = GenerycznaMetoda<string>("napis");
Console.WriteLine("Metoda z podaniem typu parametru: " + stringi);

var stringi1 = GenerycznaMetoda("napis");
Console.WriteLine("Metoda bez podania typu parametru: " + stringi1);
```

Przyjęło się, że parametry generyczne oznacza się literą `T`, jednak może to być dowolna inna litera lub słowo. Ważne są tutaj natomiast nawiasy `< >`, które mówią, że są to typy generyczne.

Klasy generyczne

Klasy generyczne pozwalają na tworzenie obiektów tej samej klasy dla różnych typów danych. Dopiero przy tworzeniu obiektu klasy generycznej podaje się typ danych, na którym będzie operował (listing 1.41).

Listing 1.41. Przykładowy kod klasy generycznej

```
class Program
{
    static void Main(string[] args)
    {
        KlasaGeneryczna<int> obiektKlasyGenerycznej =
            ➔ new KlasaGeneryczna<int>();
        obiektKlasyGenerycznej.Dodaj(12);

        KlasaGeneryczna<string> obiektKlasyGenerycznej1 =
            ➔ new KlasaGeneryczna<string>();
        obiektKlasyGenerycznej1.Dodaj("napis");
        Console.ReadKey();
    }

    public class KlasaGeneryczna<T>
    {
        public void Dodaj(T parametr)
        {
            Console.WriteLine("Wynik metody Dodaj: " +
                ➔parametr);
        }
    }
}
```

Możliwe jest ograniczenie typów, jakie może przyjąć klasa generyczna. Aby ograniczyć typ, należy użyć słowa kluczowego `where` (listing 1.42).

Listing 1.42. Przykład ograniczania typu

```
public class KlasaGeneryczna<T> where T : struct
{
    public void Dodaj(T parametr)
    {
        Console.WriteLine("Wynik metody Dodaj: " + parametr);
    }
}
```

W przypadku zaprezentowanym na listingu 1.42 nie będzie możliwe przekazanie jako parametru typu `string`, ponieważ `string` jest klasą, a nie strukturą, jak typ `int`. Aby można było przekazać typ `string`, należy użyć następującego kodu:

```
public class KlasaGeneryczna<T> where T : class
```

Możliwe jest również przekazanie kilku parametrów generycznych (listing 1.43).

Listing 1.43. Przykład przekazywania wielu parametrów generycznych

```
class Program
{
    static void Main(string[] args)
    {
        KlasaGeneryczna<int,string> obiektKlasyGenerycznej =
            ➔ new KlasaGeneryczna<int,string>();
```

```

        obiektKlasyGenerycznej.Dodaj(12, "napis");

        KlasaGeneryczna<int, int> obiektKlasyGenerycznej1 =
            ➔ new KlasaGeneryczna<int, int>();
        obiektKlasyGenerycznej1.Dodaj(12, 12);

        Console.ReadKey();
    }

    public class KlasaGeneryczna<T,T1> where T : struct
    {
        public void Dodaj(T parametr, T1 parametr1)
        {
            Console.WriteLine("Wynik metody Dodaj: " + parametr +
                ➔ " " + parametr1);
        }
    }
}

```

Kolekcje generyczne i interfejsy

Kolekcje generyczne znajdują się w przestrzeni nazw System.Collections.Generic. Przestrzeń ta zawiera klasy i interfejsy definiujące kolekcje, które pozwalają na tworzenie typowanych kolekcji generycznych. W stosunku do zwykłych kolekcji kolekcje generyczne zapewniają większe bezpieczeństwo przechowywanych typów oraz większą wydajność, ponieważ nie jest wykonywane pakowanie i wypakowywanie elementów. W przestrzeni nazw System.Collections.Generic znajdują się nie tylko gotowe klasy z kolekcjami, których można użyć, ale również interfejsy. Interfejsy nie posiadają ciała metod, zawierają jedynie deklaracje metod. Nie można tworzyć obiektu, a więc instancji interfejsu. Zatem dlaczego interfejsy znajdują się razem z kolekcjami w tej samej przestrzeni nazw? Kolekcje mogą zawierać różne typy, a więc tworzenie implementacji dla wszystkich typów, jakie człowiek może wymyślić, nie jest możliwe. Dlatego zamiast konkretnej implementacji tworzone są interfejsy, które mówią, jak ma wyglądać struktura danej kolekcji. Implementacje metod kolekcji tworzy się samodzielnie w zależności od typu przechowywanego w tej kolekcji oraz metod wymaganych przez interfejs kolekcji. Przykładowo można przypisywać różne obiekty do tego samego interfejsu. Nie jest ważne, jak dany obiekt będzie się zachowywał, musi tylko mieć zaimplementowane metody, które są wymagane w interfejsie. Pozwala to w przyszłości podmienić implementację dla obiektu, a kod dalej będzie poprawny, ponieważ z punktu widzenia interfejsu wymagane metody się nie zmieniły, czyli pobierają i zwracają te same typy⁵¹.

Interfejs **IDictionary<TKey, TValue>** — słownik

Kolekcja **IDictionary** działa podobnie do **HashTable**, ma tylko jedną zasadniczą różnicę — jest to kolekcja generyczna, a więc trzeba podać typ przechowywanych danych. Dodatkowo trzeba podać typ klucza oraz typ wartości przechowywanych w słowniku. **HashTable** pozwalało na dodanie dowolnego typu do kolekcji. Słowniki są szybsze od kolekcji **HashTable**, ponieważ nie wykonują pakowania do obiektów — typ danych jest

⁵¹ <http://msdn.microsoft.com/pl-pl/library/System.Collections.Generic.aspx>

określony. Aby korzystać ze słowników w trybie bezpiecznym wątkowo (ang. *Thread Safety*), należy użyć słownika `ConcurrentDictionary < TKey, TValue >` zamiast `Dictionary < TKey, TValue >`⁵².

Interfejs `IEnumerable<T>`

Kolekcja `IEnumerable<T>` to najbardziej podstawowa kolekcja tylko do odczytu. Pozwala na użycie pętli `foreach`. Nie pozwala dodawać, usuwać ani modyfikować elementów kolekcji. Parametr kolekcji jest kowariantny — można użyć określonego typu lub typu pochodnego. `IEnumerable<T>` ma jedną zasadniczą wadę — w przypadku gdy chce się pobrać tylko jeden element kolekcji, należy pobierać wszystkie elementy od początku aż do elementu, który jest potrzebny. Aby na przykład pobrać element numer 3, trzeba pobrać także elementy numer 1 i 2.

Kolekcja `IQueryable<T>` pozwala na odczyt danych dopiero w momencie, w którym są one potrzebne (ang. *Lazy Loading*). Nie wymaga pobierania całej kolekcji od razu⁵³.

Interfejs `ICollection<>`

Kolekcja `ICollection<T>` to kolekcja dziedzicząca po `IEnumerable<T>`. W przeciwieństwie do `IEnumerable<T>` pozwala dodawać oraz usuwać elementy kolekcji. Wykorzystywana jest w Entity Framework do opisu relacji pomiędzy tabelami. W przeciwieństwie do `IList<T>` nie posiada indeksów i nie pozwala na zarządzanie kolejnością elementów⁵⁴. Metody dostępne w `ICollection<T>` prezentuje tabela 1.10.

Tabela 1.10. Metody z `ICollection<T>`

Nazwa	Opis
Add	Dodaje element do <code>ICollection<T></code>
Clear	Usuwa wszystkie elementy z <code>ICollection<T></code>
Contains	Określa, czy <code>ICollection<T></code> zawiera określoną wartość
CopyTo	Kopiuje elementy <code>ICollection<T></code> do tablicy
GetEnumerator	Zwraca enumerator do przechodzenia po kolejnych elementach
Remove	Usuwa pierwsze wystąpienie określonego obiektu z <code>ICollection<T></code>

Interfejs `IList<>`

Kolekcja `IList<T>` dziedziczy po `ICollection<T>` oraz `IEnumerable<T>`. W porównaniu do `ICollection<T>` pozwala na manipulowanie danymi poprzez indeks, dzięki czemu można zarządzać kolejnością elementów w kolekcji i ją sortować⁵⁵.

⁵² <http://msdn.microsoft.com/pl-pl/library/xjhwa508.aspx>

⁵³ <http://msdn.microsoft.com/pl-pl/library/9eekhta0.aspx>

⁵⁴ <http://msdn.microsoft.com/pl-pl/library/92t2ye13.aspx>

⁵⁵ <http://msdn.microsoft.com/pl-pl/library/5y536ey6.aspx>

Właściwości dostępne w kolekcji `IList<T>` prezentuje tabela 1.11.

Tabela 1.11. Właściwości z `IList<T>`

Nazwa	Opis
Count	Zwraca liczbę elementów w <code>ICollection<T></code>
IsReadOnly	Wskazuje, czy <code>ICollection<T></code> jest tylko do odczytu
Item	Pobiera lub ustawia element o określonym indeksie

Metody dostępne w `IList<T>` prezentuje tabela 1.12.

Tabela 1.12. Metody z `IList<T>`

Nazwa	Opis
Add	Dodaje element do <code>ICollection<T></code>
Clear	Usuwa wszystkie elementy z <code>ICollection<T></code>
Contains	Określa, czy kolekcja posiada określoną wartość
CopyTo	Kopiuje element z <code>ICollection<T></code> do tablicy
GetEnumerator	Zwraca enumerator do przechodzenia po kolejnych elementach
IndexOf	Indeks elementu <code>IList<T></code>
Insert	Wstawia element do <code>IList<T></code>
Remove	Usuwa pierwsze wystąpienie obiektu <code>ICollection<T></code>
RemoveAt	Usuwa z <code>IList<T></code> element o konkretnym indeksie

Interfejs `IQueryable<T>`

Kolekcja `IQueryable<T>` jest częścią przestrzeni nazw `System.Linq`. Jest bardzo podobna do `IEnumerable<T>`, jednak jest między nimi jedna zasadnicza różnica — `IQueryable<T>` nie pobiera elementów kolekcji po kolei; pobiera tylko te dane, które spełniają warunki. Aby pobrać pracowników z bazy danych, którzy zarabiają więcej niż np. 4000 zł, `IEnumerable<T>` pobralaby wszystkich pracowników do pamięci (za pomocą *LINQ to Object*), a następnie wybrała tych, którzy zarabiają więcej niż 4000 zł. `IQueryable<T>` pobierze natomiast tylko tych pracowników, którzy zarabiają więcej niż 4000 zł (dla relacyjnej bazy danych za pomocą *LINQ to SQL*). Przy większych kolekcjach daje to kolosalną różnicę w wydajności. Należy tutaj zauważać, że `IQueryable<T>` jest przeznaczone do zewnętrznych źródeł danych, takich jak Web serwisy czy bazy danych, i jest częścią LINQ, natomiast `IEnumerable<T>` służy do operowania na danych już pobranych i przetrzymywanych w pamięci. `IQueryable<T>`, w przeciwieństwie do `IEnumerable<T>`, wspiera „leniwe ładowanie” (ang. *Lazy Loading*)⁵⁶.

Tabelę 1.13 czyta się z góry na dół, a więc np.: `ICollection<T>` dziedziczy po `IEnumerable<T>`.

⁵⁶ <http://msdn.microsoft.com/pl-pl/library/bb351562.aspx>

Tabela 1.13. Hierarchia dziedziczenia

	IEnumerable<T>	ICollection<T>	IQueryable<T>	IList<T>
IEnumerable<T>	—	X	X	X
ICollection<T>		—		X
IList<T>				—

Wyrażenia regularne

Wyrażenia regularne to wzorce, które opisują łańcuchy symboli. Za pomocą wyrażeń regularnych można wyodrębnić z dowolnego tekstu lub kodu interesujące części. Wyrażenia regularne są tak jakby nakładane na łańcuchy znaków w celu odnalezienia pasujących fragmentów tekstu (tabela 1.14)⁵⁷. Wyrażenie regularne składa się z dwóch typów znaków:

- ◆ literał — czyli znak, który ma się znaleźć w łańcuchu znaków,
- ◆ metaznak — znak specjalny, który jest informacją dla analizatora wyrażenia regularnego.

Wykaz wybranych, najważniejszych symboli przedstawiono w tabeli 1.14.

Przykładowy kod programu sprawdzającego poprawność adresu e-mail przedstawiono na listingu 1.44. W programie zostały użyte biblioteka *Regex* oraz operator trójargumentowy, który w zależności od poprawności adresu e-mail dodaje przedrostek *nie* do słowa *poprawny*.

Tabela 1.14. Znaczenie znaków w wyrażeniach regularnych

Znak	Znaczenie
.	Dowolny znak oprócz znaku nowego wiersza
^	Początek wiersza
\$	Koniec wiersza
*	Zero lub więcej wystąpień danego zestawu znaków (wyrażeń)
?	Zero lub jedno wystąpienie danego zestawu znaków
+	Jedno lub więcej wystąpień danego zestawu znaków
[a-c]	Dowolny znak ze zbioru znajdującego się wewnątrz nawiasów kwadratowych
[^ ab]	Wszystkie znaki oprócz tych z zestawu znajdujących się wewnątrz nawiasów kwadratowych
	Lub
{x}	Dokładnie X wystąpień danego zestawu znaków (wyrażeń)
{x,}	Co najmniej X wystąpień danego zestawu znaków (wyrażeń)
{x,y}	Co najmniej X wystąpień i maksymalnie Y wystąpień danego zestawu znaków
\d	Cyfra

⁵⁷ <http://msdn.microsoft.com/en-us/library/system.text.regularexpressions.regex.aspx>,
<http://msdn.microsoft.com/en-us/library/System.Text.RegularExpressions.aspx>

\znak	Oznacza ten znak
a b	Alternatywa — a lub b
\b	Granica słowa
\n	Nowa linia

Listing 1.44. Przykład weryfikacji adresu e-mail

```
class Program
{
    static void Main(string[] args)
    {
        Regex regEmail;
        regEmail = new Regex(@"^@[a-z][a-z0-9_]*@[a-z0-9]*\.[a-z]{2,3}$");

        Console.WriteLine("Proszę podać adres e-mail:");

        string napis = Console.ReadLine();

        // Podany przez użytkownika
        Console.WriteLine(
            String.Format("Podany adres: {0} to {1}poprawny adres
              ↪e-mail", napis, regEmail.IsMatch(napis) ? "" : "nie"));

        // Niepoprawny
        Console.WriteLine(
            String.Format("{0} to {1}poprawny adres e-mail",
              ↪"e-mail@wp.1234", regEmail.IsMatch("e-mail@wp.1234")
              ↪? "" : "nie"));

        Console.ReadKey();
    }
}
```

Data i czas

W C# są do dyspozycji dwie struktury i jedna klasa pozwalające na operacje na dacie i czasie:

- ◆ **DateTime** — struktura służąca do przechowywania informacji o konkretnym punkcie w czasie;
- ◆ **DateTimeOffset** — struktura, która reprezentuje punkt w czasie, wyrażony jako data i godzina; zawiera właściwość określającą różnicę w stosunku do skoordynowanego czasu uniwersalnego (UTC);
- ◆ **TimeSpan** — klasa, która reprezentuje przedział czasu — czas trwania lub czas, który upłynął, mierzony jako dodatnia lub ujemna liczba dni, godzin, minut, sekund i ułamków sekundy⁵⁸.

⁵⁸ <http://msdn.microsoft.com/pl-pl/library/system.datetime.aspx>, <http://msdn.microsoft.com/pl-pl/library/system.datetimeoffset.aspx>, <http://msdn.microsoft.com/pl-pl/library/system.timespan.aspx>

Właściwości dostępne w strukturze DateTime:

- ◆ Day — zwraca dzień miesiąca,
- ◆ DayOfWeek — zwraca dzień tygodnia,
- ◆ DayOfYear — zwraca dzień roku,
- ◆ Hour — zwraca godzinę,
- ◆ Minute — zwraca minutę,
- ◆ Second — zwraca sekundę,
- ◆ Month — zwraca miesiąc,
- ◆ Year — zwraca rok,
- ◆ Now — zwraca obiekt DateTime ustawiony na bieżącą datę i godzinę,
- ◆ Today — zwraca aktualną datę.

Metody dostępne w strukturze DateTime:

- ◆ Add() — dodaje określony przedział czasu (TimeSpan) i zwraca nowy obiekt DateTime,
- ◆ AddDays() — dodaje do daty określoną liczbę dni,
- ◆ AddHours() — dodaje do daty określoną liczbę godzin,
- ◆ AddMinutes() — dodaje do daty określoną liczbę minut,
- ◆ AddMonths() — dodaje do daty określoną liczbę miesięcy,
- ◆ AddYears() — dodaje do daty określoną liczbę lat,
- ◆ AddSeconds() — dodaje do daty określoną liczbę sekund,
- ◆ Compare() — porównuje dwie daty,
- ◆ DaysInMonth() — zwraca liczbę dni w bieżącym miesiącu i roku,
- ◆ Equals() — sprawdza, czy daty są takie same,
- ◆ IsLeapYear() — sprawdza, czy aktualny rok jest przestępny,
- ◆ Parse() — konwertuje łańcuch string na typ DateTime,
- ◆ ToString() — konwertuje DateTime na string,
- ◆ ToUniversalTime() — konwertuje DateTime na czas UTC.

Poniżej wymieniono operatory (przeładowane) dostępne dla typu DateTime:

- ◆ + — dodawania,
- ◆ - — odejmowania,
- ◆ == — równości,
- ◆ > — większy od,

- ◆ >= — większy bądź równy,
- ◆ != — różny,
- ◆ < — mniejszy,
- ◆ <= — mniejszy bądź równy.

Operacje wejścia, wyjścia, foldery i pliki

W przestrzeni nazw System.IO znajdują się klasy pozwalające na odczyt i zapis do plików, strumienie danych oraz operacje na plikach i folderach. Ponieważ biblioteka ta jest bardzo rozbudowana, przedstawiono tylko najważniejsze klasy i ich funkcjonalność⁵⁹.

Najpopularniejsze klasy z przestrzeni System.IO to:

- ◆ Directory — zawiera metody statyczne pozwalające na operacje na folderach, takie jak:
 - ◆ tworzenie folderów,
 - ◆ przenoszenie folderów,
 - ◆ listowanie folderów,
 - ◆ usuwanie folderów;
- ◆ DirectoryInfo — zawiera metody instancjacyjne (metody, które wymagają, aby były wywoływanie na obiekcie, czyli instancji klasy) pozwalające na operacje na folderach, takie jak:
 - ◆ tworzenie folderów,
 - ◆ przenoszenie folderów,
 - ◆ listowanie folderów,
 - ◆ usuwanie folderów;
- ◆ DriveInfo — zapewnia dostęp do informacji o dysku;
- ◆ File — zawiera metody statyczne pozwalające na operacje na plikach, takie jak:
 - ◆ tworzenie plików,
 - ◆ przenoszenie plików,
 - ◆ otwieranie plików,
 - ◆ usuwanie plików,
 - ◆ kopianie plików;

⁵⁹ <http://msdn.microsoft.com/pl-pl/library/system.io.aspx>

- ♦ `FileInfo` — zawiera metody instancyjne pozwalające na operacje na plikach, takie jak:
 - ♦ tworzenie plików,
 - ♦ przenoszenie plików,
 - ♦ otwieranie plików,
 - ♦ usuwanie plików,
 - ♦ kopiowanie plików;
- ♦ `FileStream` — udostępnia obsługę strumieni dla plików, wspiera synchroniczne i asynchroniczne operacje odczytu i zapisu;
- ♦ `StreamReader` — wczytuje dane ze strumienia;
- ♦ `StreamWriter` — zapisuje dane do strumienia;
- ♦ `Path` — udostępnia informacje o ścieżkach do plików lub katalogów.

Pozostałe elementy języka i nowości w wersji C# 5.0

Mechanizm refleksji i atrybuty

Refleksja to mechanizm, który pozwala na dostęp do metadanych klas lub obiektów. Metadane to informacje przechowywane w plikach wykonywalnych, czyli `.dll` lub `.exe`, które opisują pola, właściwości i metody znajdujące się w tych plikach. Do tworzenia metadanych służą atrybuty. Atrybuty pozwalają dodawać jako metadane dodatkowe

informacje na temat klas lub metod do plików wykonywalnych. Atrybuty są dostarczane przez środowisko CLR. Można również tworzyć własne atrybuty wykorzystywane później w mechanizmie refleksji. Atrybuty muszą być związane z pewnymi elementami programu — elementy te nazywa się adresatami atrybutu. Dostępne adresaty atrybutów prezentuje tabela 1.15.

Tabela 1.15. Adresaty atrybutów

Nazwa adresata	Zastosowanie
All	Do dowolnego elementu: pliku wykonywalnego, konstruktora, metody, klasy, zdarzenia, pola, właściwości czy struktury
Assembly	Do pliku wykonywalnego
Class	Do klasy
Constructor	Do konstruktora
Delegate	Do delegata

Enum	Do wyliczania
Event	Do zdarzenia
Field	Do pola
Interface	Do interfejsu
Method	Do metody
Parameter	Do parametru metody
Property	Do właściwości
ReturnValue	Do zwracanej wartości
Struct	Do struktury

Atrybuty

Atrybut obsoleto został zastosowany do metody o nazwie StaraMetoda(). Atrybut ten jest stosowany do tych części programu, które nie są już używane.

Listing 1.45. Przykładowy kod z wykorzystaniem atrybutu obsolete

```
static void Main(string[] args)
{
    StaraMetoda();
}

[Obsolete("Przestarzała metoda, użyj NowaMetoda")]
public static void StaraMetoda()
{
    // Ciało starej metody
}

public static void NowaMetoda()
{
    // Ciało nowej metody
}
```

Ostrzeżenie zwrócone w czasie komplikacji kodu (listing 1.45):

```
warning CS0618: 'Aplikacja1.Program.StaraMetoda()' is obsolete:
'Przestarzała metoda, użyj NowaMetoda'
```

Bardzo często stosowanym atrybutem jest Serializable, który pozwala na zapis (w tym przypadku klasy) do pliku na dysku (listing 1.46). Wynik działania prezentuje rysunek 1.1.

Listing 1.46. Przykład użycia atrybutu Serializable

```
[Serializable]
public class Program
{
    public string pole1 = "pole1";
    public int pole2 = 12;
    // Ciało klasy
}
```

```
class Wykonaj
{
    public static void Main()
    {
        Program p = new Program();
        Console.WriteLine(SerializeToXml(p));

        Console.ReadLine();
    }

    static string SerializeToXml(Program p)
    {
        StringWriter writer = new StringWriter();

        XmlSerializer serializer = new
            XmlSerializer(typeof(Program));
        serializer.Serialize(writer, p);

        return writer.ToString();
    }
}
```

Rysunek 1.1.
Wynik działania
programu
z serializacją

```
<?xml version="1.0" encoding="utf-16"?>
<Program xmlns:xsd="http://www.w3.org/2001/XMLSchema"
          |           xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
    <pole1>pole1</pole1>
    <pole2>12</pole2>
</Program>
```

Proces serializacji jest odwracalny, a więc z wygenerowanego pliku można utworzyć klasę. Serializacja przydaje się np. do przekazywania obiektów pomiędzy klientem a serwerem, np. poprzez protokół SOAP. Możliwe jest definiowanie własnych atrybutów. Aby utworzyć własny atrybut, należy zaimplementować własną klasę dziedziczącą po klasie System.Attribute.

Do odczytu metadanych, atrybutów, sprawdzania typu oraz dynamicznego tworzenia typów i wywoływania metod w czasie działania programu służy mechanizm refleksji. Listing 1.47 prezentuje program, który zwraca nazwy typów dostępnych w aktualnie wykonywanym programie.

Listing 1.47. Przykładowy program korzystający z refleksji

```
public class Testowa
{
    public int pole1;
}

public class Wykonaj
{
    public static void Main()
    {
        Assembly assembly = Assembly.GetExecutingAssembly();
```

```

Type[] typ = assembly.GetTypes();
int i = 0;
foreach (Type t in typ)
{
    Console.WriteLine(t.ToString());
    i++;
}
Console.ReadLine();
}
}

```

Mechanizm refleksji ma bardzo szerokie zastosowanie, dlatego nie sposób opisać wszystkich dostępnych metod. Ponieważ mechanizm refleksji działa w czasie pracy programu i nie jest wcześniej kompilowany, działa dużo wolniej niż wywoływanie zwykłych metod. Z refleksji należy korzystać tylko wtedy, gdy jest to konieczne⁶⁰.

IEnumerable a IEnumerator

Aby móc korzystać z pętli foreach, trzeba zaimplementować na kolekcji interfejs IEnumerable. Interfejs IEnumerable ma tylko jedną metodę GetEnumerator(), która zwraca IEnumerator. IEnumerator to mechanizm pozwalający na poruszanie się po kolekcji i zwracanie aktualnej pozycji w kolekcji. Poniżej zaprezentowano kod interfejsu IEnumerator:

```

public interface IEnumerator
{
    object Current { get; }
    bool MoveNext();
    void Reset();
}

```

Interfejs IEnumerator:

- ◆ zawiera właściwość Current — pobiera aktualny element w kolekcji;
- ◆ zawiera metodę MoveNext(), która przesuwa enumerator do kolejnego elementu kolekcji;
- ◆ zawiera metodę Reset() ustawiającą enumerator na początek, który znajduje się przed pierwszym elementem kolekcji (kolekcja może być pusta, dlatego enumerator musi być ustawiony przed pierwszym elementem).

W przypadku gdy działa się na własnych, niestandardowych kolekcjach, należy zaimplementować wszystkie elementy z interfejsu IEnumerator. Bez implementacji tego interfejsu nie będzie możliwe korzystanie z pętli foreach.

Listing 1.48 przedstawia zależność pomiędzy tymi interfejsami.

⁶⁰ <http://msdn.microsoft.com/pl-pl/library/f7ykdhsy.aspx>, <http://msdn.microsoft.com/pl-pl/library/z0w1kezw.aspx>

Listing 1.48. Zależności pomiędzy interfejsami

```
public class Program
{
    public static void Main()
    {
        I Enumerable<string> enumerable = new string[] { "A", "B",
            ↗ "C" };

        // Z użyciem foreach dostępnego w I Enumerable
        foreach (string s in enumerable)
            Console.WriteLine(s);

        // Z użyciem iteratora
        IEnumerator<string> enumerator = enumerable.GetEnumerator();
        while (enumerator.MoveNext())
        {
            string s = enumerator.Current;
            Console.WriteLine(s);
        }
        Console.ReadLine();
    }
}
```

Interfejs `IEnumerable` pozwala w bardzo prosty sposób za pomocą pętli `foreach` iterować po kolekcji, wykorzystując przy tym interfejs `IEnumerable` do obsługi tej pętli.

W przypadku interfejsu generycznego istnieje jedna różnica pomiędzy `IEnumerator` a `IEnumerator<T>`. `IEnumerator` posiada metodę `Reset()`, natomiast `IEnumerator<T>` zawiera zarówno metodę `Reset()`, jak i dziedziczy po interfejsie `IDisposable`, który zawiera metodę `Dispose()` służącą do zwalniania zasobów.

Iteratory i słowo kluczowe `yield return`

Iteratory to nowość wprowadzona w C# 2.0. Iteratory pozwalają uprościć iterowanie, czyli przechodzenie po kolejnych elementach kolekcji. Dzięki iteratorom nie trzeba implementować interfejsu `IEnumerator` wraz z jego elementami (`Reset()`, `MoveNext()`, `Current`). Kompilator automatycznie generuje te metody, gdy używa się słowa kluczowego `yield return`, służącego do zwracania wartości z pętli (listing 1.49). Gdy używa się iteratorów, można zaimplementować metodę `GetEnumerator()` prosto w klasie bez implementacji całej klasy `Enumerator`⁶¹.

Listing 1.49. Przykładowy kod z użyciem `yield return`

```
public class Program
{
    private static I Enumerable<int> PobierzDane()
    {
        Console.WriteLine("Początek metody PobierzDane.");
    }
```

⁶¹ <http://msdn.microsoft.com/pl-pl/library/dscyy5s0.aspx>

```
List<int> list = new List<int>();
for (int i = 1; i <= 10; i++)
{
    Console.WriteLine("Aktualna wartość: {0}", i);
    yield return i;
}

Console.WriteLine("Koniec metody PobierzDane.");
}

public static void Main()
{
    Console.WriteLine("Pobranie danych.");
    IEnumerable<int> data = PobierzDane();

    Console.WriteLine("Rozpoczęcie pętli.");
    foreach (int i in data)
    {
        Console.WriteLine("Odczyt wartości w pętli: {0}", i);
        if (i > 6)
            break;
    }

    Console.WriteLine("Koniec programu.");
    Console.ReadLine();
}
}
```

Można też nie używać słowa kluczowego `yield return` (listing 1.50). W tabeli 1.16 porównano działanie metody `PobierzDane()` z użyciem i bez użycia słowa kluczowego `yield return`.

Listing 1.50. Kod metody `PobierzDane()` bez użycia `yield return`

```
private static IEnumerable<int> PobierzDane()
{
    Console.WriteLine("Początek metody PobierzDane.");

    List<int> list = new List<int>();

    for (int i = 1; i <= 10; i++)
    {
        Console.WriteLine("Aktualna wartość: {0}", i);
        list.Add(i);
    }

    Console.WriteLine("Koniec metody PobierzDane.");
    return list;
}
```

Podsumowując działania metod, metoda korzystająca z `yield` zwróciła sekwencyjnie dane w pętli i je wyświetliła, natomiast metoda z listą pobrała całą listę, a następnie wyświetliła te dane, które były potrzebne⁶².

Tabela 1.16. Porównanie działania metod

Z użyciem <code>yield return</code>	Bez użycia <code>yield return</code>
Pobranie danych.	Pobranie danych.
Rozpoczęcie pętli.	Początek metody <code>PobierzDane</code> .
Początek metody <code>PobierzDane</code> .	Aktualna wartość: 1
Aktualna wartość: 1	Aktualna wartość: 2
Odczyt wartości w pętli: 1	Aktualna wartość: 3
Aktualna wartość: 2	Aktualna wartość: 4
Odczyt wartości w pętli: 2	Aktualna wartość: 5
Aktualna wartość: 3	Aktualna wartość: 6
Odczyt wartości w pętli: 3	Aktualna wartość: 7
Aktualna wartość: 4	Aktualna wartość: 8
Odczyt wartości w pętli: 4	Aktualna wartość: 9
Aktualna wartość: 5	Aktualna wartość: 10
Odczyt wartości w pętli: 5	Koniec metody <code>PobierzDane</code> .
Aktualna wartość: 6	Rozpoczęcie pętli.
Odczyt wartości w pętli: 6	Odczyt wartości w pętli: 1
Aktualna wartość: 7	Odczyt wartości w pętli: 2
Odczyt wartości w pętli: 7	Odczyt wartości w pętli: 3
Koniec programu.	Odczyt wartości w pętli: 4
	Odczyt wartości w pętli: 5
	Odczyt wartości w pętli: 6
	Odczyt wartości w pętli: 7
	Koniec programu.

W przykładzie na listingu 1.51 przedstawiono iterator zwracający co drugi element. Dodatkowo pętla została zabezpieczona przed wykroczeniem poza zakres tablicy.

Listing 1.51. Iterator zwracający co drugi element kolekcji

```
public class Iterator
{
    public int[] kolekcja;
    public Iterator()
    {
        kolekcja = new int[] {1,2,3,4,5,6,7,8,9 };
    }
    public System.Collections.IEnumerable PrzykładowyIterator()
    {
        for (int index = 1; index <= kolekcja.Length-1; index += 2)
        {
            System.Console.WriteLine("Iterator zwraca element nr: "
                + (index+1));
            yield return kolekcja[index];
        }
    }
}
```

⁶² Program działa bez użycia niestandardowego iteratora.

```

public class Program
{
    public static void Main()
    {
        Iterator iterator = new Iterator();

        // Wyświetlanie kolekcji
        System.Console.WriteLine("Kolekcja wyświetlana co drugi element:");

        foreach (int i in iterator.PrzykładowyIterator())
        {
            System.Console.WriteLine("Pętla programu wyświetla
                                     ↪element:" + i);
        }
        Console.ReadLine();
    }
}

```

Wynik działania programu prezentuje listing 1.52.

Listing 1.52. Wynik działania aplikacji

```

Kolekcja wyświetlana co drugi element:
Iterator zwraca element nr: 2
Pętla programu wyświetla element: 2
Iterator zwraca element nr: 4
Pętla programu wyświetla element: 4
Iterator zwraca element nr: 6
Pętla programu wyświetla element: 6
Iterator zwraca element nr: 8
Pętla programu wyświetla element: 8

```

Iinicjalizatory obiektów i kolekcji

Iinicjalizatory kolekcji i obiektów to nowość wprowadzona w C# 3.0. Pozwalają na inicjalizację pól lub kolekcji już podczas deklaracji⁶³.

Przy użyciu inicjalizatorów obiektów można od razu przy tworzeniu obiektu nadać wartości jego polom składowym. Podobne rozwiązanie można było uzyskać przy użyciu konstruktora. W przykładzie z listingu 1.53 za pomocą inicjalizatora obiektów bez konstruktora udało się zainicjalizować właściwości.

Listing 1.53. Przykładowy kod wykorzystujący inicjalizatory obiektów

```

class Program
{
    static void Main(string[] args)
    {
        // Inicjalizator obiektów
        Auto auto = new Auto { Marka = "Audi", Pojemnosc = 1990 };
    }
}

```

⁶³ <http://msdn.microsoft.com/en-us/library/vstudio/bb384062.aspx>

```
// Standardowa inicjalizacja
Auto auto1 = new Auto();
auto1.Marka = "Audil";
auto1.Pojemnosc = 1991;
Console.WriteLine("Auto: {0}, pojemność: {1} \nAuto1: {2},
    ↪pojemność: {3}", auto.Marka, auto.Pojemnosc,
    ↪auto1.Marka, auto1.Pojemnosc);
Console.ReadLine();
}

class Auto
{
    public string Marka { get; set; }
    public int Pojemnosc { get; set; }
}
```

Inicjalizatory kolekcji, podobnie jak inicjalizatory obiektów, pozwalają na szybką inicjalizację kolekcji (listing 1.54).

Listing 1.54. Przykładowy kod wykorzystujący inicjalizatory kolekcji

```
class Program
{
    static void Main(string[] args)
    {
        // Inicjalizator kolekcji
        List<string> auta = new
            ↪List<string>{"Audi", "Mercedes", "BMW"};

        // Standardowa inicjalizacja
        List<string> autal = new List<string>();
        autal.Add("Audi");
        autal.Add("Mercedes");
        autal.Add("BMW");

        foreach(var el in auta)
        {
            Console.WriteLine("Marka: " + el);
        }
        Console.ReadLine();
    }
}
```

Drzewa wyrażeń

Drzewa wyrażeń (ang. *Expression Trees*) reprezentują kod w strukturze drzewiastej, w której każdy węzeł odpowiada pewnemu wyrażeniu. Przykładowo może to być zwykłe wyrażenie $x < y$ — w zależności od wyniku przechodzi się do prawej lub lewej gałęzi drzewa. Drzewa wyrażeń są wykorzystywane w zapytaniach LINQ do tłumaczenia kodu C# na kod SQL, a także w wyrażeniach lambda (listing 1.55)⁶⁴.

⁶⁴ <http://msdn.microsoft.com/pl-pl/library/bb397951.aspx>

Listing 1.55. Przykładowy kod korzystający z drzewa wyrażeń

```

class Program
{
    public static void Main()
    {
        // Drzewo wyrażeń
        Expression<Func<int, bool>> expr = num => num > 4;

        // Kompilacja drzewa
        Func<int, bool> result = expr.Compile();

        // Wywołanie delegata i wypisanie na ekranie
        Console.WriteLine(result(5));
        // Zwraca wartość true

        // Sposób uproszczony, komplikacja razem z wywołaniem
        Console.WriteLine(expr.Compile()(5));

        Console.ReadLine();
    }
}

```

Metody rozszerzające

Metody rozszerzające (ang. *Extensions Methods*) zostały wprowadzone w C# 3.0 i pozwalają rozszerzać funkcjonalność typów już istniejących. Metoda rozszerzająca może być zadeklarowana tylko w klasie statycznej i musi przed pierwszym parametrem

(typem, który się rozszerza) zawierać słowo kluczowe `this`, które mówi, że jest to metoda rozszerzająca. Dzięki metodom rozszerzającym, bez użycia jawnego dziedziczenia, można rozszerzać klasy — wcześniej było to niemożliwe.

W przykładzie z listingu 1.56 zwracana jest liczba znaków a w ciągu znaków typu `string`. Wykorzystane zostały metoda anonimowa (delegat bez nazwy), aby zwrócić wystąpienie znaku, metoda `FindAll<T>`, aby znaleźć wszystkie znaki, oraz właściwość `.Length`, aby zwrócić liczbę znaków⁶⁵.

Listing 1.56. Przykładowy kod korzystający z metody rozszerzającej

```

// Rozszerzenie klasy string
public static class StringExtension
{
    // Zwraca liczbę szukanych znaków w ciągu
    public static int ZwrocLiczbeLiter(this String str, char znak)
    {
        return Array.FindAll<char>(str.ToCharArray(), delegate(char c)
        {
            return c.Equals(znak);
        }).Length;
    }
}

```

⁶⁵ <http://msdn.microsoft.com/pl-pl/library/bb383977.aspx>

```
        }

    class Program
    {
        static void Main(string[] args)
        {
            string napis = "hahaha";

            int liczbaLiter = napis.ZwrocLiczbeLiter('a');
            Console.WriteLine("Liczba szukanych liter to: {0}",
                liczbaLiter);

            Console.ReadLine();
        }
    }
```

Tę samą metodę z użyciem wyrażenia lambda przedstawiono na listingu 1.57.

Listing 1.57. Metoda z przykładu z użyciem wyrażenia lambda

```
// Rozszerzenie klasy string
public static class StringExtension
{
    // Zwraca liczbę szukanych znaków w ciągu
    public static int ZwrocLiczbeLiter(this String str, char znak)
    {
        return Array.FindAll<char>(str.ToCharArray(),
            c => c.Equals(znak)).Length;
    }
}
```

Metody i klasy częściowe

Metody i klasy częściowe (ang. *Partial Methods*, *Partial Class*) to nowość wprowadzona w C# 3.0. Dzięki słowu kluczowemu `partial` możliwe jest podzielenie klasy, struktury, interfejsu lub metody na kilka części w osobnych plikach (listing 1.58). Wszystkie klasy częściowe muszą być dostępne do komplikacji, ponieważ zostają połączone w jedną klasę, i muszą się znajdować w obrębie jednego pliku `.dll` lub `.exe`.

Aby można było tworzyć częściową klasę lub metodę:

- ♦ każda część musi być poprzedzona słowem kluczowym `partial`;
- ♦ wszystkie części muszą mieć taką samą dostępność (`public`, `private` itp.);
- ♦ jeśli jedna część dziedziczy po innym typie, to wszystkie części po nim dziedziczą;
- ♦ jeśli jakakolwiek część jest zadeklarowana jako `sealed`, to wszystkie części są `sealed`;
- ♦ jeśli jakakolwiek część jest zadeklarowana jako `abstract`, to wszystkie części są `abstract`.

Listing 1.58. Przykładowy kod klasy częściowej

```

namespace Aplikacja1
{
    partial class Czesciowa
    {
        public void Metoda1(){}
        public void Metoda2(){}
    }

    class Program
    {
        static void Main(string[] args)
        {
            Czesciowa obiekt = new Czesciowa();
            obiekt.Metoda3();
            obiekt.Metoda1();
        }
    }

    partial class Czesciowa
    {
        public void Metoda3() { }
        public void Metoda4() { }
    }
}

```

Metody częściowe

W przypadku metod częściowych w jednym pliku można zamieścić deklarację metody, a w innym pliku ciało metody. Podczas komplikacji deklaracje metod, które nie mają ciała w żadnym pliku, zostają usunięte. Dzięki temu kod jest optymalny⁶⁶.

Metody częściowe:

- ◆ muszą zwracać void i zaczynać się od słowa kluczowego `partial`;
- ◆ mogą przyjmować parametry przez referencje (`ref`), ale nie mogą przyjmować parametrów przez wyjście (`out`);
- ◆ są niejawnie prywatne (`private`) i dlatego nie mogą być wirtualne (`virtual`);
- ◆ nie mogą być zewnętrzne (`extern`);
- ◆ mogą mieć modyfikatory `static` oraz `unsafe`;
- ◆ mogą być generyczne.

A oto deklaracja i ciało metody częściowej:

```

// Plik nr 1
partial int Metoda();

// Plik nr 2

```

⁶⁶ <http://msdn.microsoft.com/pl-pl/library/wa80x488.aspx>

```
partial int Metoda()
{
    // Ciało metody
}
```

Zmienne domniemane

Zmienne domniemane (słowo kluczowe `var`) zostały wprowadzone w C# 3.0. Zmienne domniemane to zmienne, dla których nie trzeba podawać typu. Zamiast typu można użyć słowa kluczowego `var`. Kompilator sam rozpozna typ zmiennej na podstawie danych, którymi zmienna jest zainicjalizowana. Inicjalizacja podczas deklaracji jest wymagana — bez niej nie byłoby możliwości rozpoznania typu danych, który musi być znany w momencie komplikacji programu. Typ zmiennej nie może zostać zmodyfikowany w trakcie działania programu (listing 1.59)⁶⁷.

Listing 1.59. Przykładowy kod wykorzystujący zmienne domniemane

```
// Standardowa deklaracja zmiennych
int liczba = 3;
string napis = "Napis";

// Deklaracja z użyciem zmiennych domniemanych
var liczba = 3;
var napis = "Napis";
```

Zmienne domniemane są często wykorzystywane w zapytaniach LINQ, ponieważ łatwiej napisać słówko `var` niż np. `IEnumerable<Nazwa_klasy>`. Nie zawsze znany jest typ danych zwracanych z zapytania lub podczas pisania zapytania typ się zmienia. Słowo `var` jest wtedy wygodniejsze. W przypadku zmiennych takich jak np. `int` nie powinno się używać `var`, ponieważ jest to zła praktyka. Nie mówi to nic na temat typu danych, przez co niweluje część korzyści, dla których zostały wprowadzone języki ścisłe typowane.

Typy anonimowe

Typy anonimowe zostały wprowadzone w C# 3.0. Typy anonimowe pozwalają tworzyć obiekty klas, których deklaracja nie istnieje (listing 1.60). Oszczędza to czas programisty, który musiałby on poświęcić na deklarację klasy⁶⁸.

Listing 1.60. Przykładowy kod wykorzystujący typy anonimowe

```
var obiekt = new { napis = "napis", liczba = 12, liczba1 = 13 };

// Zastępuje następujący kod

class Anonimowa
{
    private string _napis = "napis";
```

⁶⁷ <http://msdn.microsoft.com/pl-pl/library/bb384061.aspx>

⁶⁸ <http://msdn.microsoft.com/pl-pl/library/bb397696.aspx>

```

private int _liczba = 12;
private int _liczba1 = 13;

public string napis {get { return _napis; } set { _napis =
    ↪value; }}
public int liczba {get { return _liczba; } set { _liczba =
    ↪value; }}
public int liczba1 {get { return _liczba1; } set { _liczba1 =
    ↪value; }}
}

```

Słowa kluczowe this i base

Słowa kluczowe base i this służą do określenia, do którego składnika klasy realizowany jest dostęp. W przypadku dwóch klas, z których jedna jest klasą bazową dla drugiej klasy dziedziczącej po klasie bazowej, mogą się znajdować metody o tej samej nazwie lub pola o tej samej nazwie. Aby rozróżnić, do składnika której klasy się odnieść, zostały wprowadzone słowa kluczowe base i this. Słowo kluczowe this wskazuje na aktualną klasę, natomiast base na klasę bazową, o ile taka klasa istnieje (listing 1.61)⁶⁹.

Listing 1.61. Kod prezentujący działanie słów kluczowych base i this

```

public class Bazowa
{
    public string napis = "Napis z klasy bazowej";
}
public class Dziedziczaca : Bazowa
{
    public string napis = "Napis z klasy dziedziczącej";
    public void Wyswietl()
    {
        Console.WriteLine(this.napis);
        Console.WriteLine(base.napis);
    }
}
class Program
{
    static void Main(string[] args)
    {
        Dziedziczaca ob = new Dziedziczaca();
        ob.Wyswietl();

        // Program wyświetli:
        // Napis z klasy dziedziczącej
        // Napis z klasy bazowej
        Console.ReadLine();
    }
}

```

⁶⁹ <http://msdn.microsoft.com/en-us/library/dk1507sz.aspx>, <http://msdn.microsoft.com/en-us/library/hfw7t1ce.aspx>

Pozostałe zastosowania słowa kluczowego this:

- ♦ przy przekazywaniu parametrów w metodach,
- ♦ przy przekazywaniu aktualnej instancji do innej klasy,
- ♦ przy deklaracji indekserów,
- ♦ nie stosuje się dla klas statycznych, ponieważ nie posiadają instancji, do których można się odwoać.

Pozostałe zastosowania słowa kluczowego base:

- ♦ wywoływanie metod bazowych,
- ♦ do wywoływania konstruktora z klasy bazowej,
- ♦ nie stosuje się dla klas statycznych, ponieważ nie posiadają instancji, do których można się odwoać.

Typy dynamiczne

W C# 4.0 został wprowadzony nowy typ danych — dynamic. Dzięki wprowadzeniu typów dynamicznych w trakcie działania programu można zmieniać typ danych zapisanych w zmiennych. Statyczne typy są komplikowane poprzez środowisko uruchomieniowe CLR. Typy dynamiczne są omijane przez kompilator CLR, przechwytywane przez środowisko uruchomieniowe DLR i obsługiwane w czasie działania programu. DLR jest częścią CLR — powstało, aby umożliwić współpracę z dynamicznymi językami. Język C# jest ściśle typowany, wszystkie zmienne muszą mieć swój typ przed uruchomieniem programu, a więc w trakcie komplikacji. Języki dynamicznie typowane (np. PHP, Ruby, Python) nie wymagają przypisywania typów do zmiennych. Typ zmiennej wynika z wartości, jaką przechowuje, i jest sprawdzany w czasie działania programu. Kod języka dynamicznie typowanego nie wymaga od programisty deklarowania typu zmiennych, jednak jest bardziej skomplikowany w razie wystąpienia błędów.

Na listingu 1.62 zwrócono uwagę na słabe strony typowania dynamicznego. W przypadku typowania statycznego instrukcja `d--` została automatycznie podkreślona przez kompilator, natomiast w przypadku typowania dynamicznego kompilator ją opuszcza i jest ona sprawdzana dopiero w czasie działania programu.

Listing 1.62. Przykładowy kod z użyciem słowa kluczowego dynamic

```
public class Program
{
    public static void Main()
    {
        dynamic d = 13;
        Console.WriteLine("Aktualny typ zmiennej d to:
                           ↪{0}", d.GetType());
        // Typ zmiennej d to: System.Int32
        d--;
        Console.WriteLine("Odejmowanie zakończone powodzeniem.\n
                           ↪Wartość zmiennej d po odejmowaniu: {0}", d);
```

```

d = "Napis";
Console.WriteLine("Aktualny typ zmiennej d to: {0}",
    ➔d.GetType());
// Teraz typ zmiennej d to: Systems.String

Console.WriteLine("Wciśnij Enter, aby wywołać wyjątek
    ➔poprzez odejmowanie na zmiennej string");
Console.ReadLine();

d--; // Błąd — zmieniona jest aktualnie typu string
}
}

```

Słowa kluczowe var oraz object również pozwalają na przechowywanie zmiennych różnych typów. Typ var przypisuje typ do zmiennej na podstawie danych, którymi się ją zainicjuje, a więc zmieniona jest kompilowana i sprawdzana przez kompilator w czasie pisania programu. Nie ma możliwości późniejszej zmiany typu dla zmiennej:

```

var zmieniona = 10;
Console.WriteLine(zmieniona.GetType());
// Wyświetli System.Int32

zmieniona = "napis"; // Błąd — zmieniona jest typu int

```

Typ object jest typem, po którym dziedziczą wszystkie inne typy, i pozwala na przyznanie zmiennych dowolnych typów. Podczas działań na zmiennej typu object trzeba wykonać rzutowanie na konkretny typ:

```

object zmieniona = 10;
Console.WriteLine(zmieniona.GetType());
// Wyświetli System.Int32

zmieniona = (int)zmieniona + 1; // Rzutowanie

zmieniona = "napis"; // OK
Console.WriteLine(zmieniona.GetType());
// Wyświetli System.String

```

Typ object jest najbardziej zbliżony do typu dynamic, jednak występują pomiędzy nimi pewne różnice. W przypadku typu object nie można korzystać z właściwości zadeklarowanych dla zmiennych danego typu, ponieważ kompilator traktuje je jak zmienne typu object. W przypadku zmiennych dynamicznych typ podczas działania na zmiennej jest już znany, a więc można wykorzystać jego właściwości:

```

object zmieniona = "napis";
Console.WriteLine(zmieniona.Length);
// Błąd: typ object nie posiada właściwości Length

dynamic zmienialna = "napis";
Console.WriteLine(zmienialna.Length);
// Wyświetli długość zmiennej, czyli 5

```

Aby sprawdzić wydajność zmiennych dynamicznych, wykorzystano kod z listingu 1.63.

Listing 1.63. Zmienne dynamiczne a wydajność

```
public class Program
{
    private static void TestDynamicznych()
    {
        var stopwatch = Stopwatch.StartNew();

        dynamic a = 10;
        dynamic b = 20;
        dynamic c = 30;

        dynamic d = a + b + c;

        stopwatch.Stop();
        Console.WriteLine("Czas zmiennych dynamicznych:{0}",
                           stopwatch.ElapsedTicks);
    }

    private static void TestStatycznych()
    {
        var stopwatch = Stopwatch.StartNew();

        int a = 10;
        int b = 20;
        int c = 30;
        int d = a + b + c;

        stopwatch.Stop();
        Console.WriteLine("Czas zmiennych statycznych:{0}",
                           stopwatch.ElapsedTicks);
    }

    public static void Main()
    {
        TestDynamicznych();
        TestStatycznych();
        TestDynamicznych();
        TestStatycznych();
        TestDynamicznych();
        TestStatycznych();
        TestDynamicznych();
        TestStatycznych();
        Console.ReadLine();
    }
}
```

Wynik działania programu prezentuje listing 1.64. Pierwsze wywołanie metody ze zmiennymi dynamicznymi było aż 82 tys. razy wolniejsze niż ta sama metoda ze zmiennymi typowanymi statycznie. Kolejne wywołania były około 5 razy wolniejsze niż dla zmiennych statycznych. W przypadku zmiennych statycznych typy były znane jeszcze przed uruchomieniem programu, ponieważ była uruchomiona komplilacja programu. Dla zmiennych dynamicznych typy były wyznaczane podczas działania programu, dlatego trwało to dużo dłużej. Ponieważ kompilator DLR sprawdza na początku, czy metoda nie

została już wcześniej skompilowana, kolejne wywołania metody są dużo szybsze, jednak nie tak szybkie jak przy zmiennych typowanych statycznie.

Listing 1.64. Wynik działania programu

```
Czas zmiennych dynamicznych:82481
Czas zmiennych statycznych:1
Czas zmiennych dynamicznych:5
Czas zmiennych statycznych:1
Czas zmiennych dynamicznych:4
Czas zmiennych statycznych:1
Czas zmiennych dynamicznych:5
Czas zmiennych statycznych:1
```

Argumenty nazwane — Named Arguments

Argumenty nazwane (ang. *Named Arguments*) to nowość w C# 4.0. Wykorzystując argumenty nazwane, nie trzeba zachowywać kolejności parametrów przekazywanych do funkcji. Parametry są kojarzone po nazwie parametru, a nie — jak to jest w przypadku zwykłych parametrów — po kolejności.

Metody z argumentami nazwanymi zadziałyły prawidłowo pomimo różnej kolejności argumentów (listing 1.65). Ostatnia metoda ze zwykłym przekazaniem parametrów zmieniła kolejność liczb i zamiast wyniku +1 otrzymano wynik -1.

Listing 1.65. Przykładowy kod wykorzystujący parametry nazwane

```
class Program
{
    static void Main(string[] args)
    {
        // Przekazanie parametru przez argumenty nazwane
        Console.WriteLine(Odejmij(arg1: 3, arg2: 2)); // Zwraca +1
        Console.WriteLine(Odejmij(arg2: 2, arg1: 3)); // Zwraca -1

        // Zwykłe przekazanie parametru
        Console.WriteLine(Odejmij(3, 2)); // Zwraca +1
        Console.WriteLine(Odejmij(2, 3)); // Zwraca -1

        Console.ReadLine();
    }

    static int Odejmij(int arg1, int arg2)
    {
        return arg1 - arg2;
    }
}
```

Parametry opcjonalne

Parametry opcjonalne zostały wprowadzone w C# 4.0. Pozwalają na określenie parametrów, których nie trzeba, ale które można przekazać do funkcji. Każde wywołanie funkcji musi dostarczyć argumenty dla wszystkich wymaganych parametrów. Parametry opcjonalne muszą mieć ustawioną wartość domyślną podczas definicji. Parametry opcjonalne są definiowane na końcu listy argumentów. Deklaracja metody z parametrami opcjonalnymi wygląda następująco:

```
static int Odejmij(int arg1, int arg2=10, int arg3 =12)
{
    // Ciało metody
}
```

W przypadku kilku parametrów opcjonalnych, jeśli chcesz przekazać tylko jeden, znajdujący się na końcu listy parametrów, musisz przekazać wszystkie wcześniejsze parametry opcjonalne. Nie możesz jednak przekazywać pustych parametrów:

```
int liczba = Odejmij(12, .12); // Źle
int liczba1 = Odejmij(12);      // Dobrze
int liczba2 = Odejmij(12,12,12); // Dobrze
```

Aby zapobiec pustym parametrom, można skorzystać z parametrów nazwanych. Przekazuje się wówczas tylko interesujące parametry i jest to całkowicie prawidłowe⁷⁰. Oto przykład z użyciem argumentów nazwanych:

```
int liczba2 = Odejmij(12, arg3: 12); // Dobrze
```

Obsługa kontra- i kowariancji oraz słowa kluczowe in i out

Pojęcia kontrawariancji i kowariancji w C# 4.0 zostały wprowadzone dla typów generycznych⁷¹. Aby wyjaśnić pojęcia, zostaną użyte następujące klasy:

```
class Pojazd{}
class Auto : Pojazd{}
class Audi : Auto{}
```

Kowarianca to konwersja z bardziej precyzyjnego typu na bardziej ogólny (typem dla zwracanego obiektu może być zarówno typ tego obiektu, jak i każdy obiekt bazowy). W C# wartość zwracana z funkcji jest kowariancją.

Kowarianca pozwala do obiektu z klasy Pojazd (listing 1.66) przypisać obiekt typu Auto. Nie można jednak przypisać obiektu typu Audi, ponieważ nie zawsze obiekt zwracany musi być typu Audi.

⁷⁰ <http://msdn.microsoft.com/pl-pl/library/dd264739.aspx>

⁷¹ <http://msdn.microsoft.com/pl-pl/library/ee207183.aspx>

Listing 1.66. Kod prezentujący kowariancję

```
class Pojazd { }

class Auto : Pojazd
{
    static Auto GetAuto()
    {
        return new Auto();
    }
}

Pojazd pojazd = GetAuto();
Auto auto = GetAuto();
}

class Audi : Auto { }
```

Kontrawariancja to przeciwnieństwo kowariancji, czyli konwersja z typu bardziej ogólnego na typ bardziej precyzyjny (obiekt może być typem takim jak argument lub każdym, który po nim dziedziczy). W C# kontrawariancją są parametry funkcji.

Kontrawariancja pozwala na odwołanie się do bardziej precyzyjnego typu, czyli do parametru typu Auto (listing 1.67) można odwołać się typem dziedziczącym po nim, a więc Audi. Nie można natomiast odwołać się do typu nadzawanego, czyli do typu Pojazd, ponieważ nie każdy pojazd to samochód.

Listing 1.67. Kod prezentujący kontrawariancję

```
public class Pojazd { }

public class Auto : Pojazd
{
    public Auto Metoda(Auto auto)
    {
        return new Auto();
    }
}

void Program()
{
    Metoda(new Auto());
    Metoda(new Audi());
}

public class Audi : Auto { }
```

W C# 3.0 typy generyczne nie używały kowariancji ani kontrawariancji (listingi 1.68 i 1.69), a więc mogły zwracać i przyjmować parametry tylko tego samego typu, dla jakiego zostały utworzone.

Listing 1.68. Przykład dla C# 3.0

```
private IEnumerable<Auto> Metoda()
{
    return null;
}

IEnumerable<Pojazd> listaPojazd = Metoda(); // Błąd —
                                                → typy generyczne nie są kowariancją w C# 3.0
IEnumerable<Auto> listaAuto = Metoda(); // Dobrze
IEnumerable<Audi> listaAudi = Metoda(); // Błąd —
                                                → typy generyczne nie są kontrawariancją w C# 3.0
```

Listing 1.69. Przykład dla C# 4.0

```
private IEnumerable<Auto> Metoda()
{
    return null;
}

IEnumerable<Pojazd> listaPojazd = Metoda(); // Dobrze
IEnumerable<Auto> listaAuto = Metoda(); // Dobrze
```

Aby zapewnić obsługę kowariancji w C# 4.0, należy użyć słowa kluczowego `out` w deklaracji interfejsu:

```
public interface IEnumerable<out T>
```

Mögliwe staje się wówczas przypisanie:

```
IEnumerable<Pojazd> listaPojazd = new Auto();
```

Natomiast aby zdefiniować kontrawariancję, należy użyć słowa kluczowego `in` w deklaracji interfejsu:

```
public interface IJakisInterfejs<in T>
```

Mögliwe staje się wówczas przypisanie:

```
public IJakisInterfejs<Auto> Metoda()
{
    return null;
}

IJakisInterfejs<Auto> listaAut = Metoda();
IJakisInterfejs<Audi> listaAudi = Metoda();
```

W C# 3.0 była możliwość korzystania z kontra- i kowariancji w zwykłych metodach oraz delegatach, jednak nie było możliwości korzystania z kowariancji oraz kontrawariancji przy użyciu typów generycznych.

Przy stosowaniu kowariancji nie należy mylić słowa kluczowego `out` z przekazywaniem parametru poprzez wyjście.

Słowa kluczowe is, as i typeof

Operator `is` sprawdza, czy zmienna jest zgodna z danym typem. Jeśli zmienna może zostać rzutowana na wybrany typ bez rzucenia wyjątku, to wyrażenie zwraca wartość `true`⁷². Operator `is` działa w przypadku opakowywania, rozpakowywania i referencji:

```
KlasaAuto a;

if (auto is KlasaAuto)
{
    a = (KlasaAuto)auto;
    // Dalsza część kodu
}
```

Operator `as` służy do konwersji bądź rzutowania typów. Jeśli zmiana typu zakończy się niepowodzeniem, to zwracana jest wartość `null` (nie zostaje rzucony wyjątek `InvalidOperationException`, jak w przypadku zwykłego rzutowania, czyli jawnej konwersji typów)⁷³.

Operatora `as` można używać tylko dla typów `Nullable`, a więc takich, które mogą przyjmować wartość `null`:

```
class KlasaBazowa {}
class KlasaDziedzicząca : KlasaBazowa
{ }

KlasaDziedzicząca d = new KlasaDziedzicząca();
KlasaBazowa b = d as KlasaBazowa; // Operator as

f(b != null) // Sprawdzenie, czy konwersja się powiodła
{
    // Operacje na zmiennej b
}
```

Operator `as` można przetłumaczyć na wyrażenie składające się z operatorów `? :` i `is`⁷⁴.

```
wyrażenie is typ ? (typ)wyrażenie : (typ)null
```

Operator `typeof` pozwala określić, jakiego typu jest dana zmienna:

```
Type t = typeof(PrzykładowaKlasa);
// Lub
int liczba = 0;
System.Type type = liczba.GetType();
```

Leniwa inicjalizacja — Lazy Initialization

Leniwa inicjalizacja (ang. *Lazy Initialization*) to nowość wprowadzona w C# 4.0, głównie na potrzeby LINQ i Entity Framework. Leniwa inicjalizacja pozwala na wczytanie tylko części pól klasy. Przykładowo jeśli klasa przechowuje informacje o użytkowniku, takie

⁷² <http://msdn.microsoft.com/pl-pl/library/scekt9xw.aspx>

⁷³ <http://msdn.microsoft.com/pl-pl/library/cscesdfbt.aspx>

⁷⁴ <http://msdn.microsoft.com/pl-pl/library/58918ffs.aspx>

jak imię i nazwisko, oraz posiada listę dodatkowych informacji, np. płatności, to przy użyciu leniwej inicjalizacji można wczytać imię i nazwisko bez pobierania listy płatności. Wpływa to bardzo znacząco na wydajność, gdyż dane nie są przechowywane w pamięci, lecz pochodzą z zewnętrznego źródła danych (np. bazy danych lub pliku XML). Dane z płatnościami będą pobierane tylko wtedy, gdy potrzebna jest np. data zamówienia i odwołanie do nich następuje w kodzie. Aby oznaczyć kolekcje jako *Lazy*, należy użyć klasy `Lazy<T>`. `Lazy<T>` może być wykorzystane w środowisku wielowątkowym i jest *Thread Safe*. Aby utworzyć obiekt niewspółdzielony pomiędzy wątkami, należy użyć klasy `ThreadLocal<T>` (listing 1.70).

Listing 1.70. Przykład z użyciem `Lazy<T>`

```
class Program
{
    static void Main()
    {
        Uzytkownik user = new Uzytkownik();
        Console.WriteLine(
            →"\r\nImie: " + user.Imie +
            →"\r\nNazwisko: " +
            →"\r\nW tym momencie płatności nie zostały jeszcze
            →pobrane" + "\r\nKliknij Enter, aby pobrać płatności.");
        Console.ReadLine();

        var lista = user.Platnosci.Value;
        Console.WriteLine("\r\nPłatności zostały pobrane:\r\n");
        foreach(var el in lista)
        {
            Console.WriteLine(
                →"\r\nNumer zamówienia: " +
                →el.NumerZamowienia +
                →"\r\nWartość zamówienia: " +
                →el.WartoscZamowienia +
                →"\r\nData zamówienia: " +
                →el.DataZamowienia);
        }
        Console.WriteLine("\r\nKliknij Enter, aby zakończyć
            →działanie programu");
        Console.ReadLine();
    }
}

public class Uzytkownik
{
    public string Imie { get { return "Marek"; } set{} }
    public string Nazwisko { get { return "Marecki"; } set{} }
    public Lazy<IList<Platnosc>> Platnosci
    {
        get
        {
            return new Lazy<IList<Platnosc>>(() 
                →=> this.PobierzPlatnosci());
        }
    }
    private IList<Platnosc> PobierzPlatnosci()
    {
```

```

List<Platnosc> platnosci = new List<Platnosc>();
Platnosc platnosc1 = new Platnosc { DataZamowienia =
    ↪DateTime.Now, NumerZamowienia = 1,
    ↪WartoscZamowienia = 100 };
Platnosc platnosc2 = new Platnosc { DataZamowienia =
    ↪DateTime.Now.AddDays(1), NumerZamowienia = 2,
    ↪WartoscZamowienia = 300 };
platnosci.Add(platnosc1);
platnosci.Add(platnosc2);
// Lub kod odpowiedzialny za pobranie płatności z bazy danych
return platnosci;
}
}
public class Platnosc
{
    public int NumerZamowienia { get; set; }
    public DateTime DataZamowienia { get; set; }
    public int WartoscZamowienia { get; set; }
}

```

Obok `Lazy<T>` i `ThreadLocal<T>`, które tworzą dodatkową klasę dla każdego pola `Lazy` (przez co zajmują dodatkową pamięć), istnieje jeszcze trzeci sposób korzystania z leniwej inicjalizacji za pomocą statycznych metod z klasy `System.Threading.LazyInitializer`. *Lazy Initializer* działa na innej zasadzie, jednak wynik działania jest taki sam. Nie tworzy dodatkowego obiektu `Lazy`, tylko wykorzystuje mechanizm refleksji i pobiera parametr przez referencje, przez co nie zajmuje dodatkowej pamięci operacyjnej (listing 1.71). Ponieważ mechanizm refleksji jest kosztowny (szczególnie dla procesora), w niektórych przypadkach może działać wolniej niż `Lazy<T>` i `ThreadLocal<T>`, jednak nie wymaga zmiany deklaracji klasy.

Listing 1.71. Przykład z użyciem *LazyInitializer*

```

class Program
{
    static void Main()
    {
        Uzytkownik user = new Uzytkownik();

        Console.WriteLine(
            ↪"\r\nImie: " + user.Imie +
            ↪"\r\nImie: " + user.Nazwisko +
            ↪"\r\nW tym momencie płatności nie zostały jeszcze pobrane" +
            ↪"\r\nKliknij Enter, aby pobrać płatności.");
        Console.ReadLine();

        // user.Platnosci ma wartość null
        user.InitializePlatnosci();
        // user.Platnosci posiada listę płatności

        Console.WriteLine("\r\nPłatności zostały pobrane:\r\n");
        foreach (var el in user.Platnosci)
        {
            Console.WriteLine(
                ↪"\r\nNumer zamówienia: " + el.NumerZamowienia +

```

```
    ↪ " \r\n Wartość zamówienia: " + el.WartoscZamowienia +
    ↪ " \r\n Data zamówienia: " + el.DataZamowienia);
}

Console.WriteLine(" \r\n Kliknij Enter, aby zakończyć działanie
    ↪ programu");
Console.ReadLine();

}

public static Platnosc PobierzPlatnosc()
{
    Platnosc platnosc = new Platnosc { DataZamowienia =
        ↪ DateTime.Now.AddDays(1), NumerZamowienia = 1,
        ↪ WartoscZamowienia = 1 * 100 };
    return platnosc;
}

}

public class Uzytkownik
{
    public string Imie { get { return "Marek"; } set { } }
    public string Nazwisko { get { return "Marecki"; } set { } }
    public List<Platnosc> Platnosc { get; set; }

    public void InitializePlatnosc()
    {
        List<Platnosc> platnosc = new List<Platnosc>();
        for (int i = 1; i < 4; i++)
        {
            Platnosc platnosc = null;
            LazyInitializer.EnsureInitialized<Platnosc>(
                ↪ ref platnosc, () =>
            {
                return PobierzPlatnosc(i);
            });
            platnosc.Add(platnosc);
        }
        Platnosc = platnosc;
    }
}

private Platnosc PobierzPlatnosc(int i)
{
    Platnosc platnosc = new Platnosc { DataZamowienia =
        ↪ DateTime.Now.AddDays(i), NumerZamowienia = i,
        ↪ WartoscZamowienia = i * 100 };
    return platnosc;
}

}

public class Platnosc
{
    public int NumerZamowienia { get; set; }
    public DateTime DataZamowienia { get; set; }
    public int WartoscZamowienia { get; set; }
}
```

Metody asynchroniczne — `async` i `await`

Słowa kluczowe `async` i `await` to nowość wprowadzona w C# 5.0. Użycie tych słów kluczowych pozwala na bardzo prostą implementację metod asynchronicznych. Przed wprowadzeniem słów kluczowych `async` i `await` asynchroniczne wywołanie metod było dużo bardziej skomplikowane i wymagało implementacji metod zwrotnych (ang. *Callback*), czyli metod wywoływanych po zakończeniu określonego zadania. W C# 5.0 odpowiedzialne jest za to słowo kluczowe `await`, które czeka na wynik działania metody. Metoda, która zawiera w sobie słowo kluczowe `await`, musi zostać poprzedzona słowem kluczowym `async`, które informuje o asynchronicznym wywołaniu metody. Asynchroniczne wywołanie metod nie oznacza, że będą wykonywane na kilku wątkach. Wręcz przeciwnie — metody asynchroniczne nie blokują wątku i są wywoływane na jednym wątku tak długo, jak to tylko możliwe⁷⁵.

Metoda asynchroniczna może zwracać następujące typy: `Task` (reprezentuje asynchroniczną akcję), `void` (nie zwraca nic) lub najczęściej używany typ `Task<T>`, czyli generyczny typ `Task<twoj_typ_danych>`, w którym podaje się typ danych. Metoda asynchroniczna musi zawierać przedrostek `async` oraz słowo `await` przed wywołaniem metod, które będą blokować wątek poprzez powolne wykonywanie, np. pobieranie danych z zewnętrznych źródeł (listing 1.72).

Listing 1.72. Przykładowa metoda asynchroniczna służąca do logowania

```
public async Task<ActionResult> Login(LoginViewModel model,
                                         ↳string returnUrl)
{
    if (ModelState.IsValid)
    {
        var user = await UserManager.FindAsync(model.UserName,
                                                ↳model.Password);
        if (user != null)
        {
            await SignInAsync(user, model.RememberMe);
            return RedirectToAction(returnUrl);
        }
        else
        {
            ModelState.AddModelError("", "Zła nazwa użytkownika lub
                                         ↳hasło");
        }
    }
    return View(model);
}
```

⁷⁵ <http://msdn.microsoft.com/pl-pl/library/programowanie-asynchroniczne-w-net-4-5.aspx>

Atrybuty Caller Info

Atrybuty *Caller Info* to nowość wprowadzona w C# 5.0. Za pomocą tych atrybutów można uzyskać informacje o wywołaniu metody, ścieżkę do pliku kodu źródłowego oraz numer wiersza. Atrybuty te przydadzą się do śledzenia, debugowania i diagnozy poprawności kodu (listing 1.73)⁷⁶.

Atrybuty zdefiniowane w System.Runtime.CompilerServices zostały omówione w tabeli 1.17.

Tabela 1.17. Atrybuty z System.Runtime.CompilerServices

Atrybut	Opis	Typ
CallerFilePathAttribute	Ścieżka pliku źródłowego, która zawiera obiekt wywołujący; jest to ścieżka do pliku w czasie komplikacji	string
CallerLineNumberAttribute	Numer wiersza w pliku źródłowym, w którym metoda jest wywoływana	int
CallerMemberNameAttribute	Nazwa metody lub właściwości obiektu wywołującego	string

Listing 1.73. Kod przykładowego programu

```
class Program
{
    static void Main(string[] args)
    {
        Program p1 = new Program();
        p1.Wykonaj();
    }

    public void Wykonaj()
    {
        WyswietlDane("Treść wiadomości");
        Console.ReadLine();
    }

    public void WyswietlDane(string message,
        [CallerMemberName] string memberName = "",
        [CallerFilePath] string sourceFilePath = "",
        [CallerLineNumber] int sourceLineNumber = 0)
    {
        Console.WriteLine("Wiadomość: " + message);
        Console.WriteLine("Nazwa metody: " + memberName);
        Console.WriteLine("Plik źródłowy: " + sourceFilePath);
        Console.WriteLine("Numer linii w pliku: " +
            sourceLineNumber);
    }
}
```

Wynik działania aplikacji prezentuje listing 1.74.

⁷⁶ <http://msdn.microsoft.com/pl-pl/library/hh534540.aspx>

Listing 1.74. Wynik działania programu

Wiadomość: Treść wiadomości
Nazwa metody: Wykonaj
Plik źródłowy: c:\Users\user\Documents\Visual Studio 2013\Projects\ConsoleApp1\bin\Debug\ConsoleApplication1\Program.cs
Numer linii w pliku: 22

Nowości w C# 6.0

Prawie wszystkie nowości zaimplementowane w C# 6.0 dotyczą jedynie uproszczenia składni, nie wnoszą jednak nowych funkcjonalności, tak jak miało to miejsce we wcześniejszych wersjach języka.

Konstruktory pierwotne — Primary Constructors

Skrócona wersja konstruktora dla automatycznej inicjalizacji prywatnych składowych. Zamiast konstruktora podaje się listę parametrów przy deklaracji klasy. Działa identycznie jak wersja z konstruktorem.

Przed C# 6.0:

```
public class Punkt
{
    private int x, y;

    public Punkt(int x, int y)
    {
        this.x = x;
        this.y = y;
    }
}
```

W C# 6.0:

```
public class Punkt(int x, int y) {
    private int x = x;
    private int y = y;
}
```

Automatyczna inicjalizacja właściwości — Initializers for Auto-properties

Wcześniej zmienne można było inicjalizować poprzez konstruktor lub dodanie dodatkowego pola i późniejsze przypisanie wartości dla właściwości. Od wersji C# 6.0 możliwa jest dużo prostsza inicjalizacja właściwości.

Przed C# 6.0:

```
private readonly int x;  
public int X  
{  
    get { return x; }  
    set { x = value; }  
}
```

W C# 6.0:

```
public int X { get; set; } = x;
```

Możliwe jest również zainicjowanie wartością początkową właściwości tylko do odczytu:

```
public int X { get; } = x;
```

Dyrektywa using dla składowych statycznych — Using Static Members

W C# 6.0 możliwe staje się użycie dyrektywy `using` dla składowych statycznych. Nie ma już potrzeby podawania całkowitej ścieżki przy każdym odwołaniu.

Przed C# 6.0:

```
public double A { get { return Math.Sqrt(Math.Round(2.422)); } }
```

W C# 6.0:

```
using System.Math;  
...  
public double A { get { return Sqrt(Round(2.422)); } }
```

Inicjalizatory słownikowe — Dictionary Initializer

Inicjalizacja kolekcji typu słownikowego staje się bardziej czytelna. Nowy format zapisu przyjmuje wartość klucza w nawiasie [], natomiast wartość dla danego klucza jest umieszczana po znaku =.

Przed C# 6.0:

```
var slownik = new Dictionary<string, int>  
{  
    { "pierwszyKlucz", 4 },  
    { "drugiKlucz", 9 }  
};
```

W C# 6.0:

```
var slownik = new Dictionary<string, int>  
{  
    ["pierwszyKlucz"] = 4,  
    ["drugiKlucz"] = 9  
};
```

Deklaracje inline dla parametrów out — Inline Declarations for Out Params

Umożliwia zdefiniowanie nowej zmiennej wyjściowej w wywołaniu metody. Nie jest to duża redukcja liczby wyrażeń koniecznych do napisania, jednak w przypadku parametrów wyjściowych taki zapis ma sens.

Przed C# 6.0:

```
int x;  
int.TryParse("456", out x);
```

W C# 6.0:

```
int.TryParse("456", out int x);
```

Wyrażenia dla właściwości — Property Expressions

Wygląda jak wyrażenie lambda, ale nie ma z nim nic wspólnego. Upraszczają składnię, skracają zapis i sprawia, że kod dla właściwości jest bardziej czytelny.

Przed C# 6.0:

```
public double Dystans {  
    get { return Math.Sqrt((X * X) + (Y * Y)); }  
}
```

W C# 6.0:

```
public double Dystans => Math.Sqrt((X * X) + (Y * Y));
```

Wyrażenia dla metod — Method Expressions

Podobnie jak w przypadku wyrażeń dla właściwości, nie ma nic wspólnego z wyrażeniami lambda. Działa identycznie jak przy właściwościach, upraszcza składnię, skracając zapis i sprawia, że kod jest bardziej czytelny.

Przed C# 6.0:

```
public Punkt Przesun(int dx, int dy)  
{  
    return new Punkt(X + dx, Y + dy);  
}
```

W C# 6.0:

```
public Punkt Przesun(int dx, int dy) => new Punkt(X + dx, Y + dy);
```

Modyfikator **private protected**

W C# 6.0 możliwe staje się zastosowanie modyfikatorów `private` i `protected` jednocześnie. Oznacza to, że składowa oznaczona tymi modyfikatorami jest typu `protected` w obrębie podzespołu (*assembly*), a typu `private` poza podzespołem. Podzespołem może być biblioteka `.dll` lub program `.exe`:

```
private protected string PobierzDane() { ... }
```

Kolekcje `IEnumerable` jako parametr

— Params for `Enumerables`

Od wersji C# 6.0 można podać całą kolekcję `IEnumerable` jako parametr do metody przy użyciu słowa kluczowego `params`. Przed wersją 6.0 konieczna była zamiana na tablice i podanie tablicy jako parametru.

Przed C# 6.0:

```
ZrobCos(kolekcjaEnumerable.ToArray());
public void ZrobCos(params int[] wartosci) { ... }
```

W C# 6.0:

```
ZrobCos(kolekcjaEnumerable);
public void ZrobCos(params IEnumerable<Kurs> kursy) { ... }
```

Jednoargumentowe sprawdzanie wartości `null`

— Monadic Null Checking

Operator `?.` oznacza „Sprawdź to, co po lewej”. Jeśli obiekt nie ma wartości `null`, to wykonaj to, co po prawej; jeśli ma wartość `null`, zwróć `null` i przerwij proces.

Przed C# 6.0:⁷⁷

```
if (liczby != null) {
    var nastepna = liczby.FirstOrDefault();
    if (nastepna != null && nastepna.X != null) return nastepna.X;
}
return -1;
```

W C# 6.0:

```
var nastepna = liczby?.FirstOrDefault()?.X ?? -1;
```

⁷⁷ W przykładzie zamiast `null` zwracana jest wartość `-1`.

Słowo kluczowe await w blokach catch i finally

Od wersji C# 6.0 możliwe jest korzystanie ze słowa `await` w blokach `catch` i `finally`. Implementacja tego udoskonalenia jest bardzo skomplikowana, jednak użycie jest bardzo proste. W poprzednich wersjach języka C# korzystanie ze słowa `await` było możliwe tylko i wyłącznie w bloku `try`:

```
Zasob zas = null;
try
{
    zas = await Zasob.OpenAsync(...); // Możliwe przed C# 6.0
}
catch(ResourceException e)
{
    await Zasob.LogErrorAsync(zas, e); // Tylko w C# 6.0
}
finally
{
    if (zas != null)
        await zas.CloseAsync(); // Tylko w C# 6.0
}
```

Filtry wyjątków — Exception Filters

Filtry wyjątków były już wcześniej dostępne w językach VB i F#, a teraz zostaną wprowadzone do języka C# 6.0. Sprawdzany jest jeden dodatkowy warunek znajdujący się w instrukcji `if`, zanim zostanie rzucony wyjątek:

```
catch(Win32Exception exception)
if (exception.NativeErrorCode == 0x00042)
{
    ZrobCos();
}
```

Literały binarne i separatory cyfr — Binary Literals, Digit Separators

Separatory cyfr umożliwiają dzielenie zmiennych na logiczne części za pomocą znaku `_`, co poprawia czytelność zmiennych, a nie wpływa na wartość zmiennej:

```
int liczba = 1_234_567_890;
```

Literały binarne pozwalają na zapis wartości zmiennych w formie binarnej. Można ich używać łącznie z separatorami cyfr, co jest dużym ułatwieniem dla osób pracujących z przesunięciami bitów. Zapis wartości w formie binarnej musi się zaczynać od przedrostka `0b`:

```
int bits = 0b0010_1110;
int hex = 0x00_2E;
```

Rozdział 2.

Wzorce architektoniczne

Wzorce architektoniczne wskazują elementy, z jakich składa się aplikacja webowa, określają funkcjonalności realizowane przez każdy z nich oraz zasady ich komunikacji.

Architektura wielowarstwowa

Architektura wielowarstwowa (ang. *Multi-tier Architecture* lub *N-tier Architecture*) to architektura typu klient-serwer. Bzuje ona na podziale systemu na kilka warstw, które mogą być oddzielnie rozwijane i aktualizowane. Koncepcja warstwy aplikacji (ang. *tier*) pozwala pogrupować różne rozwiązania architektoniczne do osobnych warstw. Aktualizacja jednej warstwy nie wpływa na inną. Warstwy komunikują się między sobą za pomocą interfejsów, klas abstrakcyjnych lub poprzez bezpośrednie odwołania.

Do zalet architektury warstwowej należą:

- ◆ podział systemu na niezależne komponenty, umożliwiające ich osobne rozwijanie i aktualizowanie,
- ◆ możliwość podmiany komponentu z dowolnej warstwy na inny przy użyciu tego samego protokołu,
- ◆ specjalizacja warstw w realizacji konkretnych zadań,
- ◆ warstwy można wykorzystywać także w innych aplikacjach,
- ◆ możliwość scalania warstw przy zachowaniu protokołów do następnej i poprzedniej warstwy,
- ◆ możliwość rozdzielenia warstwy,
- ◆ możliwość równoważenia obciążzeń pomiędzy warstwami,
- ◆ rozwiązania redundantne (nadmiarowe, zapasowe).

Do wad architektury warstwowej należą:

- ◆ trudna do implementacji w istniejących systemach,
- ◆ potrzeba obudowania funkcjonalności w dodatkowy interfejs,
- ◆ utrudnienia związane z projektowaniem architektury,
- ◆ mniejsza wydajność poprzez konieczność stosowania protokołów komunikacji pomiędzy warstwami,
- ◆ skomplikowana budowa,
- ◆ konieczność zapewnienia zgodności wstępnej podczas rozbudowy.

Architektura jednowarstwowa

Przykładem aplikacji jednowarstwowych są proste aplikacje desktopowe, niekomunikujące się poprzez sieć, aby wykonać określone zadanie, np. pobranie danych z innego serwera tej samej aplikacji. Wszystkie potrzebne elementy aplikacji działają na jednym komputerze i są słabo skalowalne, czyli ich architektura utrudnia uruchomienie na kilku maszynach jednocześnie.

Architektura dwuwarstwowa

Architektura dwuwarstwowa składa się zazwyczaj z trzech elementów: części serwerowej, części klienckiej oraz protokołu komunikacji pomiędzy warstwami. Komunikacja jest realizowana w postaci klient-serwer. Aplikacja dwuwarstwowa pozwala oddzielić warstwę prezentacji od warstwy logiki lub warstwy dostępu do danych.

Architektura trójwarstwowa

Najpowszechniejszą architekturą warstwową jest architektura trójwarstwowa. Polega na podziale aplikacji na trzy główne warstwy: prezentacji, logiki biznesowej oraz warstwy dostępu do danych. Przykładem warstwy trójwarstwowej jest wzorzec MVC.

Architektura n-warstwowa

Aplikacja wielowarstwowa może się składać z dowolnej liczby warstw. Pozostałe warstwy to:

- ◆ warstwa autentykacji i autoryzacji,
- ◆ warstwa danych (warstwa zarządzania danymi),
- ◆ warstwa multipleksacji/demultipleksacji połączeń.

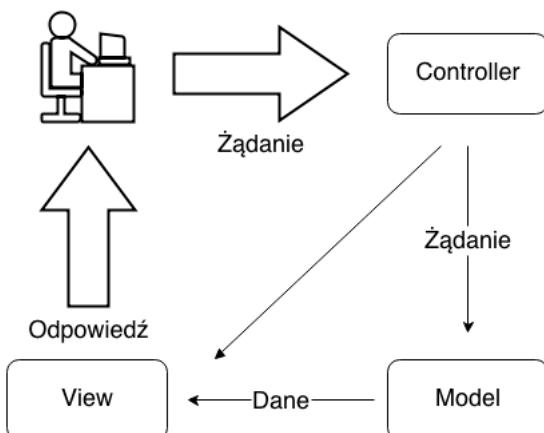
MVC

Wzorzec projektowy MVC (ang. *Model View Controller* — model, widok, kontroler) został zaprojektowany już w 1979 r. Zyskał on na popularności wraz z szybkim rozwojem inżynierii oprogramowania. Wzorzec MVC dzięki swej budowie bardzo dobrze pasuje do aplikacji internetowych. Jego głównym założeniem jest separacja kodu na trzy główne moduły:

- ◆ *Model* — reprezentuje dane (strukturę i powiązania) oraz logikę biznesową,
- ◆ *View* — reprezentuje wygląd i interfejs użytkownika,
- ◆ *Controller* — reprezentuje logikę sterującą aplikacją i obsługę zdarzeń.

Schemat wzorca MVC prezentuje rysunek 2.1.

Rysunek 2.1.
Wzorzec MVC



Do zalet MVC należą:

- ◆ podział na moduły porządkujące kod aplikacji,
- ◆ oddzielenie logiki biznesowej od widoku,
- ◆ brak zależności modelu od widoku,
- ◆ ułatwia odnalezienie konkretnej części kodu,
- ◆ łatwiejsza rozbudowa poprzez modułową budowę,
- ◆ zapobiega tworzeniu się bałaganu w kodzie,
- ◆ ułatwia pracę zespołową.

Do wad MVC należą:

- ◆ skomplikowany podział na moduły, zwiększający złożoność systemu (za wadę można to uznać tylko w przypadku bardzo małych rozwiązań),
- ◆ wymaga znajomości wzorca przez zespół,

- ◆ zależność widoku od modelu,
- ◆ utrudnione testowanie widoków.

Podczas korzystania z wzorca MVC jest bardzo prawdopodobne, że będą użyte również inne wzorce projektowe. Wzorce wykorzystywane w MVC:

- ◆ kompozyt (do widoku) — umożliwia łączenie kilku widoków częściowych (ang. *Partial View*) w jeden widok ogólny,
- ◆ obserwator (model, widok) — wykorzystywany w modelu aktywnym,
- ◆ strategia (widok, kontroler) — do wyboru implementacji kontrolera,
- ◆ metoda wytwórcza (kontrolery) — wykorzystywana do wyboru kontrolera,
- ◆ dekorator (widok) — rozszerzenie widoków o nowe funkcjonalności.

View

Widok to część kodu odpowiedzialna za sposób prezentacji użytkownikowi danych pobranych z modelu. Widok nie posiada żadnej logiki biznesowej — koncentruje się tylko na wyświetleniu danych, co ułatwia późniejsze zmiany w aplikacji. Widok może zmienić stan modelu tylko wówczas, gdy modyfikacja dotyczy zmiany sposobu wyświetlania danych. Aby zmienić wygląd aplikacji, nie trzeba ingerować w kod odpowiedzialny za logikę.

Controller

Kontroler obsługuje interakcje z użytkownikiem. Zawiera logikę obsługi zdarzeń przekazywanych z widoku. Obsługuje autoryzację oraz autentykację użytkowników i zezwala na dostęp do poszczególnych części aplikacji tylko uprawnionym użytkownikom. W zależności od poczynań użytkownika aktualizuje model oraz odświeża widok i może również przekazać sterowanie do innego kontrolera.

Model

Model jest tą częścią kodu, która jest odpowiedzialna za kontakt ze źródłami danych (bazy danych, pliki XML itp.). Jest to pośrednik pomiędzy źródłem danych a widokiem. Zajmuje się zapisem i odczytem danych, a także zapewnieniem ich spójności i validacją. W modelu znajduje się cała logika biznesowa aplikacji.

Domain Model, MVC Model i ViewModel — porównanie

Model w sensie części wzorca MVC to warstwa dostępu do danych zawierająca logikę biznesową. *Domain Model* to zbiór klas reprezentujących tabele w bazie danych lub pewną część aplikacji i jest to tylko część modelu ze wzorca MVC. *ViewModel* to model, który dostarcza dane do określonego widoku. Może łączyć dane z kilku modeli domenowych (ang. *Domain Model*) lub tabel z bazy danych w jeden model przekazywany do widoku.

Model pasywny a model aktywny

Model pasywny reprezentuje elementy, które nie mogą samoczynnie zmienić swojego stanu. Model pasywny nie dopuszcza do sytuacji, w której to model informuje kontroler o tym, że trzeba odświeżyć widok, ponieważ dane zostały zmienione. Model aktywny reprezentuje elementy, które mogą samoczynnie zmienić stan, bez ingerencji użytkownika. W przypadku modelu aktywnego wymagany jest mechanizm służący do poinformowania kontrolera o zmianie danych i konieczności odświeżenia widoku. Model aktywny jest realizowany za pomocą wzorca projektowego *Observer*, aby usunąć zależności pomiędzy warstwami.

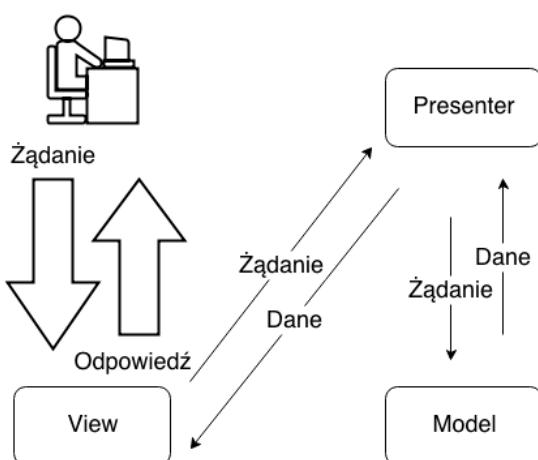
MVP

MVP (ang. *Model View Presenter*) to wzorzec powstały na bazie wzorca MVC. We wzorcu MVP prezenter jest tym samym, czym kontroler we wzorcu MVC, z jedną małą różnicą — w prezenterze jest zawarta logika biznesowa. Dane nie są przekazywane bezpośrednio z modelu do widoku, jak ma to miejsce w MVC. Prezenter wysyła zapytanie do modelu, a model zwraca dane do prezentera. Prezenter przetwarza otrzymane dane i przekazuje je do widoku. Głównym założeniem wzorca MVP jest separacja kodu na trzy główne moduły:

- ◆ *Model* — reprezentuje dane (strukturę i powiązania),
- ◆ *View* — reprezentuje wygląd i interfejs użytkownika,
- ◆ *Presenter* — reprezentuje logikę sterującą aplikacją i obsługę zdarzeń oraz logikę biznesową.

Schemat wzorca MVP prezentuje rysunek 2.2.

Rysunek 2.2.
Wzorzec MVP



Do zalet MVP należą:

- ◆ podział na moduły porządkujące kod aplikacji,
- ◆ oddzielenie logiki biznesowej od widoku,
- ◆ brak zależności modelu od widoku,
- ◆ ułatwia odnalezienie konkretnej części kodu,
- ◆ łatwiejsza rozbudowa poprzez modułową budowę,
- ◆ zapobiega tworzeniu się bałaganu w kodzie,
- ◆ ułatwia pracę zespołową.

Do wad MVP należą:

- ◆ konieczność zachowania struktury modułowej MVP,
- ◆ zbyt skomplikowany w przypadku prostych, małych aplikacji,
- ◆ wymaga zapoznania pracowników z wzorcem, co powoduje większe koszty.

Model

Model jest tą częścią kodu, która jest odpowiedzialna za kontakt z bazą danych. Jest to pośrednik pomiędzy prezenterem a bazą danych. Zajmuje się zapisem i odczytem danych, a także zapewnieniem ich spójności i walidacją.

View

Widok to część kodu odpowiedzialna za sposób prezentacji użytkownikowi danych przekazanych przez prezentera. Widok nie posiada żadnej logiki biznesowej i koncentruje się tylko na wyświetleniu danych, co ułatwia późniejsze zmiany w aplikacji. Aby zmienić wygląd aplikacji, nie trzeba ingerować w kod odpowiedzialny za logikę.

Presenter

Prezenter zawiera całą logikę biznesową aplikacji oraz obsługuje interakcje z użytkownikiem. W zależności od działań użytkownika odczytuje potrzebne dane poprzez model, wykonuje konieczne obliczenia i przekazuje rezultat swoich obliczeń do widoku. Prezenter obsługuje także autoryzację oraz autentykację użytkowników i zezwala na dostęp do poszczególnych części aplikacji tylko uprawnionym użytkownikom.

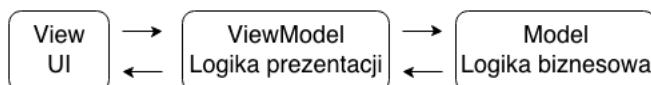
MVVM

MVVM (ang. *Model View ViewModel*) to wzorzec stosowany w aplikacjach WPF (ang. *Windows Presentation Foundation*) i Silverlight. Powstał na bazie wzorca MVC i pozwala w większym stopniu wykorzystać korzyści, jakie niesie ze sobą obsługa zdań, na których bazuje WPF. MVVM składa się z trzech części:

- ♦ *Model*,
- ♦ *View*,
- ♦ *ViewModel*.

Wzorca MVVM używa się, gdy interfejs użytkownika aplikacji jest bardzo skomplikowany, a poszczególne części mocno ze sobą związane. Widok to interfejs użytkownika, natomiast logika dostarczająca dane i sterująca interfejsem znajduje się w warstwie *ViewModel*. Model, podobnie jak we wzorcu MVC, jest odpowiedzialny za logikę biznesową oraz udostępnianie danych w sposób łatwo przetwarzalny dla WPF. Warstwa *ViewModel* nie zawiera żadnych kontrolek ani referencji do warstwy widoku, dostarcza tylko dane z modelu w sposób zrozumiały dla widoku. Widokiem w WPF i Silverlight jest plik XAML. Schemat wzorca MVVM prezentuje rysunek 2.3.

Rysunek 2.3.
Wzorzec MVVM



Do zalet MVVM należą:

- ♦ podział na *View* i *ViewModel*,
- ♦ brak duplikacji danych,
- ♦ łatwiejsza testowalność warstw aplikacji (testy jednostkowe),
- ♦ niezależność logiki od sposobu wyświetlania danych.

Do wad MVVM należą:

- ♦ komplikacja kodu — *ViewModel* nie może się komunikować z warstwą *View*, co w pewnych sytuacjach bardzo komplikuje powiązania,
- ♦ do obsługi jednego widoku należy tworzyć wiele klas, co sprawia, że projekt jest bardziej rozbudowany,
- ♦ duża liczba klas bazowych.

MVC, MVP i MVVM

W MVC kontroler obsługuje zdarzenia, manipuluje modelem, który zawiera logikę biznesową, a widok wyświetla dane z modelu. Jeden kontroler obsługuje kilka widoków.

W MVP dane z modelu są przekazywane do prezentera, a nie bezpośrednio do widoku. Prezenter przekazuje je do widoku. Jeden prezenter odnosi się do jednego widoku.

W MVVM dane z modelu są przekazywane do *ViewModel*, który może obsługiwać kilka widoków. Widok nie wie nic o *ViewModel* i wymaga tylko potrzebnych danych.

Użycie wzorców w aplikacjach:

- ◆ MVVM — WPF, Silverlight,
- ◆ MVP — ASP.NET Web Forms, Compact Framework,
- ◆ MVC — ASP.NET MVC¹.

DDD

DDD (ang. *Domain Driven Design*) to sposób tworzenia oprogramowania, który kładzie nacisk na definiowanie komponentów systemu w taki sposób, aby odpowiadały pewnej części tworzonego oprogramowania. Część ta nazywana jest domeną i może to być np. domena wystawiania faktur lub rozliczeń. Podejście DDD pozwala na modelowanie systemu informatycznego poprzez ludzi nieznających się na budowie systemów od strony programistycznej lub architektonicznej. DDD nie jest wzorcem ani metodyką. To podejście do tworzenia oprogramowania, sposób myślenia, ustalania priorytetów dla systemów ze złożoną warstwą domenową. Podejście DDD powinno się stosować tylko w sytuacjach, gdy model domenowy jest bardzo skomplikowany.

SOA

Architektura zorientowana na usługi SOA (ang. *Service-Oriented Architecture*) to architektura systemów informatycznych polegająca na definiowaniu usług spełniających wymagania użytkownika. Jako usługę określa się każdy element oprogramowania posiadający swój własny interfejs, realizujący pewne funkcje oraz działający niezależnie od innych elementów. Interfejs definiuje sposób działania usługi, ukrywając szczegóły implementacji. Istnieje wspólne, dostępne dla wszystkich usług medium komunikacyjne, umożliwiające swobodny przepływ danych pomiędzy elementami systemu. Przykładem realizacji SOA są Web serwisy (ang. *Web Services*), które wykorzystują standardowe interfejsy komunikacyjne (np. SOAP) do wymiany informacji pomiędzy systemami różnego typu, a więc realizują pełną usługę.

¹ <http://www.codeproject.com/Articles/66585/Comparison-of-Architecture-presentation-patterns-M>

EDA

Architektura zorientowana na zdarzenia EDA (ang. *Event-Driven Architecture*) to architektura systemów informatycznych polegająca na obsłudze, tworzeniu, reakcjach i detekcji zdarzeń biznesowych. Jako zdarzenie można rozumieć znaczącą zmianę stanu obiektu. System zorientowany na zdarzenia zazwyczaj posiada agentów, którzy emitują zdarzenia, i konsumentów, którzy odbierają informacje od agentów.

Konsumenci w zależności od typu zdarzenia mogą filtrować, zmieniać lub przekazywać zdarzenia do kolejnych komponentów systemu. Schemat działania EDA pozwala na pracę w środowiskach asynchronicznych.

Rozdział 3.

Microsoft .NET Framework

Microsoft .NET Framework to elastyczne środowisko do tworzenia nowoczesnego oprogramowania.

Struktura .NET

.NET Framework, w skrócie .NET, to platforma programistyczna, na którą składa się środowisko uruchomieniowe CLR (ang. *Common Language Runtime*) oraz biblioteki klas zawierające klasy i interfejsy pozwalające na tworzenie aplikacji dla systemu Windows. ASP.NET MVC jest jedną z części platformy .NET. Wszystkie aplikacje w ramach .NET mają podstawowe wspólne cechy zapewniające zgodność, stabilność oraz bezpieczeństwo. Dzięki CLI (ang. *Common Language Infrastructure*) platforma .NET nie ogranicza się tylko do jednego języka programowania. Pozwala ona pisać programy i aplikacje w wielu językach, które są kompilowane do wspólnego kodu pośredniego CIL (ang. *Common Intermediate Language*). Najważniejsze z nich to: C#, C++, Visual Basic .NET, J# oraz F#. Aby język był kompatybilny z .NET, musi spełniać odpowiednie standardy. Wiele z nich musiało zostać zmodyfikowanych, aby były zgodne z .NET, dlatego dodaje się im przyrostek .NET w nazwie (np. Visual Basic .NET)¹. Na strukturę .NET składają się:

- ◆ biblioteki klas lub framework:
 - ◆ AJAX,
 - ◆ ASP.NET,
 - ◆ ADO.NET (dostęp do danych),
 - ◆ MEF (ang. *Managed Extensibility Framework*),

¹ <http://msdn.microsoft.com/pl-pl/library/w0x726c2.aspx>

- ◆ WCF (ang. *Windows Communication Foundation*),
- ◆ WPF (ang. *Windows Presentation Foundation*),
- ◆ usługi Web Services,
- ◆ Web Forms,
- ◆ Silverlight,
- ◆ Entity Framework,
- ◆ Windows Forms,
- ◆ XML, XAML,
- ◆ LINQ,
- ◆ Networking,
- ◆ klasy bazowe;
- ◆ wspólne środowisko uruchomieniowe (CLR+DLR);
- ◆ Visual Studio .NET;
- ◆ języki:
 - ◆ Visual Basic .NET,
 - ◆ C++,
 - ◆ C#,
 - ◆ J#,
 - ◆ F#,
 - ◆ JScript .NET.

CLI

CLI (architektura wspólnego języka) to część platformy .NET, która odpowiada za komplikację programu napisanego w dowolnym kompatybilnym z CLI języku do kodu pośredniego. Każdy z ponad 40 kompatybilnych języków z CLI jest kompilowany do uniwersalnego kodu pośredniego — CIL, który przypomina kod języka Asembler. CLI powstało, aby rozwiązać problemy z nieprzenośnością programów między różnymi procesorami lub wersjami systemów operacyjnych. Specyfikacja CLI:

- ◆ wspólny zestaw typów danych CTS (ang. *Common Type System*) — zbiór typów, które są wspólne dla wszystkich obsługiwanych języków;
- ◆ metadane — informacje o strukturze kodu, pozwalające na wykorzystanie w różnych językach;
- ◆ wspólna specyfikacja języków CLS (ang. *Common Language Specification*) — zbiór reguł dla języków kompatybilnych z CLI, które należy respektować, aby zachować kompatybilność z innymi językami;

- ♦ wirtualny system wykonawczy VES (ang. *Virtual Execution System*) — ładuje i wykonuje programy kompatybilne z CLI, wykorzystując do tego celu metadane.

CIL

Wspólny język pośredni (CIL), początkowo nazywany był również MSIL (ang. *Microsoft Intermediate Language*). CIL jest komplikowany bezpośrednio na kod bajtowy i przypomina obiektowy asembler. Kod CIL jest niezależny od zestawu instrukcji procesora, co umożliwia jego translację na kod natywny. CLI definiuje instrukcje niezależne od platformy, które są przekazywane do CLR i tłumaczone na instrukcje dla konkretnej platformy.

CLR

CLR to wspólne środowisko uruchomieniowe dla całego .NET. Środowisko CLR komplkuje kod zapisany w CIL/MSIL na postać zrozumiałą dla danego komputera i wykonuje go. CLR ładuje potrzebne klasy do pamięci, po czym wykonywana jest weryfikacja, a następnie komplikacja do kodu maszynowego, który znajduje się w pamięci do czasu zakończenia działania aplikacji. CLR zawiera również mechanizmy kontroli bezpieczeństwa wykonywanych aplikacji, obsługę błędów, a także mechanizmy zarządzania pamięcią. Można wyróżnić dwie metody komplikacji wykonywanej przez CLR:

- ♦ JIT (ang. *Just-In-Time compilation*) — komplkuje metody w momencie ich pierwszego uruchomienia, a skompilowany kod nie jest zapisywany na dysku twardym, przez co przy kolejnym uruchomieniu kod będzie ponownie komplikowany;
- ♦ NGEN (ang. *Native image GENERator*) — generuje cały kod binarny, co pozwala na szybsze uruchamianie aplikacji, ponieważ podzespoły nie muszą być komplikowane przez JIT.

DLR

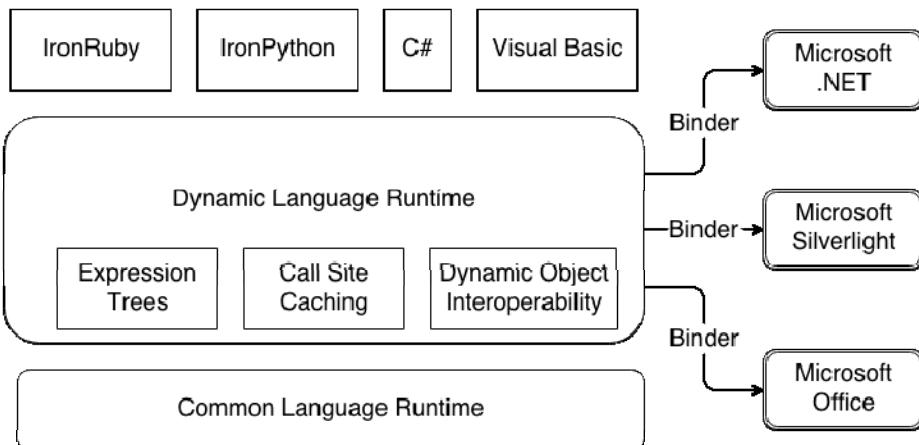
DLR (ang. *Dynamic Language Runtime*) to rozszerzenie CLR. Dodaje zestaw usług pozwalających na współpracę z językami dynamicznymi. W językach dynamicznych można zmieniać typy w czasie działania programu, na co nie pozwalało środowisko CLR. Dzięki DLR obiekt utworzony w dynamicznym języku, np. IronPython, może zostać wykorzystany w języku C# bez żadnych dodatkowych konwersji typów². Usługi wprowadzone wraz z DLR:

- ♦ dynamiczne typy — słowo kluczowe `dynamic`,
- ♦ dynamiczne drzewa wyrażeń (ang. *Expression Trees*),
- ♦ zapamiętuje w pamięci podrzcznej informacje na temat operacji i typów obiektów dynamicznych w celu przyspieszenia operacji na tych typach (ang. *Call Site Caching*),

² <http://msdn.microsoft.com/pl-pl/library/dd233052.aspx>

- ◆ zestaw klas i interfejsów dostępnych dla programistów reprezentujących dynamiczny obiekt i operacje na nim (ang. *Dynamic Object Interoperability*).

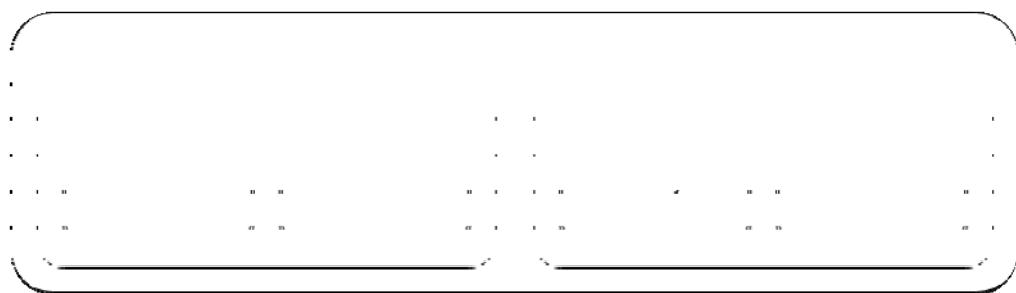
Architektura DLR została przedstawiona na rysunku 3.1.



Rysunek 3.1. Architektura DLR

Elementy .NET wykorzystywane w ASP.NET MVC

Najważniejsze elementy .NET, które są wykorzystywane do tworzenia aplikacji w ASP.NET MVC, i ich wzajemne zależności można zobaczyć na schemacie zaprezentowanym na rysunku 3.2.



Rysunek 3.2. Struktura .NET a MVC

Implementacje .NET

.NET Framework może być w pełni i bez żadnych problemów wykorzystywany tylko w systemie Windows, dlatego powstały implementacje framework'u .NET pozwalające na tworzenie aplikacji działających na darmowych platformach, takich jak np. Linux. Najpopularniejsze implementacje .NET:

- ♦ Mono,
- ♦ DotGNU,
- ♦ Portable.NET.

Projekt Mono

Mono to darmowy projekt open source prowadzony przez firmę Novell/Xamarin. Projekt ma na celu utworzenie darmowej platformy programistycznej zgodnej z Microsoft .NET Framework, w skład której wchodzi kompilator języka C# oraz CLR. Głównym celem Mono jest umożliwienie uruchomienia aplikacji .NET na innych systemach operacyjnych niż Windows oraz zachęcenie większej liczby programistów do programowania na platformie Linux. Aktualna wersja Mono jest w zgodna z .NET 4.5 i C# 5.0, jednak nie należy się spodziewać, że aplikacje uruchomią się bez problemów. Od wersji MVC 6 firma Microsoft zapowiedziała, że będzie brała pod uwagę także zgodność z Mono, co powinno zwiększyć popularność i zmniejszyć liczbę problemów napotykanych podczas uruchamiania aplikacji MVC na Linuksie³. Systemy operacyjne, które wspiera Mono:

- ♦ Linux,
- ♦ Mac OS X,
- ♦ Solaris,
- ♦ FreeBSD,
- ♦ OpenBSD,
- ♦ Wii,
- ♦ PlayStation 3,
- ♦ Android,
- ♦ Microsoft Windows.

Języki wspierane przez Mono: C#, Java, Boo, Nemerle, Visual Basic .NET, Python, JavaScript, Oberon, PHP, Object Pascal, Lua, Cobra.

WPF

WPF (ang. *Windows Presentation Foundation*) to framework do tworzenia interaktywnych aplikacji klienckich na platformę Windows. WPF opiera się na wzorcu MVVM i posiada rozbudowaną listę bibliotek graficznych. Bardzo dużą zaletą WPF jest to, że wykorzystuje grafikę wektorową, aby zachować proporcje przy różnych rozdzielczościach. WPF korzysta z języka XAML do tworzenia interfejsu użytkownika. XAML opiera się na składni XML i pozwala odseparować interfejs użytkownika od logiki aplikacji⁴.

³ <http://www.mono-project.com/>

⁴ <http://msdn.microsoft.com/pl-pl/library/ms754130.aspx>

WCF

WCF (ang. *Windows Communication Foundation*) to framework służący do tworzenia usług sieciowych (serwisów) oraz łączenia i wymiany informacji pomiędzy aplikacjami. WCF pozwala na komunikację pomiędzy aplikacjami utworzonymi w różnych technologiiach oraz obsługuje różne protokoły komunikacji, takie jak HTTP, SOAP, REST, UDP i TCP. WCF jest podobny do ASP.NET Web API, jednak ma o wiele większe możliwości. Główną przewagą WCF nad Web API jest możliwość komunikacji za pomocą TCP/IP (komunikacja binarna — szybka), a nie — jak w przypadku Web API — tylko przy użyciu protokołu HTTP (komunikacja tekstowa — wolniejsza).

Service Contract

Service Contract opisuje operacje (metody) udostępnione przez serwis. Aby oznaczyć interfejs lub klasę jako serwis WCF, należy poprzedzić ją atrybutem `[ServiceContract]`.

Operation Contract

Operation Contract to metody, z których można korzystać poprzez serwis. Aby oznaczyć metodę, która ma być udostępniona w serwisie i być częścią *Service Contract*, należy ją poprzedzić atrybutem `[OperationContract]`. Przykładowo: interfejs oznaczamy atrybutem `[ServiceContract]`, a deklaracje metod z tego interfejsu, które mają zostać udostępnione w serwisie oznaczamy jako `[OperationContract]`.

Data Contract

Data Contract określa dane, które mają być przesypane pomiędzy klientem a serwistem (zarówno przesypane do klienta, jak i przyjmowane jako parametry). Atrybut `[DataContract]` służy do serializacji danych i oznacza się nim klasę, która ma być serializowana.

Data Member

Atrybutem `[DataMember]` (*Data Member*) oznacza się właściwości z klasy (posiadającej atrybut `[DataContract]`), które mają być serializowane i dołączone do *Data Contract*. Zarówno *Data Contract*, jak i *Data Member* są opcjonalne. Jeśli się ich nie użyje, to domyślnie wszystkie pola publiczne zostaną przez WCF potraktowane jako *Data Member*.

Listing 3.1 prezentuje przykładową definicję, a listing 3.2 — implementację serwisu WCF.

Listing 3.1. Definicja interfejsu *ServiceContract* i klasy *DataContract*

```
namespace WcfService
{
    [ServiceContract]
    public interface IUzytkownikService
```

```
{  
    [OperationContract]  
    Uzytkownik PobierzDanePrzezDataContract(Uzytkownik  
        ↳uzytkownik);  
}  
  
[DataContract]  
public class Uzytkownik  
{  
    bool czyAktywny = true;  
    string imie = "Franek";  
  
    [DataMember]  
    public bool SprawdzCzyAktywny  
    {  
        get { return czyAktywny; }  
        set { czyAktywny = value; }  
    }  
  
    [DataMember]  
    public string Imie  
    {  
        get { return imie; }  
        set { imie = value; }  
    }  
}
```

Listing 3.2. Implementacja serwisu (dziedziczy po IUzytkownikService)

```
namespace WcfService  
{  
    public class UzytkownikService : IUzytkownikService  
    {  
        public Uzytkownik PobierzDanePrzezDataContract(Uzytkownik  
            ↳uzytkownik)  
        {  
            if (uzytkownik == null)  
            {  
                throw new ArgumentNullException("composite");  
            }  
  
            if (uzytkownik.SprawdzCzyAktywny)  
            {  
                uzytkownik.Imie += " jest aktywny";  
            }  
  
            return uzytkownik;  
        }  
    }  
}
```

WCF Endpoint = adres + binding + contract

W trakcie konfiguracji serwisu WCF ustawia się *Endpoint*, czyli punkt dostępowy. *Endpoint* składa się z trzech elementów. Są to:

- ◆ *adres*, pod którym można korzystać z Web serwisu; dla żądań HTTP będzie to adres internetowy, natomiast dla komunikacji TCP — adres lokalny;
- ◆ *binding*, który określa, jakiej metody transportu użyć (HTTP, TCP, UDP itp.), jak enkodować wiadomość (*text*, *binary*) oraz jakie protokoły są wymagane (sesje, bezpieczeństwo, SOAP, WebSockets);
- ◆ *contract* — wskazuje interfejs oznaczony jako *ServiceContract*, na którym ma działać dany *Endpoint*⁵.

Dla każdego kontraktu, formy transportu i „wiazania” należy utworzyć osobny *Endpoint*. Jeden serwis WCF może zawierać wiele punktów dostępowych zwracających te same lub różne dane dla różnych metod transportu i komunikacji („wiązań”). Punkty dostępowe dodaje się w pliku konfiguracyjnym *Web.config*. Aby utworzyć projekt WCF, wystarczy wybrać *Nowy projekt/WCF* w Visual Studio. Poniżej zaprezentowano przykładową konfigurację *WCF Endpoint*:

```
<configuration>
  <system.serviceModel>
    <services>
      <service name="WcfService.UzytkownikService">
        <endpoint
          address="http://localhost:8080/Uzytkownik/"
          binding="basicHttpBinding"
          contract="WcfService.IUzytkownikService">
        </endpoint>
      </service>
    </services>
  </system.serviceModel>
</configuration>
```

Silverlight

Silverlight to biblioteka pozwalająca na wyświetlanie treści multimedialnych za pomocą przeglądarki internetowej. Wykorzystuje języki XAML oraz JavaScript i pozwala na dynamiczne ładowanie danych. Silverlight jest konkurencją dla Adobe Flash — pozwala na przechwytywanie zdarzeń myszy i klawiatury, wyświetlanie grafiki oraz odtwarzanie zdjęć i filmów poprzez przeglądarkę. Silverlight umożliwia tworzenie aplikacji internetowych podobnych do ASP.NET, jednak w przypadku Silverlight aplikacja działa po stronie klienta, a nie po stronie serwera⁶.

⁵ <http://msdn.microsoft.com/en-us/library/dd456779.aspx>

⁶ [http://msdn.microsoft.com/en-us/library/cc838158\(v=vs.95\).aspx](http://msdn.microsoft.com/en-us/library/cc838158(v=vs.95).aspx)

Microsoft Azure

Microsoft Azure to rozwiązanie chmury obliczeniowej (ang. *Cloud Computing*), oferujące moc obliczeniową, przestrzeń magazynową i mechanizmy zarządzania. W chmurze Microsoftu można uruchomić praktycznie każdą aplikację webową.

Windows Azure Storage

Azure Storage to część chmury *Azure* odpowiedzialna za przechowywanie danych. Można wyróżnić trzy sposoby przechowywania danych w *Azure Storage* — każdy z nich ma inne możliwości oraz różne ceny i koszty utrzymania. Oprócz poniżej wymienionych opcji dostępnych w *Azure Storage* możemy również skorzystać z relacyjnej bazy danych *Azure SQL Database*, która stanowi osobny produkt i nie wchodzi w skład *Azure Storage*.

BLOB Storage

BLOB (ang. *Binary Large Object*) pozwala na przechowywanie danych binarnych nieposiadających struktury, takich jak zdjęcia, filmy czy długie teksty. Maksymalna wielkość przechowywanych danych to 200 GB dla Block BLOB i 1 TB dla Page BLOB. Dostęp do zawartości można uzyskać poprzez identyfikator BLOB-y, tak jak inne pojemniki w *Azure*, mają architekturę REST — co oznacza, że do danych odwołuje się za pomocą wywołań HTTP.

Table Storage

Table Storage jest certyfikowaną bazą NoSQL przechowującą dane w formacie klucz-wartość. Tabela może mieć wiele pól, jednak nie można tworzyć relacji pomiędzy tabelami. Aby korzystać z bazy relacyjnej, należy użyć Windows Azure SQL Database.

Queue Storage

Queue Storage to kolejka wiadomości pozwalająca na przechowywanie i przekazywanie komunikatów pomiędzy niezależnymi komponentami aplikacji. Komponent zlecający, czyli ten, który utworzył komunikat, dodaje go do kolejki, a następnie komponent wykonujący zdejmuję komunikat z kolejki i go wykonuje. *Queue Storage* nie gwarantuje zachowania kolejności komunikatów, nie wspiera transakcyjności i pozwala na przechowywanie wiadomości do 7 dni. Jeśli w aplikacji wymagana jest transakcyjność lub kolejność, należy skorzystać z *Service Bus Queue*.

Hostowanie aplikacji w Azure

Chmura Windows *Azure* pozwala na różne sposoby hostowania aplikacji. Każdy z nich ma inne możliwości, ceny oraz ograniczenia.

Worker Role

Worker Role to opcja służąca do hostowania długotrwałych procesów obliczeniowych działających w tle, które nie wymagają serwera IIS, np. wysyłanie wiadomości e-mail. Znajduje zastosowanie w hostowaniu aplikacji WCF i Web API. *Worker Role* mogą się komunikować z *Web* i *Worker Role* za pomocą kolejek *Queue Storage* lub *Service Bus*. *Worker Role* są otwarte na komunikację przychodząą z internetu i posiadają własny publiczny adres IP.

Web Role

Web Role pozwala na hostowanie serwisów WCF, aplikacji ASP.NET, Web API oraz Node.js. *Web Role* może się komunikować z dowolnymi komponentami Windows Azure, takimi jak kolejki, *Azure Storage* czy bazy danych SQL Azure. *Web Role* pozwala na dostęp poprzez pulpit zdalny, idealnie pasuje do aplikacji wielowarstwowych oraz posiada dostęp do *Azure Virtual Network* do komunikacji z innymi *Web Role* lub maszynami wirtualnymi. Każda *Web Role* jest hostowana na osobnej maszynie wirtualnej.

Web Site

Web Site jest bardzo podobny do *Web Role*, jednak różni się kilkoma aspektami. Pozwala na błyskawiczne skalowanie, ponieważ korzysta z już działających serwerów wirtualnych, ale nie wymaga tworzenia nowych, tak jak ma to miejsce w *Web Role*. Nie ma możliwości dostępu poprzez pulpit zdalny, nie ma dostępu do sieci *Azure Virtual Network*, ale wspiera takie narzędzia jak GIT, Mercurial i TFS. *Web Site* posiada ograniczenie do maksymalnie 10 serwerów wirtualnych i 10 GB miejsca na dysku.

Virtual Machine

Azure Virtual Machine pozwala na utworzenie maszyny wirtualnej w chmurze. Do wyboru są systemy operacyjne takie jak Windows Server oraz Linux. Maszyny wirtualne pozwalają na dowolną konfigurację. Można na nich instalować takie narzędzia jak SQL Server, SharePoint Server, BizTalk Server i wiele innych. W porównaniu do prywatnej farmy serwerów nie ma się wpływu na fizyczny sprzęt oraz możliwości wirtualizacji, ponieważ maszyna wirtualna już korzysta z tej technologii.

Azure Service Bus

Azure Service Bus to technologia, która służy do komunikacji pomiędzy aplikacjami lub rozproszonymi częściami aplikacji hostowanymi w chmurze oraz lokalnie⁷.

Service Bus Relay

Kolejka *Service Bus Relay* pozwala na przesyłanie komunikatów pomiędzy aplikacjami hostowanymi w chmurze a aplikacjami hostowanymi na serwerach lokalnych. Może także służyć do komunikacji chmura – chmura lub serwer lokalny – serwer lokalny. *Service Bus Relay* wspiera zarówno komunikację jednokierunkową, jak i komunikację

⁷ <http://azure.microsoft.com/en-us/documentation/articles/fundamentals-service-bus-hybrid-solutions/>

w formacie żądanie-odpowiedź. Nie ma bezpośredniego połączenia pomiędzy nadawcą a odbiorcą. *Service Bus Relay* jest pośrednikiem, który zarządza komunikatami i bierze odpowiedzialność za ich przekazywanie.

Service Bus Queue

Service Bus Queue to jednokierunkowa asynchroniczna kolejka przesyłająca komunikaty, podobna do *Queue Storage*, jednak w przeciwieństwie do tej drugiej działa na zasadzie mechanizmu kolejkowania FIFO (odbiorcy jest zwracana najstarsza wiadomość). Wspiera transakcje oraz może się komunikować z serwisami WCF. Maksymalna długość czasu, przez jaką może być przechowywany komunikat, to 5 minut. W *Queue Storage* było to 7 dni.

Service Bus Topic

Service Bus Topic to kolejka, która działa na zasadzie wydawcy i subskrybentów. Wydawca może mieć wielu subskrybentów. Każdy subskrybent ma swój własny filtr, który określa, jakie komunikaty mają do niego docierać.

ASP.NET Web Forms

ASP.NET *Web Forms* to najstarsza część framework'u ASP.NET do budowania stron internetowych. Bazuje na programowaniu stron zorientowanych na zdarzenia, zbudowanych z gotowych kontrolek i komponentów (przycisków, pól tekstowych itp.), języka HTML oraz kodu (C#, Virtual Basic .NET) działającego po stronie serwera. Strony są komplikowane po stronie serwera i przesyłane do przeglądarki jako kod HTML. Obecnie praktycznie nie jest już stosowane⁸.

Zalety *Web Forms*:

- ♦ oddzielenie kodu HTML i interfejsu użytkownika — UI (ang. *User Interface*) — od logiki aplikacji;
- ♦ duża liczba gotowych komponentów do wykorzystania poprzez system „przeciagnij i upuść”;
- ♦ zapewnia szybszą realizację małych projektów, a do większych stron wymagających lepszej wydajności używa się ASP.NET MVC;
- ♦ oparty na zdarzeniach i stanach;
- ♦ łatwiejszy do nauki;
- ♦ posiada wsparcie dla AJAX i JavaScript.

⁸ <http://msdn.microsoft.com/en-us/library/ms973868.aspx>

Wady *Web Forms*:

- ◆ nie zapewnia pełnej kontroli nad kodem wynikowym HTML;
- ◆ mniejsza wydajność przy większych projektach z powodu kontroli stanów kontrollek;
- ◆ brak wsparcia dla testów jednostkowych;
- ◆ słaba optymalizacja pod wyszukiwarki (SEO);
- ◆ niska „reżywalność” kodu;
- ◆ brak wsparcia dla SoC (ang. *Separation of Concern*), brak oddzielenia kodu od warstwy prezentacji;
- ◆ brak budowy modułowej.

ASP.NET Web Pages

ASP.NET *Web Pages* to framework do tworzenia małych, prostych stron internetowych. Strony *Web Pages* można tworzyć za pomocą darmowego edytora WebMatrix. WebMatrix to aplikacja, która zawiera edytor tekstu stron, narzędzia do tworzenia oraz dostępu do bazy danych, serwer testowy aplikacji oraz wsparcie dla łatwej publikacji strony w internecie. *Web Pages*, podobnie jak ASP.NET MVC, korzysta ze składni języka Razor do generacji widoku. *Web Pages* jest technologią łączącą elementy *Web Forms* i MVC, jednak nie wymaga środowiska Visual Studio. Jest to młoda technologia, która jest raczej rzadko stosowana⁹.

ADO.NET

ADO.NET jest technologią zarządzającą dostępem do danych. Składa się na nią zbiór klas i interfejsów do łączenia się i zarządzania bazą danych z poziomu aplikacji .NET. ADO.NET pozwala także na zapis i odczyt danych w plikach XML i plikach tekstowych. ADO.NET wspiera następujące źródła danych: SQL Server, Azure SQL, OLE DB, ODBC oraz Oracle.

ADO.NET zawiera obiekt o nazwie *DataSet*, który działa jak standardowa relacyjna reprezentacja danych znajdująca się w pamięci. Ponieważ obiekt *DataSet* z założenia nie ma stałego połączenia z bazą danych, doskonale nadaje się do pakowania, wymiany, buforowania, przechowywania i ładowania danych, które nie wymagają natychmiastowego zapisu do bazy danych. Obiekt *DataSet* korzysta z .NET Framework Data Provider, który jest częścią ADO.NET, i w zależności od źródła danych wybiera odpowiedni typ providera, który potrafi komunikować się z wybranym źródłem.

⁹ [http://msdn.microsoft.com/en-us/library/fddycb06\(v=vs.100\).aspx](http://msdn.microsoft.com/en-us/library/fddycb06(v=vs.100).aspx)

Obiekt DataSet

Korzyści płynące z zastosowania DataSet:

- ♦ zmniejsza obciążenie spowodowane stałym utrzymywaniem otwartego połączenia z bazą danych;
- ♦ zapobiega tworzeniu zduplikowanych połączeń z bazą danych, w przypadku gdy dwa komponenty muszą mieć dostęp do tych samych danych;
- ♦ poprawia skalowanie aplikacji, które wymagają otwartego połączenia z bazą danych;
- ♦ łączy się z bazą danych tylko na czas odczytu lub zapisu danych, dzięki czemu może obsługiwać więcej użytkowników;
- ♦ buforuje rekordy pobrane z bazy danych;
- ♦ zapewnia niezależność od źródła danych, ponieważ jest on tylko kontenerem na dane, co pozwala na przechowywanie danych z kilku źródeł;
- ♦ zawiera w sobie kompletny obraz danych, tabele oraz relacje pomiędzy tabelami.

Obiekty DataTable i DataRow

Obiekt DataSet zawiera w sobie kolekcję obiektów DataTable reprezentujących tabele w bazie danych. Wszystkie obiekty DataTable są przechowywane w kolekcji DataTableCollection. Obiekt DataTable posiada kolekcję DataRowCollection przechowującą listę obiektów DataRow, które są odpowiednikiem wierszy z tabeli w bazie danych. Każdy obiekt DataRow posiada swój własny stan oraz wartość początkową i aktualną, aby sprawdzić, czy dane zmieniły się od czasu pobrania z bazy danych.

Obiekt DataRelation

Obiekt DataSet przechowuje relacje pomiędzy tabelami w obiekcie DataRelation, który łączy wiersz z jednego obiektu DataTable (ang. *Primary Key*, PK — klucz główny) z wierszem z drugiego obiektu DataTable (ang. *Foreign Key*, FK — klucz obcy). Wszystkie obiekty DataRelation są przechowywane w kolekcji DataRelationCollection.

Obiekt DataView

Obiekt DataView pozwala na prezentację danych z DataTable w odmiennej formie, np. posortowane lub przefiltrowane dane.

.NET Framework Data Provider

.NET Framework Data Provider składa się z obiektów, które pozwalają na połączenie z bazą danych, wykonanie zapytań oraz zwrocenie wyniku. Wynik zapytania jest składowany w obiekcie DataSet, który w razie potrzeby zwraca dane użytkownikowi, łączy dane z różnych źródeł lub przechowuje dane pomiędzy warstwami aplikacji¹⁰.

¹⁰ <http://msdn.microsoft.com/pl-pl/library/e80y5yh.aspx>

Główne obiekty .NET Data Provider:

- ◆ Connection — zapewnia połączenie z określonym źródłem danych;
- ◆ Command — wykonuje zapytanie na wybranym źródle danych; udostępnia parametry oraz może zostać wykonany jako transakcja przy użyciu połączenia (Connection);
- ◆ DataReader — odczytuje jednokierunkowy strumień danych (tylko do odczytu) z wybranego źródła danych; wymaga aktywnego połączenia ze źródłem danych; pobiera dane w sposób iteracyjny — jedne po drugich; możliwe jest pobieranie tylko kolejnej porcji danych podczas iteracji; dane nie są przechowywane w pamięci (obiekcie DataSet); korzysta z obiektu Connection, aby nawiązać połączenie, oraz metody ExecuteReader() wywoływanej dla obiektu Command, aby pobrać dane z bazy danych;
- ◆ DataAdapter — jest pośrednikiem pomiędzy obiektem DataSet a bazą danych; pobiera dane z bazy danych i zapisuje je do obiektów DataSet; DataAdapter nie wymaga ciągłego połączenia z bazą danych, pobiera dane i zamkna połączenie; korzysta z obiektu Connection, aby nawiązać połączenie, oraz obiektu Command, aby pobrać dane z bazy danych.

Pozostałe obiekty .NET Data Provider:

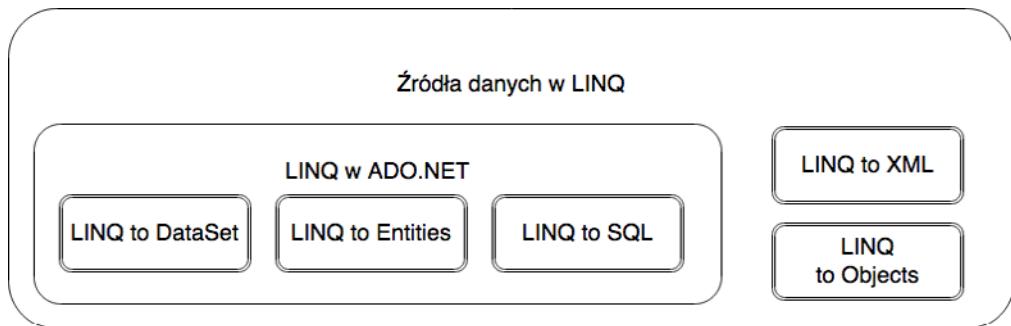
- ◆ Transaction,
- ◆ CommandBuilder,
- ◆ ConnectionStringBuilder,
- ◆ Parameter,
- ◆ Exception,
- ◆ Error i ClientPermission.

LINQ

LINQ (ang. *Language INtegrated Query*) to nowość wprowadzona w C# 3.0. Jest to rodzaj mostu pomiędzy danymi składowanymi w obiektach (pamięci) a danymi pochodzącymi ze źródeł danych. Dzięki LINQ — bez względu na to, z jakiego źródła danych się korzysta — zapytania napisane za pomocą składni LINQ wyglądają tak samo. W zależności od źródła danych wykorzystywany jest inny pośrednik tłumaczący zapytanie LINQ na kod zrozumiały przez dane źródło danych; dla relacyjnych baz danych jest to np. język SQL (rysunek 3.3).

LINQ to XML

LINQ to XML dostarcza interfejs programistyczny (bibliotek) pozwalający na proste tworzenie i odczytywanie dokumentów XML za pomocą składni LINQ.



Rysunek 3.3. Struktura LINQ

LINQ to Objects

LINQ to Objects pozwala na odpytywanie kolekcji (znajdujących się w pamięci operacyjnej), takich jak `IEnumerable<T>`, `List<T>`, `Dictionary< TKey, TValue >`. Dzięki *LINQ to Objects* można za pomocą składni LINQ pobierać dane z kolekcji bez użycia pętli `foreach`. Zapytania LINQ są dużo bardziej czytelne, pozwalają filtrować wyniki bez użycia instrukcji warunkowych, a ponadto umożliwiają sortowanie i grupowanie danych z kolekcji.

LINQ to SQL

LINQ to SQL to technologia mapująca bazę danych do postaci obiektowej i pozwalająca na odpytywanie bazy danych za pomocą zapytań języka LINQ. Składnia języka LINQ jest bardzo podobna do języka SQL. Zapytania LINQ są tłumaczone na zapytania SQL i przesyłane do bazy danych. Następnie wynik zapytania zwrocony z bazy danych jest tłumaczony na obiekty, na których działa LINQ.

LINQ to DataSet

LINQ to DataSet pozwala na odpytywanie za pomocą zapytań LINQ danych przechowywanych w obiekcie `DataSet`. Dane w obiekcie `DataSet` są przechowywane w pamięci i mogą pochodzić z różnych źródeł. Składnia LINQ bardzo upraszcza odczyt danych i skracą czas potrzebny na napisanie zapytań.

LINQ to Entities

LINQ to Entities to implementacja LINQ pozwalająca na wykorzystanie składni LINQ do odpytywania źródła danych Entity Framework. Podstawowe zapytania wyglądają tak samo jak w *LINQ to SQL*, jednak technologia *LINQ to Entities* w połączeniu z Entity Framework ma dużo więcej funkcjonalności, przez co pozwala pisać bardziej rozbudowane zapytania.

Najważniejsze operatory zapytań LINQ:

- ◆ Select — wybiera z kolekcji odpowiedniego rodzaju dane (zbiory lub podzbiory danego obiektu);
- ◆ Where — zwraca obiekty spełniające dany warunek;
- ◆ Take — jest używany do wybrania pierwszych n obiektów z kolekcji;
- ◆ Sum — zwraca sumę;
- ◆ Min — zwraca minimalną wartość;
- ◆ Max — zwraca maksymalną wartość;
- ◆ OrderBy — sortuje wyniki po wybranym elemencie kolekcji;
- ◆ Count — zwraca liczbę wierszy w tabeli;
- ◆ First — zwraca pierwszy wynik spełniający dany warunek;
- ◆ Last — zwraca ostatni wynik spełniający dany warunek.

Przykłady zapytań LINQ

Zapytania LINQ można zapisywać na dwa różne sposoby, które mają identyczne znaczenie. Pierwszy z nich to zapis w postaci metod, natomiast drugi to zapis w postaci zapytań bardzo podobnych do tych z języka SQL. Każde zapytanie LINQ w środowisku .NET jest komplikowane i tłumaczone na wywołania drzew wyrażeń, a więc na składnię metod. Składnia zapytań jest tłumaczona na składnię metod. Tłumaczenie odbywa się w trakcie komplikacji, dlatego nie wpływa na wydajność aplikacji podczas działania. Składnia metod wydaje się prostsza, jednak nie każde zapytanie można zapisać za jej pomocą, a jeśli nawet, to może być ono bardzo skomplikowane (dużo bardziej niż w postaci składni zapytań). W większości prostych zapytań składnia metod jest bardziej odpowiednia, natomiast w skomplikowanych zapytaniach bardziej odpowiednia będzie składnia zapytań.

Składnia metod — Method Syntax

Składnia metod (ang. *Method Syntax*) bardzo dobrze pasuje do prostych zapytań. Dodatkowo Visual Studio podpowiada, jakich metod użyć. Składnia metod korzysta z wyrażeń lambda dla skrócenia zapisu metod. Wyrażenia lambda zostały wprowadzone do języka C# równocześnie z LINQ.

Oto przykład zapytania — składnia metod:

```
 IEnumerable<int> zapytanie = zrodloDanych.Where(dane =>
    ➔ dane.id > 100).OrderBy(dane => dane.id);

// Lub równoznaczny, bardziej czytelny zapis odpowiedni dla dłuższych zapytań

 IEnumerable<int> zapytanie = zrodloDanych
    ➔ .Where(dane => dane.id > 100)
    ➔ .OrderBy(dane => dane.id);
```

Składnia zapytań — Query Syntax

Składnia zapytań (ang. *Query Syntax*) jest podobna do zapytań z języka SQL. Razem z LINQ zostały wprowadzone nowe słowa kluczowe do języka C# (`select`, `from`, `where`), które są używane właśnie w zapytaniach LINQ.

Poniżej znajduje się przykład zapytania — składnia zapytań:

```
IEnumerable<int> zapytanie =  
    from dane in zrodloDanych  
    where dane.id > 100  
    orderby dane.id  
    select dane;
```

PLINQ

PLINQ (ang. *Parallel LINQ*) to równolegle wykonywane zapytania, które zostały wprowadzone w .NET 3.5. W przypadku zapytań, które muszą zostać wykonane sekwencyjnie (jedno po drugim — zadanie B potrzebuje danych z zadania A), nie powinno się stosować PLINQ ze względu na mniejszą wydajność. Natomiast przy operacjach, które mogą być wykonywane równolegle (zadanie B nie potrzebuje danych z zadania A), można, a nawet powinno się użyć PLINQ zamiast zwykłego LINQ. PLINQ dzieli jednocześnie zapytanie na części i obliczenia na kilka wątków, domyślnie dla każdego rdzenia w procesorze po jednym wątku. Istnieje możliwość zmiany liczby wątków za pomocą metody `WithDegreeOfParallelism()`, która określa, ile maksymalnie wątków może zostać utworzonych. W przypadku kilku wątków działających równocześnie pojawia się problem kolejności. Wyniki zapytania mogą nie być zwrócone w takiej samej kolejności, w jakiej byłyby zwrócone za pomocą zwykłego sekwencyjnego zapytania. Istnieje metoda `AsOrdered()`, która pozwala zachować kolejność, jednak powoduje utratę wydajności.

PLINQ powinno być stosowane tylko na komputerach z procesorem wielordzeniowym oraz w sytuacji, gdy jest do wykonania rozbudowane zapytanie, którego wykonanie zajmuje dużo czasu (listing 3.3). W przypadku prostych i krótkich zapytań nie ma sensu stosować PLINQ.

W każdym wypadku należy się zastanowić, które zapytanie będzie korzystniejsze. W niektórych sytuacjach można potrzebować więcej pamięci kosztem większego użycia procesora, natomiast w innych sytuacjach istnieje konieczność oszczędzania użycia procesora kosztem pamięci.

Listing 3.3. Przykładowy kod programu z użyciem PLINQ

```
IEnumerable<int> dane1 = Enumerable.Range(1, 100000000);  
  
var zapytanie1 = dane1.AsParallel().Where(x => x % 20000000 == 0)  
    ↳.Select(i => i);  
  
System.Diagnostics.Stopwatch sw = System.Diagnostics.Stopwatch.StartNew();  
  
foreach (var d in zapytanie1)  
    Console.WriteLine(d);  
Console.WriteLine("Czas PLINQ: {0} ms", sw.ElapsedMilliseconds);
```

```

    sw.Restart();

    var zapytanie = dane1.Where(x => x % 20000000 == 0)
        .Select(i => i);

    foreach (var d in zapytanie)
        Console.WriteLine(d);

    Console.WriteLine("Czas LINQ: {0} ms", sw.ElapsedMilliseconds);

    Console.ReadKey();

```

Wynik działania programu dla 100 000 000 liczb prezentuje tabela 3.1. Można zauważyc, że PLINQ było szybsze, jednak kolejność wyników nie jest taka sama jak w przypadku LINQ. Test był wykonywany przy użyciu 100 000 000 liczb typu int w kolekcji Enumerable. Wyniki testu dla stukrotnie mniejszej liczby zaprezentowano w tabeli 3.2. Dla stukrotnie mniejszej kolekcji wyniki są całkowicie odmienne — nie dość, że LINQ było niemal dwukrotnie szybsze, to jeszcze zwróciło prawidłową kolejność danych. Przy wykorzystaniu metody AsOrdered() w zapytaniu PLINQ czas wykonania był jeszcze większy i wynosił 60 ms.

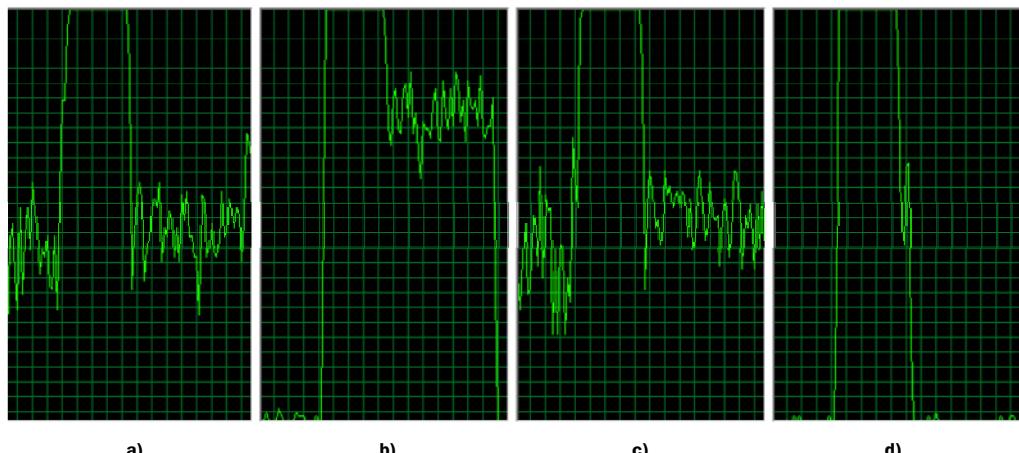
Tabela 3.1. Wynik działania programu dla 100 000 000 liczb

PLINQ — czas wykonania: 1623 ms	LINQ — czas wykonania: 2280 ms
60 000 000	20 000 000
80 000 000	40 000 000
20 000 000	60 000 000
100 000 000	80 000 000
40 000 000	100 000 000

Tabela 3.2. Wynik działania programu dla 1 000 000 liczb

PLINQ — czas wykonania: 46 ms	LINQ — czas wykonania: 26 ms
20 000 000	20 000 000
80 000 000	40 000 000
40 000 000	60 000 000
60 000 000	80 000 000
100 000 000	100 000 000

Jak przedstawiono na zrzutach ekranu na rysunku 3.4, podczas wykonywania zapytania PLINQ (mniej więcej środkowa część wykresu 3.4) wszystkie rdzenie pracowały z maksymalną szybkością, natomiast przy użyciu LINQ jeden rdzeń pracował na około 80% i dwa rdzenie po 50%, jednak były one wykorzystywane do innych obliczeń.



Rysunek 3.4. Użycie procesora LINQ i PLINQ: a) pierwszy rdzeń, b) drugi rdzeń, c) trzeci rdzeń i d) czwarty rdzeń

Narzędzia ORM w .NET

ORM (ang. *Object Relational Mapping*) to narzędzia mapujące bądź odwzorowujące dane relacyjne z tabel w bazie danych na obiekty wykorzystywane w aplikacji.

Entity Framework

Microsoft ADO.NET Entity Framework (EF) to biblioteka i zestaw narzędzi ORM, które pozwalają mapować dane relacyjne na obiekty. W aplikacji pracuje się z obiektami, które są automatycznie tłumaczone na zapytania zrozumiałe dla danego źródła danych. EF korzysta z technologii *LINQ to Entities*, pozwala na tworzenie i aktualizację baz danych i plików oraz na operacje na nich.

NHibernate

NHibernate to framework ORM dla .NET. Powstał na bazie Hibernate, czyli najwięcej ORM-a dla języka Java. NHibernate jest projektem open source i jest ciągle rozwijany przez użytkowników. NHibernate — w przeciwieństwie do Hibernate — korzysta z zapytań LINQ i posiada bardzo długą listę dostępnych bibliotek i rozszerzeń bazujących na tych z Hibernate. Największym problemem NHibernate jest brak dokumentacji. Jeszcze kilka lat temu NHibernate był dużo bardziej popularny niż Entity Framework, jednak kiedy ten drugi stał się projektem open source, sytuacja diametralnie się zmieniła. Entity Framework rozwija się w zawrotnym tempie, a NHibernate praktycznie stoi w miejscu¹¹.

¹¹ <http://nhforge.org/>

NHibernate 3 a Entity Framework 6

Różnice przedstawiono w formie tabeli 3.3.

Tabela 3.3. NHibernate a Entity Framework

	Entity Framework	NHibernate
Wsparcie baz danych	SQL Server B2 Wsparcie dla innych silników za pomocą Data Providers	SQL Server Oracle Microsoft Access Firebird PostgreSQL DB2 UDB MySQL SQLite
async/await (.NET 4.5)	TAK	NIE
Open source	TAK (wspierany przez MS)	TAK
Wsparcie .NET 4.5	TAK (z korzyściami)	TAK (bez korzyści)
Dokumentacja i wsparcie	TAK	NIE
Klasy POCO	TAK	TAK
Liczba dodatków/rozszerzeń	Mniej	Więcej
Caching	TAK	TAK
Zapytania do bazy	LINQ to Entities, Entity SQL, SQL	LINQ, HQL, Criteria API, SQL
Batching (grupowanie zapytań)	TAK (EF Extensions)	TAK
Cascading (kaskadowe operacje)	TAK	TAK
Lazy Loading	TAK	TAK
Eager Loading	TAK	TAK
Generatory ID	Identity generation, GUIDs, Assigned values	Identity, HiLo, Sequence, GUID, Sequence-style, Assigned, Native, Pooled itp.
Migracje	TAK	TAK

Alternatywa dla Entity Framework i NHibernate

Entity Framework i NHibernate to bardzo rozbudowane ORM-y z ogromną liczbą funkcji. Jednak istnieje prosta zależność pomiędzy abstrakcją lub liczbą funkcji dostępnych w narzędziu a jego wydajnością. Im więcej dodatków, tym mniejsza wydajność, dlatego w przypadku gdy niezbędna jest większa wydajność, wskazane jest użycie mikro ORM-ów , takich jak BLToolkit, PetaPoco itp. Są to ORM-y z minimalną liczbą funkcji, jednak mające bardzo dużą wydajność. Wymagają poświęcenia większej ilości czasu na utworzenie tego samego projektu, ale w późniejszym czasie włożona praca

zwraca się w większej wydajności aplikacji. W ekstremalnych przypadkach, gdy wymagana jest bezkompromisowa wydajność, nie powinno się używać narzędzi typu ORM, tylko zwracać dane bezpośrednio z procedur składowanych w bazie danych, bez opakowywania ich w obiekty. Oszczędza się wtedy czas wymagany na tworzenie obiektu z danymi zwróconymi z bazy danych, jednak traci się możliwość późniejszego na nich operowania. Aktualnie takie podejście jest wykorzystywane bardzo rzadko, ponieważ dane są kilkakrotnie przekształcane, obrabiane czy też przeliczane, zanim dotrą do użytkownika.

Rozdział 4.

Entity Framework 6

Entity Framework (EF) jest odpowiedzialny za kontakt aplikacji z bazą danych. Domyślne operacje wykonywane na bazie danych to operacje CRUD (ang. *Create, Read, Update, Delete*). Dzięki EF i Scaffolding (automatyczny generator kodu) można automatycznie wygenerować kod odpowiedzialny za operacje CRUD dla każdej tabeli z bazy danych.

Podejście do pracy z modelem danych

EF udostępnia trzy podejścia do pracy z warstwą dostępu do danych DAL (ang. *Data Access Layer*):

- ◆ *Model First* — tworzy się diagram ERD (ang. *Entity-Relationship Diagram*) w Visual Studio za pomocą Model Designera i na podstawie diagramu generowana jest baza danych (kod SQL) oraz wszystkie klasy z modelem danych, po jednej dla każdej tabeli.
- ◆ *Code First* — tworzy się samodzielnie klasy (odpowiadające tabelom w bazie danych) i na podstawie klas generowana jest baza danych (kod SQL); dodatkowo, aby uaktualnić bazę danych po zmianach wprowadzonych w plikach z klasami, wykorzystuje się migracje (aktualizacje bazy danych).
- ◆ *Database First* — klasy i diagram ERD są generowane lub uaktualniane na podstawie istniejącej bazy danych.

Porównanie różnych podejść

Model First pozwala na szybkie utworzenie bazy danych poprzez wstawianie elementów w edytorze graficznym, ale nie daje takiej kontroli nad kodem jak *Code First*. W przypadku korzystania z *Code First* trzeba ręcznie tworzyć pliki z klasami, co powoduje, że można się łatwo pomylić. *Database First* z kolei jest wykorzystywany w przypadku, gdy tworzy się aplikację na podstawie już istniejącej bazy danych. Dodatkowo po wybraniu bazy danych, na podstawie której chce się utworzyć projekt, wybiera się, czy w dalszej części zostanie użyte podejście *Code First*, czy *Model First*. W zależności od wyboru utworzone zostaną klasy (*Code First*) lub diagram (*Model First*).

Największą kontrolę nad kodem daje podejście *Code First*. Dla początkujących podejście *Model First* będzie bardziej intuicyjne, ponieważ baza jest prezentowana w formie diagramu ERD.

Model dla podejścia Model First

Model danych dla podejścia *Model First* jest zawarty w pliku *.edmx*. Domyślnie model jest prezentowany jako diagram ERD w Entity Designerze (narzędzie w Visual Studio do tworzenia diagramu ERD).

Na pełny model danych w podejściu *Model First* składają się trzy części:

- ◆ model konceptualny (ang. *Conceptual Schema Definition Language*),
- ◆ model źródła danych (ang. *Storage Schema Definition Language*),
- ◆ część odwzorowująca (ang. *Mapping Specification Language*).

Wszystkie części znajdują się w jednym pliku *.edmx*. Jest to plik XML. Na podstawie pliku *.edmx* generowany jest schemat bazy danych — diagram ERD.

Przykładowy schemat bazy danych utworzony w Entity Designerze prezentuje rysunek 4.1.

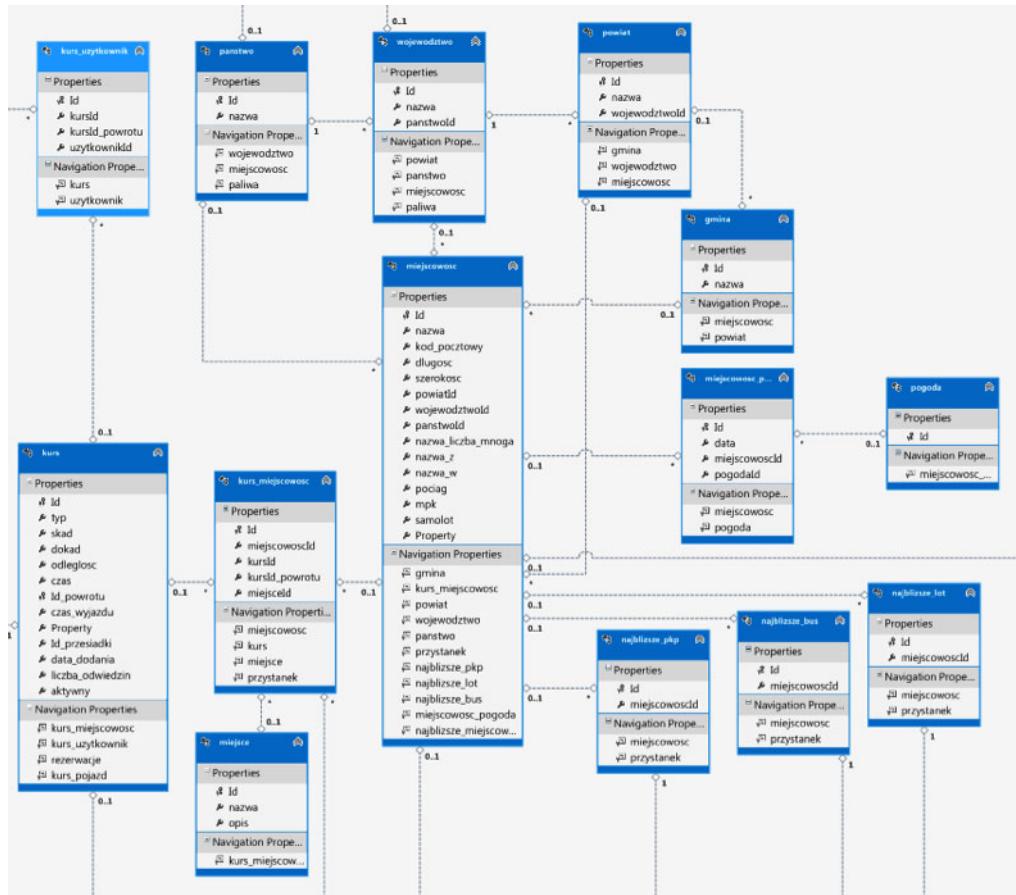
Model dla podejścia Code First

Przykładowa klasa z modelem dla podejścia *Code First* znajduje się na listingu 4.1. Jest to klasa POCO (ang. *plain-old CLR objects*), czyli klasa, która odwzorowuje struktury tabeli z bazy danych na klasę w podejściu obiektowym.

Listing 4.1. Przykładowa klasa POCO

```
public partial class Kurs
{
    public Kurs()
    {
        this.KursUser = new HashSet<KursUser>();
    }
    public int Id { get; set; }
    public string Skad { get; set; }
    public string Dokad { get; set; }
    public System.DateTime DataWyjazdu { get; set; }
    public System.DateTime DataDodania { get; set; }

    public virtual ICollection<KursUser> KursUser { get; set; }
}
```



Rysunek 4.1. Diagram ERD z Entity Designera

Nowości wprowadzane w kolejnych wersjach EF

Nowości wprowadzone w EF 5

Najważniejsze nowości wprowadzone w EF 5 to:

- ◆ możliwość tworzenia wielu diagramów dla jednego modelu (*Model First*);
- ◆ wsparcie dla typu wyliczeniowego enum w *Code First* i *Model First*;
- ◆ wsparcie dla typów danych przestrzennych (ang. *spatial data types*) w *Code First* i *Model First*;

- ◆ TVF (ang. *Table-Valued Functions*) w *Model First* — podobne do procedur składowanych, z jedną małą różnicą — dane z TVF mogą być użyte w zapytaniach LINQ, natomiast dane z procedury składowanej nie.

Nowości wprowadzone w EF 6

Najważniejsze nowości wprowadzone w EF 6 to:

- ◆ własne konwencje w *Code First*;
- ◆ poprawiona wydajność dla dużych modeli w podejściu *Code First*;
- ◆ migracje dla wielu kontekstów do jednej bazy danych;
- ◆ automatyczne ponawianie transakcji (ang. *Connection Resiliency*), w przypadku gdy zakończy się ona niepowodzeniem;
- ◆ tworzenie nowych kontekstów dla otwartego połączenia;
- ◆ poprawione wsparcie dla transakcji;
- ◆ wstrzykiwanie implementacji w czasie działania programu (ang. *Dependency Resolution*) oraz wsparcie dla wzorców *Service Locator* i *IoC*;
- ◆ łatwe przechwytywanie logów (ang. *Interception/SQL Logging*), zapytań SQL generowanych przez EF.

Relacyjne bazy danych i EF

Krótki opis baz relacyjnych

Bazy relacyjne powstały w czasach, gdy przestrzeń dyskowa była dość dużym ograniczeniem dla programów. Dyski miały pojemności liczone w megabajtach, a nie — jak obecnie — w terabajtach. Aby oszczędzić miejsce na dysku, dane musiały być przechowywane w oszczędny sposób bez duplikacji wpisów. Relacyjne bazy danych składają się z tabel, pomiędzy którymi zachodzą relacje. Każda tabela składa się z wierszy i kolumn. Na każdy wiersz składają się jednakowo ułożone kolumny wypełnione wartościami, które z kolei w każdym wierszu mogą być inne. Każdy wiersz ma swój unikalny numer zwany kluczem podstawowym PK (ang. *Primary Key*), po którym można rozpoznać konkretny wiersz danych (np. konkretnego pracownika z tabeli *Pracownik*). Aby powiązać wiersz z jednej tabeli z wierszem znajdującym się w innej tabeli, należy umieścić w niej klucz obcy FK (ang. *Foreign Key*). Jest to ta sama wartość, która w pierwszej tabeli jest kluczem podstawowym (PK). Druga tabela może np. zawierać miejsce zamieszkania pracownika, które może być wspólne dla kilku pracowników. Dzięki takiemu rozwiązaniu zostanie utworzona relacja pomiędzy tabelami. Istnieje kilka rodzajów relacji.

Relacja „jeden do wielu”

Relacja „jeden do wielu” polega na tym, że do jednego wiersza w tabeli może być przypisanych kilka wierszy z innej tabeli. Przykładowo jeden klient może mieć kilka zamówień. Klucz obcy znajduje się w tabeli z zamówieniem i ma taką samą wartość jak klucz podstawowy z tabeli klienta. Po kluczu obcym można odczytać więcej danych na temat klienta, który złożył zamówienie, dzięki czemu nie trzeba do każdego zamówienia zapisywać tych samych danych pojedynczego klienta. Taka operacja odczytu dodatkowych danych z drugiej tabeli nazywana jest „joinem”. Na listingu 4.2 przedstawiono przykładową relację „jeden do wielu” dla podejścia *Code First*, a na rysunku 4.2 zaprezentowano tę samą relację w podejściu *Model First*.

Listing 4.2. Przykład relacji „jeden do wielu” w podejściu *Code First*

```
public class Ogloszenie
{
    public Ogloszenie()
    {
    }
    public int Id { get; set; }
    public string Tresc { get; set; }
    public System.DateTime DataDodania { get; set; }

    public string UzytkownikId { get; set; }

    public Uzytkownik Uzytkownik { get; set; }
}

public class Uzytkownik
{
    public Uzytkownik()
    {
        this.Ogloszenie = new HashSet<Ogloszenie>();
    }
    public int Id { get; set; }
    public string Imie { get; set; }
    public string Nazwisko { get; set; }

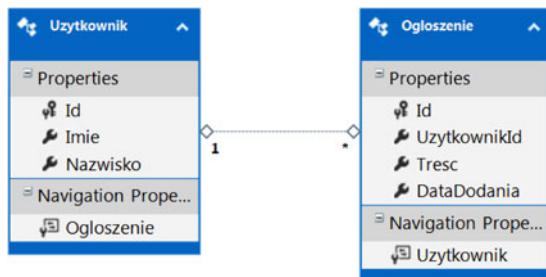
    public virtual ICollection<Ogloszenie> Ogloszenie { get; private
        set; }
}
```

Relacja „jeden do jednego”

Relacja „jeden do jednego” występuje, gdy jeden wiersz z pierwszej tabeli jest powiązany z tylko jednym rekordem z drugiej tabeli. Taka relacja może mieć sens np. wtedy, gdy chcesz przechowywać dodatkowe informacje na temat użytkownika, jednak nie chcesz, aby były one w tej samej tabeli, lub gdy użytkownik może pełnić tylko jedną funkcję w firmie. Na listingu 4.3 przedstawiono przykładową relację „jeden do jednego” dla podejścia *Code First*, a na rysunku 4.3 zaprezentowano tę samą relację w podejściu *Model First*.

Rysunek 4.2.

Przykład relacji „jeden do wielu” w podejściu Model First

**Listing 4.3.** Przykład relacji „jeden do jednego” w podejściu Code First

```

public class Uzytkownik
{
    public Uzytkownik()
    {
    }
    public int Id { get; set; }
    public string Imie { get; set; }
    public string Nazwisko { get; set; }

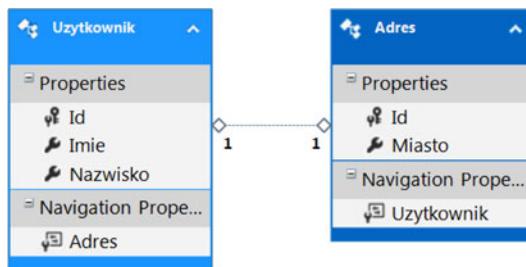
    [Required]
    public virtual Adres Adres { get; private set; }
}

public class Adres
{
    public Adres()
    {
    }
    [Key, ForeignKey(..Uzytkownik")]
    public int Id { get; set; }
    public string Miasto { get; set; }

    public virtual Uzytkownik Uzytkownik { get; set; }
}
  
```

Rysunek 4.3.

Przykład relacji „jeden do jednego” w podejściu Model First

**Relacja „wiele do wielu”**

Relacja „wiele do wielu” występuje, gdy jeden wiersz z pierwszej tabeli może być powiązany z wieloma wierszami z drugiej tabeli, a jeden wiersz z drugiej tabeli może być powiązany z kilkoma wierszami z pierwszej tabeli. Przykładowo pracownik może

mieć przypisanych wiele adresów, a jeden adres może być przypisany do wielu pracowników. Oczywiście nie chodzi tu o to, że w tabeli są zapisane te same dane, tylko o to, że klucz obcy pokazuje na ten sam wiersz w tabeli. Relacje „wiele do wielu” rozkłada się na dwie relacje „jeden do wielu” i dodaje trzecią tabelę pomiędzy tabelami, które mają relację „wiele do wielu”. Proces taki zabezpiecza przed redundancją danych (powtarzaniem tych samych danych). Nowa, środkowa tabela ma dwa klucze obce: jeden do tabeli pierwszej, a drugi do drugiej. W podejściu obiektowym w EF możliwe jest tworzenie relacji „wiele do wielu” bez dodatkowej klasy dla pośredniej tabeli. EF automatycznie przetłumaczy takie powiązanie i w bazie danych pojawi się trzecia tabela. Tabela pośrednia będzie niewidoczna z punktu widzenia kodu, jednak będzie się znajdowała w bazie danych. Na listingach przedstawiono przykładową relację „wiele do wielu” dla podejścia *Code First* bez dodatkowej tabeli (listing 4.4) i z dodatkową tabelą (listing 4.5), a na rysunkach zaprezentowano tę samą relację w podejściu *Model First* bez dodatkowej tabeli (rysunek 4.4) i z dodatkową tabelą (rysunek 4.5).

Listing 4.4. Przykład relacji „wiele do wielu” w podejściu *Code First* bez dodatkowej tabeli

```
public class Uzytkownik
{
    public Uzytkownik()
    {
        this.Wiadomosc = new HashSet<Wiadomosc>();
    }
    public int Id { get; set; }
    public string Imie { get; set; }

    public string Nazwisko { get; set; }

    public virtual ICollection<Wiadomosc> Wiadomosc { get; private
        set; }
}

public class Wiadomosc
{
    public Wiadomosc ()
    {
        this.Uzytkownik = new HashSet<Uzytkownik>();
    }
    public int Id { get; set; }
    public string Tresc { get; set; }

    public virtual ICollection<Uzytkownik> Uzytkownik { get; private
        set; }
}
```

Listing 4.5. Przykład relacji „wiele do wielu” w podejściu *Code First* z dodatkową tabelą

```
public class Uzytkownik
{
    public Uzytkownik()
    {
        this.Wiadomosc = new HashSet<Wiadomosc>();
    }
    public int Id { get; set; }
    public string Imie { get; set; }
```

```

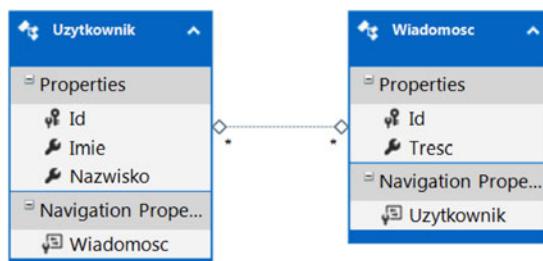
public string Nazwisko { get; set; }
public virtual ICollection<Uzytkownik_Wiadomosc>
    ↪Uzytkownik_Wiadomosc { get; private set; }
}

public class Uzytkownik_Wiadomosc
{
    public int Id { get; set; }
    public string UzytkownikId { get; set; }
    public int WiadomoscId { get; set; }
    public virtual Uzytkownik Uzytkownik { get; set; }
    public virtual Wiadomosc Wiadomosc { get; set; }
}

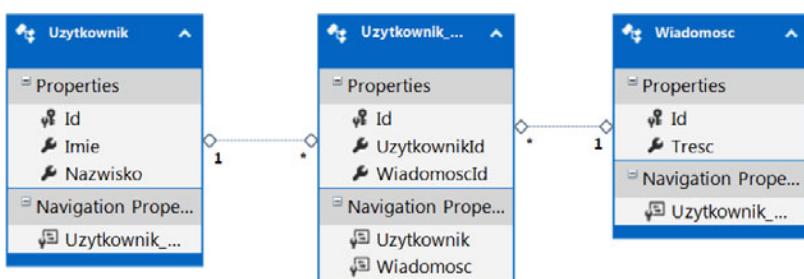
public class Wiadomosc
{
    public Wiadomosc()
    {
        this.Uzytkownik_Wiadomosc = new
            ↪HashSet<Uzytkownik_Wiadomosc>();
    }
    public int Id { get; set; }
    public string Tresc { get; set; }

    public virtual ICollection<Uzytkownik_Wiadomosc>
        ↪Uzytkownik_Wiadomosc { get; private set; }
}

```



Rysunek 4.4. Przykład relacji „wiele do wielu” w podejściu Model First bez dodatkowej tabeli



Rysunek 4.5. Przykład relacji „wiele do wielu” w podejściu Model First z dodatkową tabelą

Relacje opcjonalne

Relacje opcjonalne oznaczają, że dana relacja może, ale nie musi wystąpić. Duże znanie ma koniec, na którym znajduje się relacja opcjonalna. Przykładowo klient może, ale nie musi posiadać dodatkowych danych, dlatego relacja opcjonalna musi być po stronie tabeli klienta, ponieważ dane dodatkowe zawsze będą powiązane z klientem. Nie może być danych dodatkowych bez klienta, natomiast klient może być bez danych dodatkowych. Taka relacja to relacja 0..1 do 1.

Podobnie wygląda sytuacja z relacją 0..1 do wielu. W przypadku takiej relacji może istnieć wiele ogłoszeń, ale ogłoszenie niekoniecznie musi być powiązane z użytkownikiem.

Obiekty DbContext i DbSet

DbContext i DbSet

DbContext to obiekt odpowiedzialny za pracę z danymi. DbContext zajmuje się odczytem danych z bazy danych, śledzeniem stanu obiektów, zarządzaniem i powiązaniami pomiędzy danymi w pamięci, tłumaczeniem danych relacyjnych na obiektowe i zapisem danych do bazy. W klasie DbContext znajdują się właściwości DbSet, które reprezentują kolekcje określonych klas (tabel) znajdujących się w obiekcie kontekstu (listing 4.6).

Listing 4.6. Przykładowa klasa DbContext

```
public class ProduktContext : DbContext
{
    public DbSet<Kategoria> Kategorie { get; set; }
    public DbSet<Produkt> Produkty { get; set; }
}
```

Obiekt kontekstu jest przechowywany w pamięci do czasu, aż zostanie usunięty przez *Garbage Collector* lub zostanie wywołana metoda `Dispose()` na jego instancji. W aplikacji internetowej obiekt DbContext powinien być tworzony dla każdego żądania HTTP. Czas życia takiego obiektu powinien być jak najkrótszy, ponieważ nie zostanie on ponownie użyty.

Metody Attach i Detach

Dane pobierane z bazy są automatycznie przypisywane do kontekstu. Wyjątkiem są tutaj zapytania `AsNoTracking()`, których wynik nie jest zapisywany do kontekstu, więc dane nie są śledzone (możliwe jest przyłączenie takich danych za pomocą metod). Ponieważ DbContext przechowuje dane w postaci obiektowej, możliwe jest odłączenie obiektu od kontekstu lub przyłączenie obiektu pochodzącego z innego źródła, innego kontekstu bądź

obiektu wcześniej odłączonego od kontekstu (również z innej instancji kontekstu). Obiekty odłączone od kontekstu nie mają śledzonego stanu przez DbContext i nie mogą zostać zapisane do bazy danych.

Istnieją metody pozwalające na przyłączenie obiektu do kontekstu:

- ◆ `AddObject()` dodaje i ustawia stan obiektu na `Added` (dodany). Dla obiektów ze stanem `Added` klucz podstawowy (PK) nie musi mieć unikalnej wartości, ponieważ podczas zapisu i tak zostanie wygenerowany przez bazę danych.
- ◆ `Attach()` dodaje do kontekstu i ustawia stan na `Unchanged` (niezmieniony). Wartość PK musi być unikalna, w przeciwnym wypadku EF wywoła wyjątek.
- ◆ `AttachTo()` działa identycznie jak `Attach()`, jednak zamiast wywoływać ją dla obiektu kontekstu, to nazwę kontekstu podaje się jako parametr.

Za pomocą metody `Detach()` można odłączyć obiekt od kontekstu, jednak niesie to za sobą pewne konsekwencje. Jeśli obiekt posiadał relacje do innych obiektów, to powiązane obiekty nie zostaną usunięte z kontekstu. Stan obiektu nie jest śledzony, odłączenie nie wpływa na dane w bazie danych, a jeśli istnieje, nie zostaje wywołane kaskadowe usuwanie w trakcie wykonywania operacji `Detach()`:

```
context.Detach(zamowienie.Szczegoly.First());
```

Relacje poprzez klucz FK a relacje niezależne (obiektowe)

Relacje „jeden do jednego” zawsze są relacjami poprzez klucz (FK), natomiast relacje „wiele do wielu” zawsze są relacjami niezależnymi (ang. *Independent*). Dla relacji „jeden do wielu” można zastosować relację FK lub niezależną.

Relacje poprzez klucz obcy — FK Association

Relacja FK (ang. *FK Association*) polega na dodaniu klucza obcego do klasy odwołującej tabeli. Relacje pomiędzy klasami polegają na kluczu obcym, tak samo jak w bazie danych (listing 4.7). Takie rozwiązanie nie jest rozwiązaniem obiektowym, lecz relacyjnym. Natomiast klasy są częścią programowania obiektowego. Takie rozwiązanie ma swoje plusy i minusy.

Plusy relacji FK:

- ◆ znając wartość klucza, można dodać dane do sąsiedniej tabeli, nie trzeba posiadać wczytanego całego obiektu z bazy danych;
- ◆ podejście bardziej intuicyjne, identyczne jak w bazie danych i kodzie SQL.

Minusy relacji FK:

- ♦ podejście relacyjne, a nie obiektowe, w środowisku obiektowym;
- ♦ posiadanie dwóch relacji (zarówno FK, jak i relacji niezależnej) może powodować problem z synchronizacją pomiędzy tymi dwoma rodzajami relacji.

Listing 4.7. Przykład relacji FK w EF

```
public class Ogloszenie
{
    public Ogloszenie()
    {
    }
    public int Id { get; set; }
    public int UzytkownikId { get; set; }
    public Uzytkownik Uzytkownik { get; set; }
}

public class Uzytkownik
{
    public Uzytkownik()
    {
        this.Ogloszenie = new HashSet<Ogloszenie>();
    }
    public int Id { get; set; }
    public string Imie { get; set; }
    public string Nazwisko { get; set; }

    public virtual ICollection<Ogloszenie> Ogloszenie { get; private
        set; }
}
```

Relacje niezależne — Independent Association

Relacja niezależna (ang. *Independent Association*) polega na tym, że nie dodaje się do klasy kluczy obcych, tylko obiekt klasy, z którą występuje relacja (listing 4.8). Konsekwencją takiego podejścia jest konieczność posiadania całego obiektu w pamięci — bez tego EF nie mógłby zamodelować relacji, ponieważ nie opiera się ona na FK.

Plusy relacji niezależnej:

- ♦ istnieje dostęp do wszystkich danych z sąsiedniej tabeli, a nie tylko do klucza, ponieważ w pamięci znajduje się cały obiekt;
- ♦ obiektowe podejście.

Minusy relacji niezależnej:

- ♦ skomplikowane tłumaczenie relacji pomiędzy obiektami na relacje w bazie danych;
- ♦ mniej intuicyjne niż relacje FK;
- ♦ jeśli tworzy się relacje, potrzebne są obiekty dla obu końców relacji (w FK wystarczyło podać wartość klucza obcego);

- ◆ nie ma stanu modified — za każdym razem relacja jest usuwana i dodawana jest nowa;
- ◆ mniej naturalne z punktu widzenia baz danych i języka SQL, trudniejsze do wychwycenia błędów.

Listing 4.8. Przykład relacji niezależnej w EF

```

public class Ogloszenie
{
    public Ogloszenie()
    {
    }
    public int Id { get; set; }

    public Uzytkownik Uzytkownik { get; set; }
}

public class Uzytkownik
{
    public Uzytkownik()
    {
        this.Ogloszenie = new HashSet<Ogloszenie>();
    }
    public int Id { get; set; }
    public string Imie { get; set; }
    public string Nazwisko { get; set; }
    public virtual ICollection<Ogloszenie> Ogloszenie { get; private
        set; }
}

```

Odpytywanie bazy danych za pomocą EF i LINQ

Wczytywanie zachłanne — Eager Loading

Wczytywanie zachłanne (ang. *Eager Loading*) to proces, w którym zapytanie o jeden typ tabeli pobiera również dane z tabeli, które są z nią powiązane (posiadają relacje). Użytkownik posiada wiele ogłoszeń. Jeśli pobiera się użytkownika, to pobierane są również jego ogłoszenia. Aby użyć *Eager Loading*, gdy włączone jest *Lazy Loading*, należy wywołać metodę `Include()` w zapytaniu LINQ (listing 4.9).

Listing 4.9. Przykład zapytania Eager Loading

```

using (var context = new ApplicationContext())
{
    var uzytkownicy = context.Uzytkownik
        .Include(b => b.Ogloszenia)
        .ToList();
}

```

Wczytywanie leniwe — Lazy Loading

Wczytywanie leniwe (ang. *Lazy Loading*) pozwala na opóźniony odczyt danych. Nie ma konieczności pobierania danych ogłoszeń, jeśli chce się pobrać tylko dane użytkownika, pomimo że w klasie `Uzytkownik` znajduje się kolekcja ogłoszeń. Gdy używa się *Lazy Loading*, podczas pobierania danych do klasy użytkownika nie są ładowane jego ogłoszenia (listingi 4.10 i 4.11). *Lazy Loading* wymaga oznaczenia właściwości relacji jako wirtualnej (`virtual`). Aby było możliwe leniwe ładowanie, tworzone są dynamiczne puste obiekty proxy, które nadpisują pola wirtualne właściwości. Jeśli użytkownik próbuje się odwołać do obiektu proxy po raz pierwszy (chce wyświetlić również informacje o ogłoszeniach użytkownika), to tworzona jest instancja prawdziwego obiektu (kolekcji ogłoszeń) w miejsce pustego obiektu proxy.

Listing 4.10. Przykład klasy *Lazy Loading*

```
public class Uzytkownik
{
    public Uzytkownik()
    {
        this.Ogloszenia = new HashSet<Ogloszenie>();
    }

    public string Imie { get; set; }
    public string Nazwisko { get; set; }

    public virtual ICollection<Ogloszenie> Ogloszenia { get; private
        set; }
}
```

Listing 4.11. Przykład zapytania *Lazy Loading*

```
using (var context = new AppContext())
{
    var uzytkownicy = context.Uzytkownik.Take(10);
```

Podczas korzystania z Web API wymagana jest serializacja danych (zapis w formacie JSON lub XML). Aby serializacja była możliwa, konieczne jest wyłączenie *Lazy Loading*, ponieważ powoduje to ładowanie dużej ilości niepotrzebnych danych, a nawet zapętlenie. *Lazy Loading* można wyłączyć poprzez usunięcie modyfikatora `virtual` przed właściwością relacji lub globalnie w klasie z kontekstem, co zostało zaprezentowane w poniższym kodzie:

```
public AppContext()
{
    this.Configuration.LazyLoadingEnabled = false;
}
```

Zalety *Lazy Loading*:

- ♦ szybsze ładowanie aplikacji,
- ♦ mniejsze zużycie pamięci poprzez wczytywanie danych na życzenie,

- ◆ zmniejszenie liczby zapytań do bazy danych.

Wady *Lazy Loading*:

- ◆ konieczność sprawdzania, czy *Lazy Loading* jest włączone, co skutkuje nieznacznie mniejszą wydajnością, gdyż dane i tak muszą zostać wczytane.

Jawne ładowanie — **Explicit Loading**

Jeśli chcemy, aby przy wyłączonym trybie *Lazy Loading* zapytanie zachowywało się tak jak w trybie *Lazy Loading*, trzeba skorzystać z zapytania *Explicit Loading*. Aby wykonać *Explicit Loading*, należy wywołać metodę `Load()` na powiązanej relacją tabeli (listing 4.12).

Listing 4.12. Przykład *Explicit Loading*

```
using (var context = new ApplicationContext())
{
    var użytkownik = context.Użytkownik.Find(2);
    context.Entry(użytkownik).Collection(p => p.Ogłoszenia).Load();
}
```

Problem N+1

Problem *N+1* występuje, gdy nie korzysta się z *Lazy Loading* i chce się pobrać w jednym zapytaniu do bazy listę obiektów, które posiadają relacje z innymi obiektami (listing 4.13). Następnie dla każdego pobranego obiektu jest wykonywane osobne zapytanie pobierające dodatkowe dane z tabel, które są powiązane (posiadają relacje). Problem występuje w zapytaniach bez wykorzystania operacji `join`. Przykładowo każdy klient (wiersz w tabeli *klient*) ma przypisany jeden adres (wiersz z tabeli *adres*). Przy pobieraniu listy klientów, gdy każda klasa klienta ma pole *adres*, wywołane zostaje zapytanie zwracające listę klientów (bez adresów), a potem automatycznie wywoływanie są dodatkowo osobne zapytania o adres. Każde zapytanie jest wywoływanie osobno, a więc będzie tyle zapytań do bazy danych, ile zostanie zwróconych klientów. W *Lazy Loading Problem N+1* wystąpi dopiero wtedy, gdy zajdzie potrzeba pobrania adresów. Jeśli tabela posiada *index*, dane zostaną bardzo szybko zwrócone — i takie podejście jest prawidłowe (szybsze niż operacja `join`), jednak gdy danych jest dużo i trzeba wykonać bardzo dużo pojedynczych zapytań do bazy, operacja `join` powinna być szybsza (listing 4.14). Najlepszym wyjściem jest napisanie dwóch zapytań i sprawdzenie, które jest szybsze w danym przypadku i w jakim stopniu obciążą bazę danych.

Listing 4.13. Przykład kodu prezentującego Problem *N+1*

```
using (var context = new ApplicationContext())
{
    foreach (var użytkownik in context.Użytkownik)
    {
        foreach (var ogłoszenie in użytkownik.Ogłoszenia)
        {
            Console.WriteLine("{0}: {1}", użytkownik.Imię,
                ogłoszenie.Tytuł);
```

```
        }  
    }  
}
```

Listing 4.14. Przykład kodu korzystającego z operacji join (metoda *Include*) bez Problem N+1

```
using (var context = new AppContext())  
{  
    foreach (var użytkownik in  
            ↪context.Uzytkownik.Include("Ogłoszenia"))  
    {  
        foreach (var ogłoszenie in użytkownik.Ogłoszenia)  
        {  
            Console.WriteLine("{0}: {1}", użytkownik.Imie,  
                ↪ogłoszenie.Tytuł);  
        }  
    }  
}
```

Metoda **AsNoTracking()**

Metoda rozszerzająca `AsNoTracking()` zwraca z zapytania dane, które nie będą śledzone przez EF ani nie będą przechowywane w pamięci obiektu `DbContext`. Zastosowanie `AsNoTracking()` skutkuje około dwukrotnie większą wydajnością przy odczytzie. Jeśli dane mogą lub mają zostać zmodyfikowane, nie powinno się korzystać z tej metody. EF będzie wówczas śledził zmiany w obiekcie, co umożliwi edycję lub zapis danych do bazy (listing 4.15).

Listing 4.15. Przykładowy kod wykorzystujący metodę *AsNoTracking()*

```
using (var context = new WebAppContext())  
{  
    var dane = context.Tabela  
        ↪.Where(d => d.Nazwa.Contains(".NET"))  
        ↪.AsNoTracking()  
        ↪.ToList();  
}
```

Odroczone i natychmiastowe wykonanie

Natychmiastowe (ang. *Immediate*) wykonanie zapytania pobiera całość danych od razu podczas wykonywania zapytania. Odroczone (ang. *Deferred*) wykonanie zapytania korzysta z iteratorów (tabela 4.1) i `yield return`, a więc musi zwracać typ danych, który dziedziczy po `IEnumerable`, i zwraca dane pojedynczo w momencie, kiedy są potrzebne. Domyślnie używane jest odroczone wykonanie. Aby wywołać natychmiastowe wykonanie, należy użyć jednej z następujących metod: `ToListAsync()`, `ToDictionary()` lub `ToArray()`. Wszystkie zapytania zwracające pojedyńczą wartość są wykonywane natychmiastowo, np. `Average`, `Count`, `First` lub `Sum`, ponieważ nie ma możliwości użycia opóźnionego wykonania, jeśli jest tylko jeden element w kolekcji.

Tabela 4.1. Lista operatorów, które posiadają zaimplementowane iteratory

Metoda	Iterator
Cast	CastIterator
Concat	ConcatIterator
DefaultIfEmpty	DefaultIfEmptyIterator
Distinct	DistinctIterator
Except	ExceptIterator
GroupJoin	GroupJoinIterator
Intersect	IntersectIterator
Join	JoinIterator
OfType	OfTypeIterator
Range	RangeIterator
Repeat	RepeatIterator
Reverse	ReverseIterator
Select	SelectIterator
SelectMany	SelectManyIterator
Skip	SkipIterator
SkipWhile	SkipWhileIterator
Take	TakeIterator
TakeWhile	TakeWhileIterator
Union	UnionIterator
Where	WhereIterator
Zip	ZipIterator

Entity SQL

Entity SQL to język zapytań bazujący na SQL (alternatywa dla *LINQ to Entities*). Pozwala na pisanie zapytań w języku SQL, jednak w treści zapytania zamiast nazw tabel podaje się nazwy klas z aplikacji. *Entity SQL* powinien być używany tylko w momencie, gdy zapytanie pisane za pomocą LINQ jest zbyt skomplikowane. Zapytania *Entity SQL* nie są kompilowane, lecz konstruowane są w czasie działania programu. Zapytania LINQ są kompilowane i typowane (zwiększa się bezpieczeństwo i szybkość), dlatego powinny być używane, jeśli tylko jest to możliwe. Aby wykonać zapytanie *Entity SQL*, należy uzyskać `ObjectContext` z obiektu `DbContext` i skorzystać z generycznej metody `Object` → `Query<T>` (listing 4.16)¹.

Listing 4.16. Przykład zapytania Entity SQL

```
using(ApplicationDbContext context = new ApplicationDbContext())
{
    var adapter = (IObjectContextAdapter)context;
    var objectContext = adapter.ObjectContext;
```

¹ <http://msdn.microsoft.com/en-us/library/bb399554.aspx>

```
string esqlQuery = @"SELECT VALUE Ogloszenie FROM
    ↪Models.Ogloszenia as Ogloszenie
    ↪WHERE Ogloszenie.Id > @parametrId";

ObjectQuery<Ogloszenie> query = new
    ↪ObjectQuery<Ogloszenie>(esqlQuery, objectContext,
    ↪MergeOption.NoTracking);

query.Parameters.Add(new ObjectParameter("parametrId", 1));

return query;
}
```

Bezpośrednie zapytania SQL do bazy (Direct/RAW SQL) i procedury składowane w EF

Zapytania *Direct/RAW SQL* powinny być stosowane tylko wtedy, gdy nie ma możliwości skorzystania z zapytań *LINQ to Entities* lub *Entity SQL*. W zapytaniu SQL podaje się nazwy tabel w bazie danych, a nie nazwy klas, jak to było w *Entity SQL*. W *Entity SQL* operuje się na typach pochodzących z aplikacji, co wpływa na bezpieczeństwo, ponieważ typy danych są znane i walidowane. Dla zapytań *RAW SQL*, czyli składających się z czystego kodu SQL, występuje niebezpieczeństwo ataku SQL Injection. Aby pobrać dane za pomocą języka SQL, należy użyć metody `SqlQuery()`, a w celu wysłania zapytania do bazy niezwracającego żadnych wartości korzysta się z `SqlCommand()`.

Przykład zapytania zwracającego typ klasy POCO:

```
using (var context = new ApplicationContext())
{
    var blogs = context.Ogloszenia.SqlQuery("SELECT * FROM dbo.Ogloszenia")
        .ToList();
}
```

Przykład wywołania procedury składowanej z parametrem:

```
using (var context = new ApplicationContext())
{
    var ogloszenieId = 1;
    var blogs = context.Ogloszenia.SqlQuery("dbo.GetOgloszenieById
        ↪@p0", ogloszenieId).Single();
}
```

Przykład zapytania zwracającego typ, który nie jest klasą POCO:

```
using (var context = new ApplicationContext())
{
    var tytuly = context.Database.SqlQuery<string>("SELECT Tytuł FROM
        ↪dbo.Ogloszenia").ToList();
}
```

Przykład zapytania SQL niezwracającego wartości:

```
using (var context = new ApplicationContext())
{
    context.Database.SqlCommand("UPDATE dbo.Ogloszenia SET Tytuł =
        ↳'Jakis tytuł' WHERE Id = 1");
}
```

Transakcje w EF

Transakcja to traktowanie kilku operacji na bazie danych jak jednej. Jeśli choć jedno zapytanie nie zostanie wykonane, to reszta zapytań także nie może zostać wykonana. Jeśli błąd wystąpił w ostatniej operacji, to wcześniej wykonane operacje należy wycofać, czyli przywrócić dane do stanu, w jakim znajdowały się przed rozpoczęciem wykonywania transakcji. Transakcje powinny być wykonywane, jeśli kolejne operacje zależą od wcześniejszych operacji lub gdy nie chce się, aby rezultat pojedynczej operacji (tylko części z całej transakcji) był widoczny dla innych użytkowników bazy danych. W przypadku transakcji wszystkie operacje zostaną wykonane jak pojedyncza operacja. Transakcja może działać na kilku źródłach danych — dopiero gdy wszystkie operacje zakończą się powodzeniem, transakcja będzie zaakceptowana i dane zostaną zapisane. Aby skorzystać z transakcji, należy utworzyć nowy obiekt TransactionScope przy użyciu bloku using (listing 4.17), następnie utworzyć obiekty kontekstu i wykonać operacje powiązane z transakcją. Na koniec wywołuje się metodę Complete(), która oznacza, że wszystkie operacje się powiodły i cała transakcja zakończyła się sukcesem. Jeśli którakolwiek operacja się nie powiedzie, metoda Complete() nie zostanie wywołana, co będzie oznaczać, że trzeba przywrócić dane do stanu oryginalnego (odwrócić już wykonane operacje)².

Listing 4.17. Przykład użycia TransactionScope

```
using (var scope = new TransactionScope(TransactionScopeOption.Required))
{
    using (var conn = new SqlConnection("..."))
    {
        conn.Open();

        var sqlCommand = new SqlCommand();
        sqlCommand.Connection = conn;
        sqlCommand.CommandText = @"UPDATE Blogs SET Rating = 5" +
            ↳" WHERE Name LIKE '%Entity Framework%';

        sqlCommand.ExecuteNonQuery();

        using (var context = new ApplicationContext(conn, contextOwnsConnection:
            ↳false))
        {
            var query = context.Ogloszenia.Where(p => p.Id > 7);
            foreach (var ogl in query)
            {

```

² <http://msdn.microsoft.com/en-us/data/dn456843.aspx>

```
        ogl.Tytul += "Nazwa firmy";
    }
    context.SaveChanges();
}
}
scope.Complete();
}
```

Śledzenie zmian

Obiekt przechowywany w DbContext może być w jednym ze stanów:

- ◆ Added — nowy obiekt dodany do kontekstu;
- ◆ Modified — obiekt został zmieniony od czasu, gdy został pobrany z bazy danych;
- ◆ Unchanged — obiekt nie został zmieniony od czasu pobrania z bazy danych;
- ◆ Detached — obiekt odłączony od kontekstu, a zmiany w takim obiekcie nie będą miały wpływu na zmiany w bazie danych;
- ◆ Deleted — obiekt oznaczony w kontekście jako obiekt do usunięcia — nie został jeszcze usunięty z bazy danych, ale po wywołaniu metody SaveChanges() zostanie usunięty z bazy danych.

Migawkowe śledzenie zmian — Snapshot Change Tracking

Aby zapisać zmiany wprowadzone w obiektach do bazy danych, należy wywołać metodę SaveChanges() dla kontekstu. Jeśli nie wywoła się tej metody, zmiany będą widoczne tylko w obiekcie kontekstu, ale nie w bazie danych. Aby dostać się do informacji o stanie obiektu, należy wywołać metodę Entry(obiekt).State. Każdy obiekt posiada dwie wartości: oryginalną (pobraną z bazy danych) oraz aktualną. Po wywołaniu metody SaveChanges() uruchamiana jest automatycznie metoda DetectChanges(), która poprzez porównanie z wartością początkową sprawdza, które dane zostały zmienione. Jeśli dane zostały zmienione, zostaje wygenerowane i uruchomione zapytanie, które wprowadzi zmiany w bazie danych. Takie podejście do śledzenia zmian nazywane jest migawkowym śledzeniem zmian (ang. *Snapshot Change Tracking*) i jest domyślnie wykorzystywane przez najnowszą wersję EF 6.

Dynamiczne śledzenie zmian — Dynamic Change Tracking (proxy)

Dynamiczne śledzenie zmian (ang. *Dynamic Change Tracking*) to podejście polegające na informowaniu kontekstu o zmianach przy każdej operacji na danych. Aby możliwe było korzystanie z trybu *Dynamic Change Tracking*, konieczne jest oznaczenie wszystkich

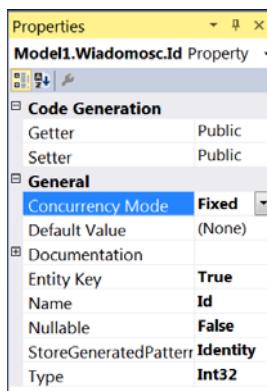
właściwości w klasie jako `virtual`, ponieważ wykorzystywane są klasy proxy. We wcześniejszych wersjach EF była to domyślna opcja, jednak przy bardziej skomplikowanych scenariuszach powodowała problemy z wydajnością. Dlatego w najnowszej wersji EF domyślnym sposobem jest *Snapshot Change Tracking*. Problemy wydajności w podejściu *Dynamic* pojawiają się, gdy dane zmieniają się kilkakrotnie, ale ostatecznie ich wartość jest taka sama jak po odczytce z bazy danych. Dane zostaną oznaczone jako zmodyfikowane i będą zapisywane do bazy danych, chociaż ich wartość się nie zmieniła. W podejściu *Snapshot Change Tracking* została porównana tylko wartość z oryginalną bez wykonywania zapisu do bazy danych. Ponieważ dane są przechowywane w pamięci, operacja sprawdzenia wartości oryginalnej z aktualną jest bardzo szybka w porównaniu do operacji wejścia-wyjścia, czyli np. zapisu danych do bazy danych.

Zarządzanie operacjami współbieżnymi

Od momentu pobrania danych z bazy danych i jednocześniej edycji do czasu zapisu ktoś inny może zmienić pobrane wartości w bazie. Domyślnie EF nie śledzi takich zmian i nadpisuje zmienione dane. Aby śledzić dane, należy ustawić `ConcurrencyMode="Fixed"` dla podejścia *Model First* (rysunek 4.6) lub dodać atrybut `[ConcurrencyCheck]` w klasie w podejściu *Code First* (listing 4.18). Atrybut ustawia się osobno dla każdego pola lub klasy. Aby śledzić zmiany ogólnie dla całej klasy, dodaje się dodatkowe pole `Timestamp`, które określa, kiedy nastąpiły ostatnie zmiany³.

Rysunek 4.6.

Przykład
`ConcurrencyCheck`
 dla *Model First*



Listing 4.18. Przykład `ConcurrencyCheck` dla *Code First*

```
[ConcurrencyCheck]
public int ParentId { get; set; }
// i/lub
[Timestamp]
public byte[] RowTimeStamp { get; set; }
```

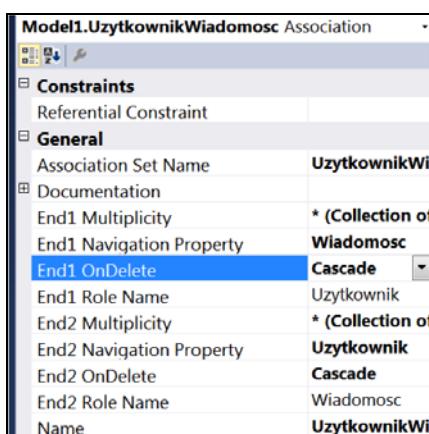
³ <http://www.asp.net/mvc/tutorials/getting-started-with-ef-using-mvc/handling-concurrency-with-the-entity-framework-in-an-asp-net-mvc-application>

Kaskadowe usuwanie

— Cascade Delete

Cascade Delete to operacja polegająca na kaskadowym usuwaniu elementów. Oznacza to, że razem z obiektem usuwa się obiekty, które są z nim powiązane relacją. Przykładowo podczas usuwania klienta z włączoną opcją kaskadowego usuwania zostaną usunięte wszystkie powiązane z nim zamówienia. W EF dla podejścia *Model First* kaskadowe usuwanie ustawia się przy dodawaniu relacji (rysunek 4.7). W podejściu *Code First* trzeba skorzystać z narzędzia *Fluent API*.

Rysunek 4.7.
Cascade Delete
w *Model First*



Istnieją zasady, których należy przestrzegać podczas kaskadowego usuwania:

- ◆ Jeśli włączono kaskadowe usuwanie w EF, należy także zaznaczyć odpowiednią opcję **CASCADE DELETE** w bazie danych.
- ◆ Jeśli w bazie danych nie ma włączonego kaskadowego usuwania, a chcemy je wykonać, najpierw należy załadować wszystkie dane, które mają zostać usunięte. Nie jest to dobre rozwiązanie, ponieważ odpowiedzialność za usuwanie spoczywa wtedy na EF, a nie na bazie danych, więc zostaną usunięte tylko te dane, które są załadowane do pamięci. W przypadku wielokrotnego kaskadowego usuwania wystąpią problemy z synchronizacją, ponieważ może być konieczne usunięcie danych z dalszych tabel, dla których także zostało włączone kaskadowe usuwanie. Przykładowo w sytuacji kategoria -> produkt -> zamówienie będzie wymagane usunięcie danych z trzech tabel, a w pamięci będą załadowane tylko dwie tabele.

Cascade Delete jest domyślnie włączone. Aby globalnie wyłączyć *Cascade Delete*, należy użyć następującej metody w klasie kontekstu:

```
protected override void OnModelCreating(DbModelBuilder modelBuilder)
{
    modelBuilder.Conventions.Remove<OneToManyCascadeDeleteConvention>();
}
```

Aby włączyć *Cascade Delete* (jeśli jest globalnie wyłączone) dla wybranej relacji, należy skorzystać z *Fluent API*:

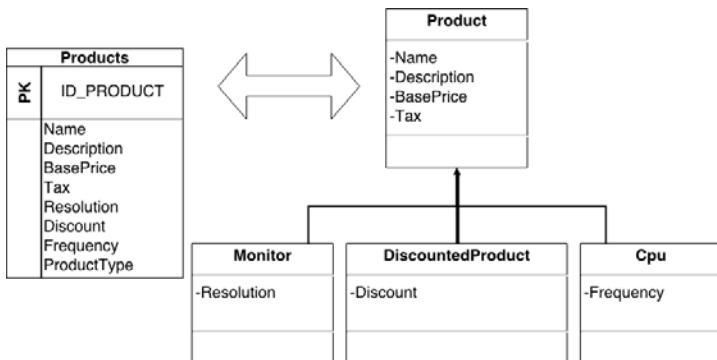
```
modelBuilder.Entity<Ogloszenie>().HasRequired(x => x.Kategoria)
    .WithMany(x => x.Ogloszenia)
    .HasForeignKey(x => x.KategoriaId)
    .WillCascadeOnDelete(true);
```

Strategie dziedziczenia w bazie danych — TPT, TPH i TPC

Z punktu widzenia programowania obiektowego możliwe jest dziedziczenie klas. W bazach relacyjnych nie ma takiego pojęcia jak dziedziczenie, dlatego EF pozwala tłumaczyć mechanizm dziedziczenia na strukturę bazy danych. EF ma różne wzorce służące do zamodelowania mechanizmu dziedziczenia w bazie danych: TPH, TPT i TPC. Strategię wybiera się w zależności od potrzeb.

TPH

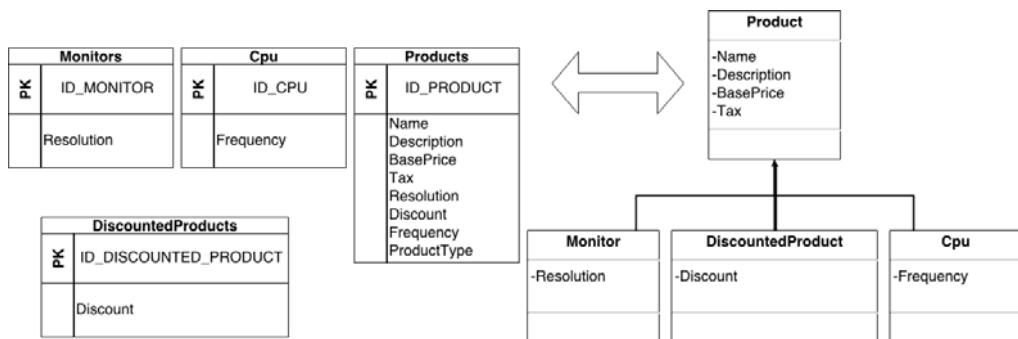
TPH (ang. *Table Per Hierarchy*) polega na utworzeniu jednej wspólnej tabeli dla wszystkich klas w hierarchii dziedziczenia. W zależności od typu niektóre wiersze w tabeli nie posiadają wartości. Dla typu Monitor puste pozostaną pola *Discount* oraz *Frequency* z klas *DiscountedProduct* i *Cpu*. W tabeli pojawia się jedno dodatkowe pole, które określa, jakiego typu jest dana tabela (*ProductType*) (rysunek 4.8).



Rysunek 4.8. Wzorzec TPH

TPT

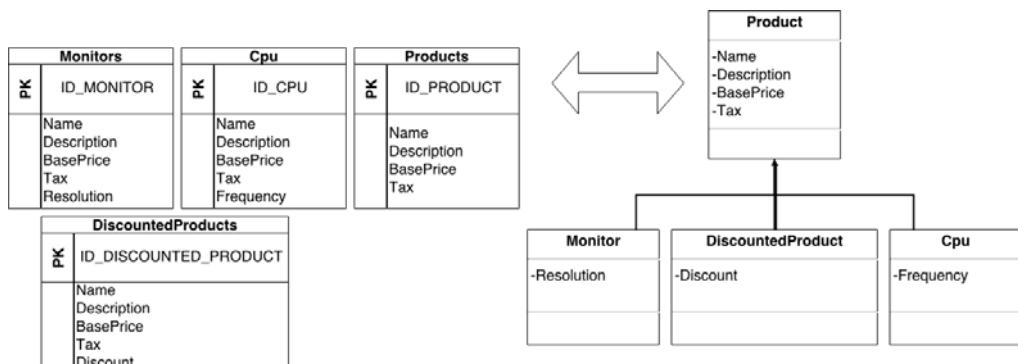
TPT (ang. *Table Per Type*) polega na utworzeniu dla każdej klasy, nawet tej abstrakcyjnej, osobnej tabeli. Struktura w bazie danych wygląda tak jak struktura obiektów, klasa bazowa posiada tabele ze swoimi polami oraz każda klasa dziedzicząca ma osobną tabelę ze swoimi polami. PK w tabelach dziedziczących odpowiadają PK w tabeli bazowej, po której dziedziczą (rysunek 4.9).



Rysunek 4.9. Wzorzec TPT

TPC

W TPC (ang. *Table Per concrete Class*) każda klasa w hierarchii dziedziczenia odpowiada osobnej tabeli w bazie danych. W podejściu tym występuje duża nadmiarowość danych, ponieważ te same dane są kopowane do wielu tabel. Skutkuje to lepszą wydajnością, ponieważ nie wymaga wykonywania operacji join do innych tabel (rysunek 4.10).



Rysunek 4.10. Wzorzec TPC

SQL Logging

SQL Logging to nowość wprowadzona w EF 6, która pozwala na proste logowanie, zapisywanie bądź wyświetlanie zapytań wysyłanych do bazy danych.

Aby pobrać informacje na temat zapytania, należy skorzystać z obiektu `context.Database.Log`.

Oto przykład użycia do wyświetlenia za pomocą mechanizmu *Trace* (wynik widoczny w oknie *Output*):

```
public bool Dodaj(Ogloszenie ogl)
{
    context.Database.Log = message => Trace.WriteLine(message);

    context.Ogloszenia.Add(ogl);
    context.SaveChanges();
    return true;
}
```

Poniżej pokazano treść wyświetlaną w oknie *Output*:

```
INSERT [dbo].[Ogloszenie]([Tresc], [Tytul], [DataDodania],
                           ↑[UzytkownikaId], [KategoriaId])
VALUES (@0, @1, @2, NULL, @3)
SELECT [Id]
FROM [dbo].[Ogloszenie]
WHERE @@ROWCOUNT > 0 AND [Id] = scope_identity()

-- @0: 'gndf' (Type = String, Size = 500)
-- @1: 'dfng' (Type = String, Size = 72)
-- @2: '2014-06-20 23:46:23' (Type = DateTime2)
-- @3: '1' (Type = Int32)

-- Executing at 2014-06-20 23:46:26 +02:00
-- Completed in 2 ms with result: SqlDataReader

Committed transaction at 2014-06-20 23:46:26 +02:00

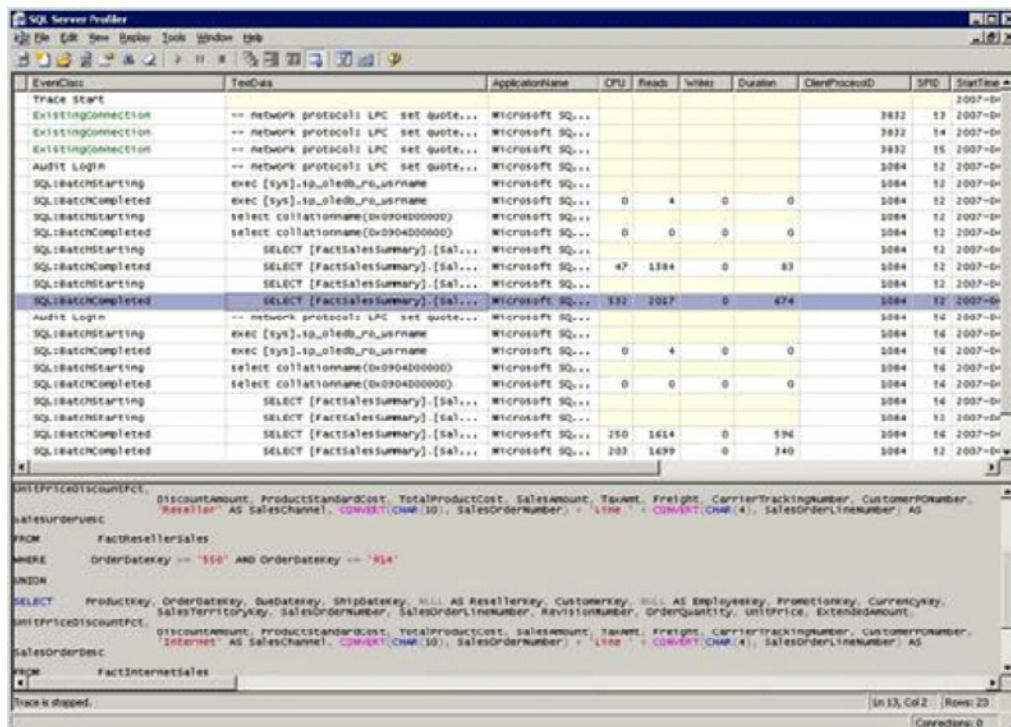
Closed connection at 2014-06-20 23:46:26 +02:00
```

Logi można dowolnie formatować za pomocą *DatabaseLogFormatter* lub powierzyć logowanie osobnemu narzędziu, jak np. NLog, co będzie zazwyczaj najlepszym rozwiązaniem.

SQL Server Profiler to aplikacja pozwalająca na przeglądanie zapytań wysyłanych do bazy danych SQL Server. Aby uruchomić aplikację, należy przejść do menu *Start/Wszystkie programy/Performance Tools* i wybrać *SQL Server Profiler* (rysunek 4.11).

Code First Fluent API i Data Annotations

Fluent API to narzędzie służące do konfiguracji właściwości oraz opisu powiązań pomiędzy klasami POCO w podejściu *Code First*. Domyslnie EF korzysta z konwencji, na podstawie których generowana jest baza danych (np. klucz podstawowy musi zawierać słowo *Id*). Jednak nie wszystko da się załatwić poprzez użycie samych konwencji.



Rysunek 4.11. Wygląd aplikacji SQL Server Profiler

Z pomocą przychodzą atrybuty *Data Annotations*. Dzięki nim można np. oznaczyć jako klucz podstawowy (PK) pole, które nie posiada w nazwie słowa Id (atrybut [Key]). Zdarzają się jednak takie sytuacje, w których nie ma możliwości rozwiązania problemu za pomocą *Data Annotations* i niezbędne staje się użycie *Fluent API*⁴.

Fluent API to bardzo rozbudowane narzędzie, które pozwala m.in. na:

- ◆ konfigurację kluczy podstawowych i obcych,
- ◆ oznaczanie właściwości jako wymaganych,
- ◆ określanie maksymalnej długości właściwości,
- ◆ ustalanie indeksów (od EF 6.1),
- ◆ ustalanie typów właściwości w bazie danych (np. nvarchar),
- ◆ mapowanie powiązań pomiędzy klasami,
- ◆ ustalanie sposobu dziedziczenia (TPT, TPH, TPC),
- ◆ mapowanie jednej klasy na kilka tabel,
- ◆ mapowanie wielu klas na jedną tabelę,

⁴ <http://msdn.microsoft.com/pl-pl/data/jj591617.aspx>

- ◆ ustalanie właściwości, które nie mają być tworzone w bazie danych, np. w bazie znajdują się dwa pola: Imię i Nazwisko, a w klasie jest jeszcze właściwość PełneNazwisko zwracająca Imię + Nazwisko. Aby PełneNazwisko nie zostało dodane do bazy danych, należy je oznaczyć jako niemapowane/ignorowane.

Aby skorzystać z *Fluent API*, należy dodać (nadpisać) metodę `OnModelCreating` w klasie z kontekstem (listing 4.19).

Listing 4.19. Nadpisywanie metody `OnModelCreating` w klasie z kontekstem

```
protected override void OnModelCreating(DbModelBuilder modelBuilder)
{
    base.OnModelCreating(modelBuilder);

    // using System.Data.Entity.ModelConfiguration.Conventions;

    // Wyłączamy konwencję, która każe zamieniać nazwę tabeli na liczbę mnoga
    // Zamiast Kategorie utworzyłyby Kategories
    modelBuilder.Conventions.Remove
        =><PluralizingTableNameConvention>();

    // Wyłączamy domyślną konwencję Cascade Delete dla powiązań

    // CascadeDelete zostanie włączone za pomocą Fluent API
    modelBuilder.Conventions.Remove
        =><OneToManyCascadeDeleteConvention>();
    // Dodajemy relację jeden do wielu pomiędzy tabelami Kategoria i Ogloszenie
    // Tworzymy relację FK i włączamy dla tej relacji CascadeDelete

    modelBuilder.Entity<Ogloszenie>()
        .IsRequired(x => x.Kategoria)
        .WithMany(x => x.Ogloszenia)
        .HasForeignKey(x => x.KategoriaId)
        .WillCascadeOnDelete(true);
}
```

Migracje

Migracje to aktualizacje struktury bazy danych wywoływanie po wprowadzeniu zmian w klasach z modelem. Migracje dzielą się na typy stratne i bezstratne. Migracje bezstratne to takie, w których nie zostają usunięte żadne dane z bazy danych. Taka migracja występuje np. podczas dodawania nowej klasy z modelem, czyli dodatkowej tabeli w bazie danych. Migracje stratne to takie, w których nie ma możliwości aktualizacji bazy danych w taki sposób, aby zachować wcześniejsze dane (np. zmiana typu pola na pole innego typu lub zmiana powiązań pomiędzy tabelami). W Entity Framework dostępne jest wbudowane narzędzie automatycznie generujące skrypt do aktualizacji.

W celu zainstalowania migracji należy w pierwszej kolejności mieć zainstalowany Entity Framework, a następnie uruchomić w oknie *Package Manager Console* komendę:

```
enable-migrations -ContextTypeName ApplicationDbContext  
    ↳-MigrationsDirectory:.MigrationsUser
```

Dodanie nowej migracji wykonuje się komendą:

```
Add-Migration -configuration  
    ↳MigrationsUser.Configuration <Migrations-Name>
```

Do uruchomienia migracji służy komenda:

```
Update-Database -configuration  
    ↳MigrationsUser.Configuration (-Verbose)
```

Aby zobaczyć zapytania, które są wykonywane w czasie aktualizacji bazy danych, należy dodać do komendy parametr *-Verbose*.

Domyślnie migracje znajdują się w folderze *Migrations*. W wersji MVC 5 zostały wprowadzone migracje dla wielu kontekstów, dlatego każda migracja musi być w osobnym folderze i posiadać osobny plik konfiguracyjny. Istnieje możliwość ręcznego edytowania lub poprawiania wygenerowanych automatycznie plików migracji. Narzędzie do migracji zapamiętuje wcześniejszy stan bazy danych i na tej podstawie generuje skrypt do migracji. Gdy struktura bazy danych zostanie zmieniona bezpośrednio w bazie danych (bez użycia migracji), migracje nie będą działać poprawnie. W takim wypadku należy poprawić wygenerowany skrypt lub usunąć całą historię migracji i uruchomić je ponownie.

Migracje są ścisłe powiązane z metodą *Seed*⁵.

Metoda Seed

Seed to metoda uruchamiana po każdej migracji bazy danych lub przy pierwszym uruchomieniu aplikacji. Dzięki takiemu rozwiązaniu po aktualizacji i usunięciu danych z bazy danych zostają wprowadzone dane startowe, przez co baza nie jest całkowicie pusta. Od pierwszego uruchomienia można testować funkcjonalność aplikacji bez konieczności ręcznego dodawania danych. Gdy zostaną wprowadzone zmiany w strukturze bazy danych, wystarczy zaktualizować te części metody, które odnoszą się do zaktualizowanej części.

Metoda *Seed* znajduje się w klasie konfiguracyjnej migracji. Jako parametr przekazuje się do niej kontekst, na którym ma operować metoda (listing 4.20). W tym przypadku zostały uruchomione automatyczne migracje, co oznacza, że po każdej aktualizacji w klasie z modelem zostaną uruchomione migracje i metoda *Seed*. Zezwolono również na migracje stratne, wymagające usunięcia wcześniejszych danych. Nazwa folderu, w którym znajdują się migracje dla kontekstu, to *MigrationsUser*. W przykładzie został dołączony kod do debugowania metody *Seed*. Ponieważ działa ona jeszcze przed uruchomieniem programu, debugowanie musi się odbywać w osobnej instancji Visual Studio.

⁵ <http://msdn.microsoft.com/en-us/data/jj591621.aspx>

Listing 4.20. Przykładowa metoda Seed

```
internal sealed class Configuration :  
    ↪DbMigrationsConfiguration<FullContext>  
{  
    public Configuration()  
    {  
        AutomaticMigrationsEnabled = true;  
        AutomaticMigrationDataLossAllowed = true;  
        MigrationsDirectory = @"MigrationsUser";  
    }  
  
    protected override void Seed(ApplicationDbContext context)  
    {  
        // Do debugowania metody Seed  
        // if(System.Diagnostics.Debugger.IsAttached == false)  
        // System.Diagnostics.Debugger.Launch();  
        SeedRoles(context);  
    }  
    private void SeedRoles(ApplicationDbContext context)  
    {  
        var roleManager = newRoleManager<Microsoft.AspNet.Identity  
            ↪.EntityFramework.IdentityRole>(new  
            ↪RoleStore<IdentityRole>());  
  
        if (!roleManager.RoleExists("Admin"))  
        {  
            var role = new  
            ↪Microsoft.AspNet.Identity.EntityFramework.IdentityRole();  
            role.Name = "Admin";  
            roleManager.Create(role);  
        }  
    }  
}
```

Rozdział 5.

ASP.NET MVC 5

ASP.NET MVC to platforma aplikacyjna firmy Microsoft służąca do budowy aplikacji internetowych opartych na wzorcu MVC. ASP.NET MVC jest alternatywą dla ASP.NET Web Forms i Web Pages. Wszystkie technologie platformy .NET są równolegle rozwijane. Web Forms i Web Pages, w przeciwieństwie do MVC, nie pozwalają na oddzielenie warstwy logiki od warstwy prezentacji, przez co nie nadają się do większych projektów.

Kolejne wersje ASP.NET MVC

ASP.NET MVC 1

Pierwsza wersja ASP.NET MVC została wydana w 2009 r. Dzięki zastosowaniu wzorca MVC i odseparowaniu warstwy logiki od warstwy prezentacji możliwe stało się stosowanie testów jednostkowych i podejścia TDD (ang. *Test Driven Development*).

ASP.NET MVC 2

Nowości w ASP.NET MVC 2¹:

- ◆ typowane HTML helpery,
- ◆ nowe sposoby walidacji danych,
- ◆ automatyczne generowanie kodu (ang. *Scaffolding*),
- ◆ grupowanie projektów w obszary (ang. *Areas*),
- ◆ asynchroniczne kontrolery,
- ◆ generowanie sekcji stron dzięki `Html.RenderSection`.

¹ [http://msdn.microsoft.com/en-us/library/dd394709\(v=vs.100\).aspx](http://msdn.microsoft.com/en-us/library/dd394709(v=vs.100).aspx)

ASP.NET MVC 3

Nowości w ASP.NET MVC 3²:

- ◆ Entity Framework 4,
- ◆ .NET Framework 4,
- ◆ Razor — nowy silnik renderujący (warstwa widoku),
- ◆ ułatwienie wstrzykiwania zależności dzięki interfejsowi `IDependencyResolver`,
- ◆ cachowanie częściowe,
- ◆ nowe typy *Action Result*,
- ◆ nowe sposoby walidacji danych.

ASP.NET MVC 4

Nowości w ASP.NET MVC 4³:

- ◆ Entity Framework 5,
- ◆ .NET Framework 4.5,
- ◆ ASP.NET Web API,
- ◆ nowy szablon projektu,
- ◆ nowy szablon dla urządzeń mobilnych,
- ◆ wsparcie dla aplikacji mobilnych,
- ◆ wsparcie dla migracji,
- ◆ minimalizacja i tworzenie paczek skryptów (ang. *Bundling and Minification*),
- ◆ OpenID, OAuth,
- ◆ Azure SDK,
- ◆ nowy pusty projekt,
- ◆ kontrolery w dowolnym folderze.

ASP.NET MVC 5

Nowości w ASP.NET MVC 5⁴:

- ◆ Entity Framework 6,
- ◆ ASP.NET Identity,

² <http://www.asp.net/mvc/mvc3>, <http://www.asp.net/whitepapers/mvc3-release-notes>,
<http://msdn.microsoft.com/pl-pl/library/wprowadzenie-do-asp-net-mvc-3-0.aspx>

³ <http://www.asp.net/mvc/mvc4>

⁴ <http://www.asp.net/mvc/mvc5>

- ♦ Twitter Bootstrap,
- ♦ filtry autentykacji (ang. *Authentication Filters*),
- ♦ nadpisywanie filtrów (ang. *Filter Overrides*),
- ♦ ASP.NET Web API 2,
- ♦ routing na podstawie atrybutów (ang. *Attribute Routing*),
- ♦ Portable ASP.NET Web API Client.

ASP.NET MVC 6 (zapowiedź)

ASP.NET MVC 6 będzie miało premierę w połowie 2015 r. Razem z MVC 6 zostanie wydany Entity Framework 7. Zarówno MVC, Web API, jak i Web Pages zostaną połączone w jedną całość pod nazwą MVC 6. MVC 6 zostanie zoptymalizowane pod rozwiązania serwerowe. Planuje się usunięcie zależności od biblioteki *System.Web* oraz niepotrzebnych, z punktu widzenia rozwiązań internetowych, bibliotek. Odchudzenie spowoduje szybsze uruchamianie aplikacji i mniejsze zużycie pamięci. Obiekt *HttpContext* tworzony dla każdego żądania HTTP ma znacznie zmniejszyć swój rozmiar. Dla „krótkich” zapytań może to stanowić bardzo duży wzrost wydajności⁵.

Konwencje w MVC

Struktura projektu

W głównym projekcie ASP.NET MVC znajdują się następujące katalogi (rysunek 5.1):

- ♦ *App_Start* — zawiera wszystkie skrypty wykonywane przed uruchomieniem aplikacji (routing, autoryzacja, filtry).
- ♦ *Content* — w tym katalogu znajdują się takie pliki jak grafika, pliki CSS itp.
- ♦ *FONTS* — zawiera pliki z czcionkami.
- ♦ *Controllers* — przechowuje klasy kontrolerów.
- ♦ *Models* — przechowuje klasy modelu.
- ♦ *Scripts* — tutaj znajdują się pliki JavaScript oraz biblioteki AJAX i JQuery.
- ♦ *Views* — w tym katalogu znajdują się katalogi dla poszczególnych widoków. Nazwy katalogów odpowiadają nazwom kontrolerów. W każdym katalogu znajdują się pliki z widokami powiązanymi z danym kontrolerem. Jest to konwencja, którą w razie potrzeby można zmienić. Oprócz katalogów powiązanych z kontrolerami znajduje się tu także folder *Shared* zawierający

⁵ <http://www.asp.net/vnext/overview/aspnet-vnext/overview>,
<http://blogs.msdn.com/b/webdev/archive/2014/05/13/asp-net-vnext-the-future-of-net-on-the-server.aspx>

Rysunek 5.1.
Struktura projektu



widoki *Master Page*, czyli wspólne dla wszystkich widoków, oraz *Partial View*, czyli widoki częściowe dostępne także dla różnych widoków. Nazwa widoku *Partial* powinna się zaczynać od znaku _.

Poza wymienionymi folderami w głównym katalogu projektu znajdują się jeszcze plik konfiguracyjny aplikacji (*Web.config*), plik konfiguracyjny zainstalowanych bibliotek (*packages.config*), plik startowy aplikacji z metodą ApplicationStart (*Global.asax*), plik favicon (mała ikona strony widoczna w zakładce przeglądarki) oraz plik *Startup.cs*, w którym uruchamiany jest OWIN.

Konwencje a ASP.NET MVC

W MVC przyjęte są pewne założenia odnośnie do tego, co i jak użytkownik zamierza zrobić, dzięki czemu użytkownik nie musi samodzielnie ustawiać każdej drobnej opcji w plikach konfiguracyjnych — takie podejście nazywane jest konwencją ponad konfiguracją (ang. *Convention over Configuration*). Jest to bardzo praktyczne rozwiązanie, ponieważ dostaje się domyślne ustawienia. Istnieje również możliwość zmian w konwencji. Konwencje w ASP.NET MVC:

- ◆ klasa `ViewModel` powinna zawierać w nazwie słowa `ViewModel`;
- ◆ domyślnie zwracany widok w kontrolerze to widok z folderu o tej samej nazwie co nazwa kontrolera i nazwie pliku z widokiem takiej jak nazwa akcji, a więc kontroler o nazwie `Account` zwraca widoki z folderu `Views/Account`, natomiast akcja `Register` kontrolera `Account` domyślnie zwraca widok o nazwie `Register`;
- ◆ klasy z modelem zawierają w nazwie słowo `Model`, np. `AccountModels`;
- ◆ klasy kontrolerów zawierają w nazwie słowo `Controller`, np. `AccountController`;
- ◆ pliki z folderu `App_Start` zawierają ustawienia startowe i w większości przypadków w nazwie mają słowo `Config`, np. `BundleConfig`.

MVC Pipeline — ścieżka wywołań, handlery i moduły

Aplikacje ASP.NET działają jako rozszerzenie ISAPI (ang. *Internet Server Application Programming Interface*) na serwerze (zazwyczaj IIS) i mają dostęp do wszystkich funkcji serwera IIS. ISAPI składa się z dwóch komponentów: rozszerzeń (ang. *ISAPI extensions*), czyli aplikacji, oraz filtrów (ang. *ISAPI filters*), które pozwalają na modyfikację lub rozszerzenie funkcjonalności serwera. Rozszerzenia ISAPI to biblioteki DLL ładowane w procesie, który jest kontrolowany przez serwer IIS. Najpopularniejsze typy aplikacji zaimplementowanych jako rozszerzenia ISAPI to: ASP.NET, Perl, PHP, Active Server Pages (ASP) i Active Visual FoxPro (ActiveVFP).

Ścieżka wywołań

Serwer na podstawie żądania (rozszerzenia pliku z żądania) wybiera aplikację (rozszerzenie ISAPI), która ma obsłużyć dane żądanie. Do aplikacji ASP.NET kierują następujące rozszerzenia: *.aspx*, *.ascx*, *.ashx* i *.asmx*. Przykładowo pliki *.html* lub pliki ze zdjęciami (*.jpg*, *.png*) nie są kierowane do obsługi poprzez ASP.NET. Aby skierować na aplikację ASP.NET inne rozszerzenia niż te podstawowe, należy ustawić w serwerze IIS mapowanie wybranego rozszerzenia na aplikację ASP.NET oraz w pliku *Web.config* zarejestrować *HttpHandler*, który zostanie użyty do przetworzenia żądania.

Pierwsze żądanie do aplikacji ASP.NET

Po otrzymaniu pierwszego żądania klasa *ApplicationManager* tworzy domenę aplikacji (ang. *Application Domain*), która uruchamia i izoluje od siebie różne aplikacje (zmienne globalne). Pozwala na zarządzanie cyklem życia obiektów tworzonych przez aplikacje i posiada listę aplikacji działających w tym samym procesie. W domenie aplikacji tworzony jest obiekt (instancja statycznej klasy) *HostingEnvironment*, który pozwala na zarządzanie aplikacjami (przechowuje informacje o aplikacjach i ścieżkę do folderu, w którym znajduje się aplikacja) (rysunek 5.2).

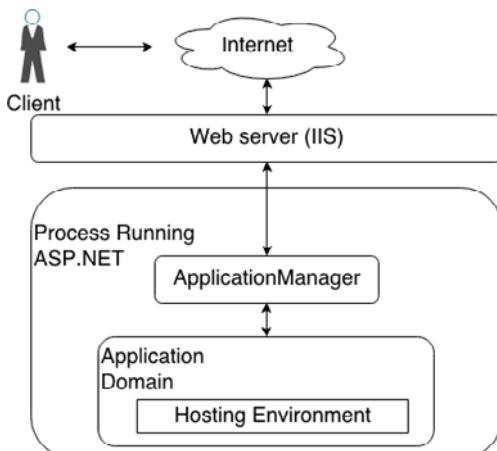
Podstawowe obiekty tworzone dla każdego żądania

Po wygenerowaniu przez ASP.NET domeny aplikacji i obiektu *HostingEnvironment*, które są tworzone tylko jeden raz (podczas uruchamiania aplikacji), dla każdego żądania tworzone są podstawowe obiekty. *HttpContext* składa się z obiektów *HttpRequest* i *HttpResponse*:

- ◆ *HttpRequest* zawiera informacje o żądaniu, łącznie z ciasteczkami i informacjami o przeglądarce klienta;
- ◆ *HttpResponse* zawiera odpowiedź, która zostanie wysłana z powrotem do klienta razem z ciasteczkami.

Rysunek 5.2.

Schemat prezentujący ścieżkę wywołań od klienta przez serwer IIS aż do aplikacji ASP.NET

**HttpApplication**

Po utworzeniu obiektów HttpContext, HttpRequest i HttpResponse tworzony jest obiekt HttpApplication. Jeśli aplikacja ma plik *Global.asax* (listing 5.1), a w nim klasę dziedziczącą po HttpApplication, to zostaje utworzony obiekt klasy, która dziedziczy po HttpApplication (dla aplikacji MVC to klasa MvcApplication) (rysunek 5.3).

Listing 5.1. Zawartość pliku Global.asax w aplikacji MVC

```

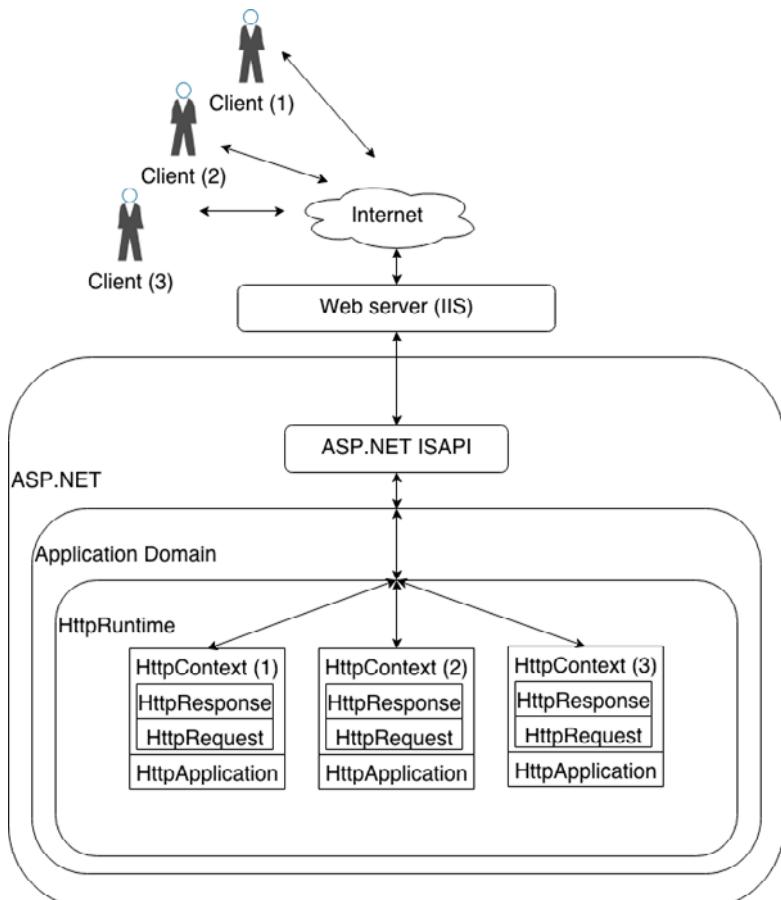
namespace OGL
{
    public class MvcApplication : System.Web.HttpApplication
    {
        protected void Application_Start()
        {
            AreaRegistration.RegisterAllAreas();
            UnityConfig.RegisterComponents();
            GlobalConfiguration.Configure(WebApiConfig.Register);
            FilterConfig.RegisterGlobalFilters
                ↳(GlobalFilters.Filters);
            RouteConfig.RegisterRoutes(RouteTable.Routes);
            BundleConfig.RegisterBundles(BundleTable.Bundles);
        }
    }
}

```

Obiekt HttpApplication podczas pierwszego żądania może zostać wykorzystany dla wielu żądań w celu zwiększenia wydajności (nie jest tworzony osobno dla każdego żądania). Po utworzeniu obiektu HttpApplication uruchamiane są moduły HTTP (HttpModule). Po uruchomieniu wszystkich modułów wywoływana jest metoda Init().

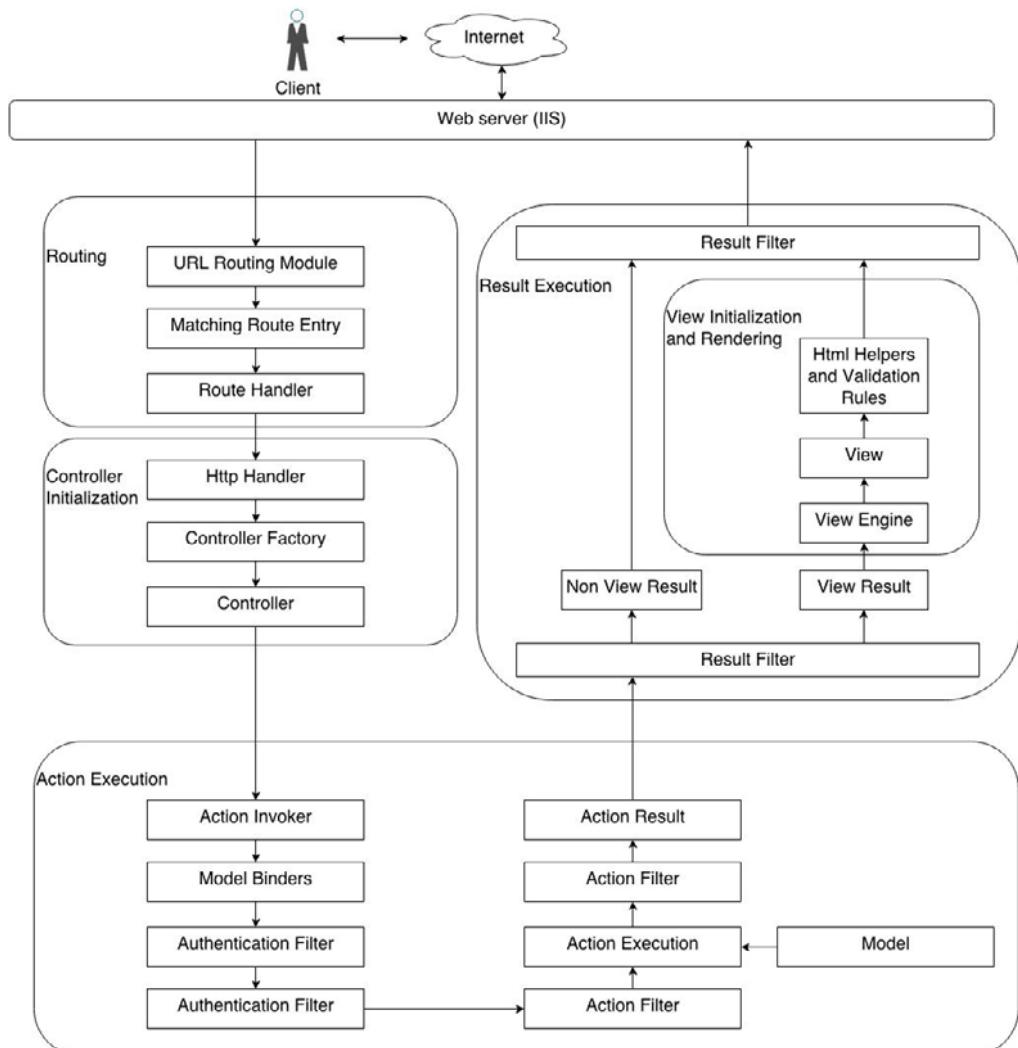
Rysunek 5.3.

Pelny schemat prezentujacy obsługę żądań kierowanych do aplikacji ASP.NET



Ścieżka wywołań dla `HttpApplication` (rysunek 5.4) wygląda następująco:

1. Walidacja żądania (sprawdzenie, czy w żądaniu nie ma potencjalnie niebezpiecznych części kodu).
2. Wykonanie mapowania adresów URL, jeśli zostały skonfigurowane w sekcji `UrlMappingsSection` w pliku `Web.config`.
3. Wywołanie zdarzenia `BeginRequest`.
4. Wywołanie zdarzenia `AuthenticateRequest`.
5. Wywołanie zdarzenia `PostAuthenticateRequest`.
6. Wywołanie zdarzenia `AuthorizeRequest`.
7. Wywołanie zdarzenia `PostAuthorizeRequest`.
8. Wywołanie zdarzenia `ResolveRequestCache`.
9. Wywołanie zdarzenia `PostResolveRequestCache`.



Rysunek 5.4. Schemat wywołań dla aplikacji ASP.NET MVC

10. Na podstawie rozszerzenia zostaje utworzona klasa implementująca `IHttpHandler` do przetworzenia żądania. Jeśli żądanie jest kierowane do strony WWW, przed utworzeniem instancji obiektu wykonywana jest komplikacja.
11. Wywołanie zdarzenia `PostMapRequestHandler`.
12. Wywołanie zdarzenia `AcquireRequestState`.
13. Wywołanie zdarzenia `PostAcquireRequestState`.
14. Wywołanie zdarzenia `PreRequestHandlerExecute`.
15. Wywołanie metody `ProcessRequest` (lub asynchronicznej: `BeginProcessRequest`) dla żądania na podstawie rozszerzenia (klasy `IHttpHandler`).

16. Wywołanie zdarzenia PostRequestHandlerExecute.
17. Wywołanie zdarzenia ReleaseRequestState.
18. Wywołanie zdarzenia PostReleaseRequestState.
19. Filtry odpowiedzi (ang. *Result Filter*) — jeśli są zdefiniowane.
20. Wywołanie zdarzenia UpdateRequestCache.
21. Wywołanie zdarzenia PostUpdateRequestCache.
22. Wywołanie zdarzenia EndRequest.

Uchwyty i moduły HTTP

Uchwyty HTTP

Uchwyty HTTP (ang. *Http Handlers*) są powiązane z określonymi rozszerzeniami i są używane np. w przypadku kanałów RSS, dynamicznego generowania obrazów lub modyfikacji. Domyślnym handlerem w aplikacjach ASP.NET jest *ASP.NET Page Handler* (rozszerzenie *.aspx*), który zwraca widok (stronę WWW, w adresach URL aplikacji nie ma rozszerzenia *.aspx*, ponieważ adres tłumaczony jest poprzez routing i przepisywanie adresów URL (ang. *friendly URL*)). Uchwyty — w przeciwieństwie do modułów HTTP — są wywoływane tylko dla żądań z określonymi rozszerzeniami, a nie dla każdego żądania. Własne *HttpHandler* należy rejestrować w pliku *Global.asax*.

Moduły HTTP

Moduły HTTP (ang. *Http Modules*) są wykonywane przy każdym żądaniu, bez względu na jego typ. Moduły HTTP są wykorzystywane m.in. do tworzenia statystyk, logowania wyjątków i implementacji bezpieczeństwa w aplikacjach. Moduły są wspólne dla wszystkich aplikacji ASP.NET (MVC, Web Forms) i można nimi zarządzać zarówno w konfiguracji serwera IIS, jak i w aplikacji. W trybie *Integrated Pipeline* konfiguracja modułów pomiędzy aplikacją a serwerem IIS jest współdzielona.

HttpHandler a HttpModule

Różnicę pomiędzy modułem a handlerem zobrazowano na przykładzie modułu logowania zdarzeń w aplikacji oraz uchwycie do pliku. Po odwiedzeniu adresu *www.domena.pl/plik.pdf* zostaną uruchomione zarówno moduł logowania (zapisze wywołane żądanie), jak i uchwyt, który zwróci plik. Na podstronie logowania *www.domena.pl/podstrona* zostanie uruchomiony tylko moduł logowania⁶.

⁶ <http://msdn.microsoft.com/en-us/library/bb470252.aspx>

Kontroler

Kontroler odpowiada za reakcje i obsługę zdarzeń użytkownika, np. kliknięcie przycisku lub odwiedzenie danej podstrony. Kontroler nie powinien zawierać logiki biznesowej ani logiki odpowiedzialnej za dostęp do danych. W ASP.NET MVC określone akcje są wywoływanie na podstawie routingu — adresu URL. Zgodnie z konwencją zostaje wywołana odpowiednia metoda z kontrolera. Każda metoda w kontrolerze zwraca określony typ rezultatu akcji (ang. *Action Result*). Oczywiście poza metodami, które zwracają typy rezultatu, można także tworzyć zwykłe pomocnicze metody zwracające np. wartość int.

Zgodnie z konwencją zwrócenie widoku bez podania nazwy wybiera widok o takiej samej nazwie jak nazwa metody kontrolera. Dla metody `ViewResult()` z kontrolera `HomeController` zwrócony zostanie widok o nazwie *View Result* z katalogu *Home* (listing 5.2).

Listing 5.2. Przykładowa klasa kontrolera o nazwie *HomeController*

```
public class HomeController : Controller
{
    public ActionResult ViewResult()
    {
        ViewBag.Message = "Strona główna";

        return View();
    }
}
```

W metodzie `ViewResult()` (listing 5.2) podane dwie linijki są równoważne:

```
return View();
return View("ViewResult");
```

Aby zwrócić widok o innej nazwie niż nazwa metody, wystarczy podać nazwę widoku, który ma zostać zwrócony:

```
return View("Nazwa_widoku");
```

Typy rezultatu

Dostępnych jest kilka typów rezultatu dla akcji z kontrolera (ang. *Action Results*):

- ◆ `ContentResult` — zwraca zdefiniowany przez użytkownika typ danych:

```
public ActionResult ContentResult()
{
    return Content("<p>Akapit</p>");
}
```

- ◆ **EmptyResult** — zwraca wartość null:

```
public ActionResult EmptyResult()
{
    return new EmptyResult();
}
```

- ◆ **FileResult** — zwraca plik na jeden ze sposobów:

- ◆ **FileContentResult** — zwraca tablicę bitów znaków jako plik,
- ◆ **FileStreamResult** — zwraca strumień danych jako plik,
- ◆ **FilePathResult** — zwraca ścieżkę do pliku na dysku.

```
public ActionResult FilePathResult()
{
    string path = AppDomain.CurrentDomain.BaseDirectory
        ↪+ "pliki/";
    string fileName = "plik.txt";
    return File(path + fileName, "text/plain", "plik.txt");
}
```

- ◆ **JavaScriptResult** — zwraca kod JavaScript wykonywany po stronie klienta:

```
public ActionResult JSResult()
{
    return JavaScript("var zmienna = 0;");
}
```

- ◆ **JsonResult** — zwraca dane w formacie JSON:

```
public ActionResult JsonResult()
{
    return Json(new { Numer = 1, Napis = "JSON result" });
}
```

- ◆ **RedirectResult** — przekierowuje do innej metody/akcji za pomocą adresu URL:

```
public ActionResult RedirectResult()
{
    return RedirectToAction("index", "home");
}
```

- ◆ **RedirectToRouteResult** — przekierowuje do innej metody/akcji bez odwiedzania innego adresu URL:

```
public ActionResult RedirectToRouteResult()
{
    return RedirectToRoute("index", "home");
}
```

- ◆ **ViewResult** — zwraca widok, czyli wyświetla stronę WWW:

```
public ActionResult ViewResult()
{
    ViewBag.Message = "Strona główna";
    return View();
}
```

- ◆ PartialViewResult — zwraca widok częściowy, który może być częścią całego widoku (strony WWW):

```
public ActionResult PartialViewResult()
{
    return PartialView("_NazwaWidokuPartial");
}
```

Parametry akcji

Parametry do akcji można przekazywać za pomocą żądań GET (w adresie URL) lub żądań POST (w treści żądania). Parametry są przyporządkowywane na podstawie nazw i przy użyciu reguł routingu.

Żądanie GET

Żądania GET nie powinny być wykorzystywane do przesyłania danych przeznaczonych do zapisania na serwerze. Można przekazać dane do kontrolera poprzez żądanie GET, jednak długość adresu URL jest ograniczona i przeglądarki mogą się różnie zachowywać. Dodatkowo dochodzą problemy ze znakami specjalnymi, spacją oraz znakami narodowymi. Dlatego żądania GET powinny tylko wskazywać na zasób, jaki ma zostać pobrany z serwera, lub informować, jaką stronę wyświetlić. Przykładowe żądanie GET:

http://www.jakasStrona.pl/ogloszenie/pokaz?Tresc=TrescOgłoszenia&Tytuł=TitułOgłoszenia.

Żądanie POST

Aby wykonać żądanie POST, należy użyć formularza HTML (w składni silnika Razor należy użyć @using (Html.BeginForm())), jak pokazano na listingu 5.3, lub wysłać żądanie za pomocą języków JavaScript i AJAX, jak na listingu 5.4. Przedstawione sposoby dadzą ten sam efekt. Następuje wysłanie tytułu i treści ogłoszenia do wybranej akcji, w tym przypadku do akcji, która doda nowe ogłoszenie do bazy danych.

Listing 5.3. Formularz HTML w MVC i Razor

```
@model Models.Ogłoszenie
@using (Html.BeginForm("dodaj", "ogłoszenie", FormMethod.Post))
{
    @Html.AntiForgeryToken()

    <div class="form-horizontal">
        <h4>Ogłoszenie</h4>
        <hr />
        @Html.ValidationSummary(true, "", new { @class =
            "text-danger" })
        <div class="form-group">
            @Html.LabelFor(model => model.Tresc, htmlAttributes:
                new { @class = "control-label col-md-2" })
            <div class="col-md-10">
```

```
    @Html.EditorFor(model => model.Tresc, new
    =>{ htmlAttributes = new { @class = "form-control" } })
    @Html.ValidationMessageFor(model => model.Tresc, "", 
    =>new { @class = "text-danger" })
  </div>
</div>

<div class="form-group">
  @Html.LabelFor(model => model.Tytul, htmlAttributes: new {
    =>@class = "control-label col-md-2" })
  <div class="col-md-10">
    @Html.EditorFor(model => model.Tytul,
    =>new { htmlAttributes = new {
      =>@class = "form-control" } })
    @Html.ValidationMessageFor(model => model.Tytul, "", 
    =>new { @class = "text-danger" })
  </div>
</div>

<div class="form-group">
  <div class="col-md-offset-2 col-md-10">
    <input type="submit" value="Create" class="btn
    =>btn-default" />
  </div>
</div>
</div>
}
```

Listing 5.4. Żądanie POST w AJAX

```
var dane = { Tytuł: $("#TytulId").val(), Tresc: $("#TrescId").val(),}

$.ajax({
  url: 'www.jakasStrona.pl/ogloszenie/dodaj',
  type: 'POST',
  dataType: "json",
  contentType: 'application/json',
  data: JSON.stringify(dane)
});
```

Filtry akcji

Filtry akcji (ang. *Action Filters*) to atrybuty, które można przypisać do akcji kontrolera lub do całej klasy kontrolera (wszystkich akcji w kontrolerze), aby zmienić sposób wykonywania akcji. Wszystkie filtry dziedziczą po interfejsach, dlatego można w łatwy sposób napisać swoje własne implementacje filtrów. Można również pisać własne filtry do innych metod, niekoniecznie tych z kontrolerów⁷.

⁷ [http://msdn.microsoft.com/en-us/library/system.web.mvc.filterattribute\(v=vs.108\).aspx](http://msdn.microsoft.com/en-us/library/system.web.mvc.filterattribute(v=vs.108).aspx)

Główne typy filtrów akcji:

- ◆ filtry autoryzacji (ang. *Authorization Filters*) — implementują interfejs `IAuthorizationFilter`; są wywoływanie na samym początku — przed metodą (akcją) i wszystkimi innymi filtrami;
- ◆ filtry akcji (ang. *Action Filters*) — implementują interfejs `IActionFilter`, który deklaruje metody:
 - ◆ `OnActionExecuting` — metoda wywoływana przed rozpoczęciem danej akcji,
 - ◆ `OnActionExecuted` — metoda wywoływana po zakończeniu danej akcji;
- ◆ filtry rezultatu (ang. *Result Filters*) — implementują interfejs `IResultFilter`, który deklaruje metody:
 - ◆ `OnResultExecuting` — metoda wywoływana przed zwróceniem rezultatu,
 - ◆ `OnResultExecuted` — metoda wywoływana po zwróceniu rezultatu;
- ◆ filtry wyjątków (ang. *Exception Filters*) — implementują interfejs `IExceptionFilter` i są wywoływanie, gdy wystąpi wyjątek.

Najważniejsze filtry zaimplementowane w MVC:

- ◆ `AuthorizeAttribute` — atrybut wymagający autoryzacji, aby wywołać daną metodę:

```
[Authorize]
public ActionResult Contact(int id)
{
    ViewBag.Message = "Kontakt.";
    return View();
}
```

- ◆ `HandleErrorAttribute` — określa, w jaki sposób mają być wyłapywane wyjątki:

```
[HandleError]
public ActionResult Contact(int id)
{
    ViewBag.Message = "Kontakt.";
    return View();
}
```

- ◆ `RequireHttpsAttribute` — wymaga połączenia szyfrowanego HTTPS:

```
[RequireHttps]
public ActionResult Create()
{
    return View();
}
```

- ◆ `ValidateAntiForgeryTokenAttribute` — sprawdza poprawność klucza:

```
[ValidateAntiForgeryToken]
public ActionResult Login(LoginModel model, string returnUrl)
{
    if (ModelState.IsValid && WebSecurity.Login(model.UserName,
                                                model.Password, persistCookie: model.RememberMe))
    {
```

```

        return RedirectToAction(returnUrl);
    }

    // Jeśli coś pójdzie nie tak, wyświetl ponownie formularz
    ModelState.AddModelError("", "Niepoprawne dane");
    return View(model);
}

```

- ◆ **ValidateInputAttribute** — wymusza walidację danych wejściowych.
- ◆ **HttpGetAttribute** — metoda będzie obsługiwać tylko żądania GET. Domyslnie metoda obsługuje żądania GET, dlatego nie trzeba dodawać atrybutu [HttpGet].
- ◆ **HttpPostAttribute** — metoda będzie obsługiwać tylko żądania POST:

```

[HttpPost]
public ActionResult ExternalLogin(string provider, string returnUrl)
{
    return new ExternalLoginResult(provider,
        Url.Action("ExternalLoginCallback", new {
            returnUrl = returnUrl }));
}

```

- ◆ **OutputCacheAttribute** — metoda, która będzie cachowana:

```

[OutputCache(Duration = 10, VaryByParam = "none")]
public ActionResult Index()
{
    return View();
}

```

Atrybuty umieszcza się w nawiasach kwadratowych []. Atrybut umieszczony przed metodą bądź akcją odnosi się tylko do danej akcji. Aby atrybut odnosił się do wszystkich akcji w kontrolerze, należy umieścić go przed klasą kontrolera. Atrybut odnoszący się do wszystkich akcji z kontrolera został przedstawiony na listingu 5.5. Atrybut [AllowAnonymous] umieszczony przed akcją Login udostępnia daną akcję dla niezalogowanych (uwierzytelnionych) użytkowników. Wykorzystuje się go, gdy cała klasa została oznaczona jako wymagająca uwierzytelniania (zalogowania — atrybut [Authorize]).

Listing 5.5. Atrybut dla wszystkich akcji kontrolera

```

[Authorize]
public class HomeController : Controller
{
    // GET: /Account/Login
    [AllowAnonymous]
    public ActionResult Login(string returnUrl)
    {
        ViewBag.ReturnUrl = returnUrl;
        return View();
    }

    // Ciało klasy i akcje kontrolera
}

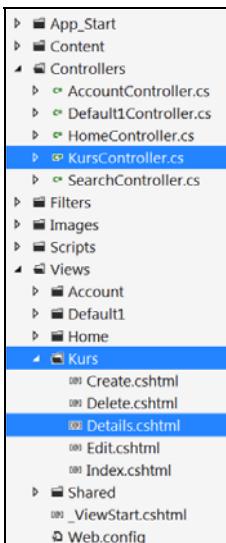
```

Widok

Kontroler może zwrócić widok, czyli stronę WWW (kod HTML), jako rezultat wywołanej akcji. Widok przetwarza dane z modelu i wyświetla je w określony w widoku sposób. Widoki znajdują się w folderze *Views* (rysunek 5.5). Nazwa kontrolera jest nazwą folderu, w którym znajdują się widoki. Nazwy konkretnych widoków są takie same jak nazwy akcji w kontrolerze. Dla kontrolera *KursController* został utworzony folder *Kurs* w folderze *Views*, a w nim widoki dla każdej akcji w kontrolerze: Create, Delete, Details, Edit i Index.

Rysunek 5.5.

Widok foldera Views



Zasady odnajdywania widoków

Widoki w aplikacji są poszukiwane poprzez silnik renderujący (Razor) według następującej kolejności:

- ◆ jeśli projekt nie znajduje się w obszarze (ang. *Area*):

```

~/Views/{1}/{0}.cshtml
~/Views/{1}/{0}.vbhtml
~/Views/Shared/{0}.cshtml
~/Views/Shared/{0}.vbhtml
    
```

- ◆ jeśli projekt znajduje się w obszarze:

```

~/Areas/{2}/Views/{1}/{0}.cshtml
~/Areas/{2}/Views /{1}/{0}.vbhtml
~/Areas/{2}/Views /Shared/{0}.cshtml
~/Areas/{2}/Views /Shared/{0}.vbhtml
    
```

gdzie:

- ◆ {2} — nazwa przestrzeni,
- ◆ {1} — nazwa kontrolera,
- ◆ {0} — nazwa widoku.

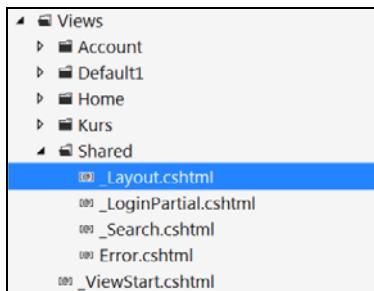
Aby zaimplementować własne rozwiązanie poszukujące widoków w niestandardowy sposób, można napisać klasę dziedziczącą po klasie widoku Razor (`RazorViewEngine`) albo napisać własny silnik renderujący poprzez implementację interfejsu `IView`.

Folder Shared

W folderze *Shared* znajdują się współdzielone widoki (layouty lub widoki częściowe), a więc części strony WWW wspólne dla wielu podstron. Plik `_Layout.cshtml` (rysunek 5.6) odpowiada za ogólny wygląd strony (ang. *Template*).

Rysunek 5.6.

Widok folderu Shared

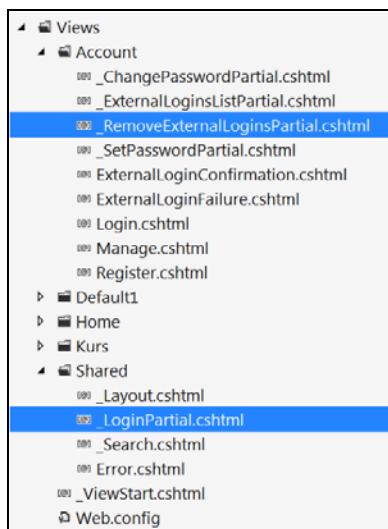


Widoki częściowe

Widok częściowy (ang. *Partial View*) to widok odpowiadający części strony (rysunek 5.7). Widok *Partial View* może się kilkakrotnie powtarzać na jednej podstronie, np. tabela z komentarzami, przy czym każdy komentarz to pojedynczy widok *Partial View*. Kolejne widoki *Partial View*, czyli komentarze, są wyświetlane np. za pomocą pętli `foreach`. Widokiem częściowym może być także powtarzająca się część strony na wielu podstronach, tak jak widok `_LoginPartial` służący do logowania. Bardzo szybko można rozpoznać, że jest to widok częściowy, ponieważ według konwencji nazwa widoku *Partial View* powinna się zaczynać od znaku `_`. Widok częściowy może wyświetlać te same dane w różny sposób w zależności od podstrony (jeśli wczytany zostanie inny plik CSS).

Do widoku typowanego można przekazać tylko jeden model (lub `ViewModel`). Jeśli strona składa się z kilku widoków częściowych, możliwe staje się wykorzystanie kilku modeli na jednej stronie WWW, ponieważ każdy widok częściowy posiada swój własny model. Można utworzyć jeden widok na podstawie klasy `ViewModel` lub kilka widoków częściowych — każdy z innym modelem. Widoki częściowe pozwalają na ponowne wykorzystanie na innych typach podstron, natomiast `ViewModel` pasuje tylko do jednej pod-

Rysunek 5.7.
Pliki widoku
częściowego



strony. Przy podejściu wykorzystującym `ViewModel` jest wykonywana tylko jedna akcja z kontrolera. Przy użyciu widoków częściowych musi zostać wykonanych kilka akcji, oddziennie dla każdego widoku częściowego, co skutkuje mniejszą wydajnością.

Razor

Razor to silnik renderujący wprowadzony w MVC 3, pozwalający na bardzo łatwe oddzielenie kodu HTML od kodu aplikacji. W porównaniu do starszych silników renderujących, aby uzyskać taki sam efekt, wymaga napisania mniejszej ilości kodu. Aby wyświetlić wartość zmiennej, wystarczy postawić przed nią znak @. Analogicznie postępuje się w przypadku pętli oraz innych elementów nienależących do składni języka HTML czy JavaScript. Dodatkowo Razor obsługuje klamry, które są bardzo pomocne, gdy użycie tego silnika wymaga więcej niż jednej linii kodu. Znakiem @ oznacza się kod, który ma zostać wykonany po stronie serwera, a więc nie jest kodem HTML (listing 5.6). Aby zakomentować kod, należy użyć komentarza wielolinijkowego:

`@*fragment kodu*@`

Listing 5.6. Przykładowy kod widoku z silnika Razor

```

@{
    ViewBag.Title = "Transport | Rozkłady jazdy | Car-Match - 
                    szukam-transportu.pl";
}

@section featured {
    <section class="featured">
        <div class="content-wrapper">
            <hgroup class="title">
                <h1>@ViewBag.Title.</h1>
                @*<h2>@ViewBag.Message</h2>
                @Gui.pl.About*@

```

```
</hgroup>
</div>
</section>
}
<div>
    @Html.Action("Search", "Search")
</div>
```

Dodatkowe właściwości silnika Razor

Znakiem @: oznacza się kod, który nie powinien być wykonywany po stronie serwera, a znajduje się w bloku kodu oznaczonym jako wykonywany na serwerze, np.:

```
@if(item.Id != 3)
{
    @: Id różne od 3
}
```

Wnętrze każdego znacznika HTML automatycznie zostaje potraktowane jako kod HTML, nawet jeśli znacznik znajduje się w bloku kodu oznaczonym @:, dlatego konieczne jest ponowne użycie znaku @:, aby oznaczyć kod wykonywany na serwerze:

```
@foreach (var item in Model) {
    <li>
        @item.Id

        @if(item.Id != 3)
        {
            @: Id różne od 3
        }
        else
        {
            @: Id jest równe @item.Id
        }
        @:cbgbfbgf
    </li>
}
```

Aby wyświetlić zawartość w kilku liniach, można użyć tagu `<p>` (listing 5.7), wielokrotnie znacznika @ (listing 5.8) lub elementu `<text>` (listing 5.9).

Listing 5.7. Użycie tagu `<p>`

```
@if(item.Id != 3)
{
    <p>
        Linia pierwsza
        Linia druga
    </p>
}
```

Listing 5.8. Wielokrotne użycie znacznika

```
@if(item.Id != 3)
{
    @: Linia pierwsza
    @: Linia druga
}
```

Listing 5.9. Użycie elementu <text>

```
@if(item.Id != 3)
{
    <text>
        Linia pierwsza
        Linia druga
    </text>
}
```

ViewBag, ViewData i TempData

W ASP.NET MVC istnieje możliwość przekazywania danych z kontrolera do widoku. ViewBag i ViewData są prawie identyczne, natomiast TempData posiada dłuższy czas istnienia⁸. Zarówno ViewBag, jak i ViewData to obiekty krótkotrwałe, które są tworzone na czas otwarcia widoku. Po kliknięciu linka do innego adresu URL zostają skasowane. Różnice pomiędzy ViewBag i ViewData są następujące:

- ◆ ViewData to kolekcja typu słownik (ang. *Dictionary*), zawierająca typy object i dziedzicząca po klasie ViewDataDictionary. Kluczami w tym słowniku są zmienne typu string.
- ◆ ViewData wymaga rzutowania dla typów złożonych oraz sprawdzania wartości null, aby zapobiec błędom.
- ◆ ViewBag jest typu dynamic (C# 4.0), a więc można zmieniać typ zmiennej w dowolnym momencie i nie wymaga rzutowania, jak jest to w przypadku ViewData.

Oto przykład ViewBag w kontrolerze:

```
public ActionResult Login(string returnUrl)
{
    ViewBag.ReturnUrl = returnUrl;
    return View();
}
```

Tak przebiega odczyt wartości z ViewBag w widoku:

```
@ViewBag.ReturnUrl
```

⁸ <http://www.dotnet-tricks.com/Tutorial/mvc/9KHW190712-ViewData-vs-ViewBag-vs-TempData-vs-Session.html>

Poniżej zamieszczono przykład ViewData w kontrolerze:

```
public ActionResult Login(string returnUrl)
{
    ViewData["ReturnUrl"] = returnUrl;
    return View();
}
```

Odczyt wartości z ViewData w widoku przebiega następująco:

```
@ViewData["ReturnUrl"]
```

TempData pozwala przekazywać dane pomiędzy obecnym i kolejnym żądaniem HTTP — nie tylko pojedynczym, jak jest w przypadku ViewData i ViewBag. Dane z TempData są zapisywane w sesji, ale tylko na czas jednego żądania. Wiąże się to z pewnymi konsekwencjami. Przeglądarka użytkownika może mieć wyłączone ciasteczka, przez co nie uda się przekazać danych z TempData. Możliwe jest również trzymanie sesji na serwerze, jednak to rozwiązanie wiąże się z kolejnym problemem. W przypadku gdy portal obsługuje więcej niż jeden fizyczny serwer, ruch jest dzielony w zależności od obciążenia konkretnego serwera. Może się zdarzyć, że użytkownik przy kolejnym żądaniu zostanie skierowany na inny serwer, który nie posiada zapisanej sesji. Za pomocą rozdzielania ruchu (ang. *Load Balancer*) można kierować użytkownika za każdym razem na ten sam serwer, jednak nie jest to dobre rozwiązanie. W razie awarii dane są tracone. Jedynym słusznym rozwiązaniem w tym wypadku jest przechowywanie sesji we wspólnej bazie danych lub na osobnym serwerze dedykowanym. Dane sesji są wówczas współdzielone pomiędzy serwerami. Do przechowywania sesji najlepiej nadają się słownikowe bazy NoSQL, ponieważ dane sesji są typu słownikowego.

Poniżej zaprezentowano przykład TempData w kontrolerze:

```
public ActionResult Login(string returnUrl)
{
    TempData["ReturnUrl"] = returnUrl;
    return View();
}
```

Odczyt wartości z ViewData w widoku przebiega następująco:

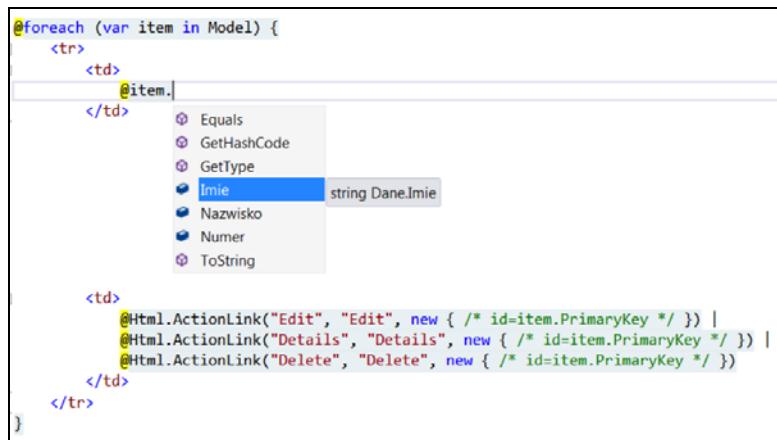
```
@TempData["ReturnUrl"]
```

Widoki typowane — Strongly Typed Views

Oprócz przekazywania danych do widoku za pomocą ViewBag, ViewData i TempData istnieje jeszcze jeden podstawowy i zarazem zalecany sposób umożliwiający przekazywanie danych do widoku. Polega na przekazaniu danych określonego typu (modelu danych) do widoku. Dzięki takiemu sposobowi przekazania danych otrzymuje się widok ściśle typowany. Znane są zatem typ i struktura danych przekazywanych do widoku. Umożliwia to automatyczne wygenerowanie widoku na podstawie przekazanego modelu danych oraz umożliwia podpowiedź *IntelliSense* w pliku z widokiem (rysunek 5.8). Klasy z modelem nie zawierają wszystkich danych potrzebnych dla widoku, dlatego tworzy się ViewModel, czyli klasy na potrzeby określonego widoku. ViewModel może zawierać dane z różnych modeli (listing 5.10).

Rysunek 5.8.

*Podpowiedź
IntelliSense dostępny w widoku typowanym*

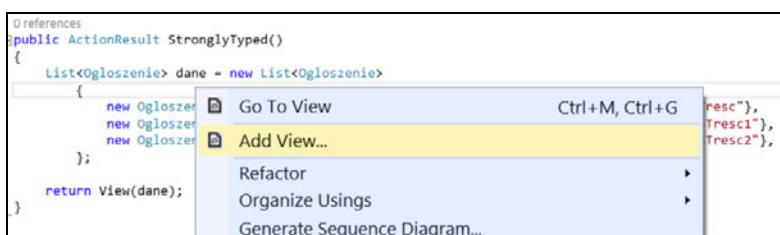
**Listing 5.10.** Kod akcji zwracającej dane jako parametr

```
public ActionResult StronglyTyped()
{
    List<Ogloszenie> dane = new List<Ogloszenie>
    {
        new Ogloszenie{ DataDodania = DateTime.Now, Tytul =
            ↳"JakisTytul", Tresc = "JakasTresc"} ,
        new Ogloszenie{ DataDodania = DateTime.Now, Tytul =
            ↳"JakisTytul1", Tresc = "JakasTresc1"} ,
        new Ogloszenie{ DataDodania = DateTime.Now, Tytul =
            ↳"JakisTytul2", Tresc = "JakasTresc2"} ,
    };
    return View(dane);
}
```

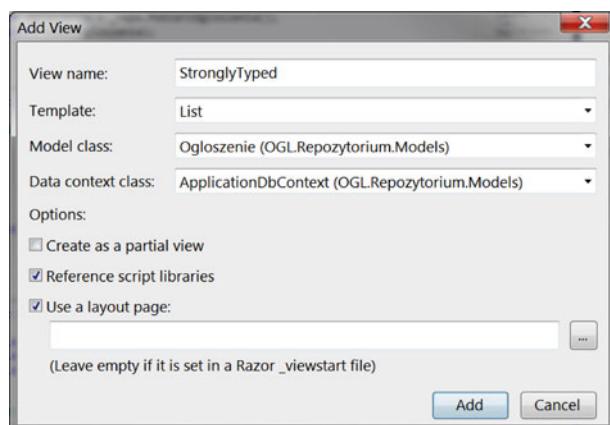
Aby dodać widok do akcji z kontrolera, należy kliknąć prawym przyciskiem myszy i z menu kontekstowego wybrać opcję *Add View...* (rysunek 5.9).

Rysunek 5.9.

Kliknięcie w bloku kodu dla danej akcji



Pojawi się okno *Add View* (rysunek 5.10) — można w nim wybrać klasę z modelem danych (ang. *Model class*) oraz typ szablonu, na podstawie którego ma zostać wygenerowany widok (ang. *Template*). *Model class* to klasa danych przekazywanych do widoku, na podstawie której zostaną utworzone kolumny w widoku. *Template* w tym przypadku to lista, czyli w pętli zostaną wyświetcone kolejne elementy z listy przekazanej do widoku.

Rysunek 5.10.
Okno dodawania widoku

Listingi przedstawiają widok wygenerowany przed zaznaczeniem opcji *Strongly Typed* (listing 5.11) i po jej zaznaczeniu (listing 5.12).

Listing 5.11. *Widok wygenerowany bez zaznaczenia opcji Strongly Typed*

```
@{  
    ViewBag.Title = "StronglyTyped";  
}  
<h2>StronglyTyped</h2>
```

Listing 5.12. *Widok wygenerowany jako Strongly Typed z szablonem List*

```
@model IEnumerable<OGL.Repozytorium.Models.Ogłoszenie>  
  
{@  
    ViewBag.Title = "StronglyTyped";  
}  
  
<h2>StronglyTyped</h2>  
  
<p>  
    @Html.ActionLink("Dodaj nowy", "Dodaj")  
</p>  
<table class="table">  
    <tr>  
        <th>  
            @Html.DisplayNameFor(model => model.Tresc)  
        </th>  
        <th>  
            @Html.DisplayNameFor(model => model.Tytul)  
        </th>  
        <th>  
            @Html.DisplayNameFor(model => model.DataDodania)  
        </th>  
        <th></th>  
    </tr>  
    @foreach (var item in Model) {  
        <tr>
```

```

<td>
    @Html.DisplayFor(modelItem => item.Tresc)
</td>
<td>
    @Html.DisplayFor(modelItem => item.Tytul)
</td>
<td>
    @Html.DisplayFor(modelItem => item.DataDodania)
</td>
<td>
    @Html.ActionLink("Edytuj", "Edytuj",
        new { id=item.Id }) |
    @Html.ActionLink("Szczegóły", "Szczegoly",
        new { id=item.Id }) |
    @Html.ActionLink("Usuń", "Usun", new { id=item.Id })
</td>
</tr>
}
</table>

```

HTML helpery

Rzeczą, na którą warto zwrócić uwagę, są HTML helpery⁹. Umożliwiają one wydzielenie powtarzającej się części kodu HTML, przez co nie ma potrzeby powielania tego samego kodu w kilku miejscach na jednej stronie. Wiele helperów jest domyślnie zaimplementowanych, np. lista rozwijana czy pola tekstowe. Dodatkowo mają one zaimplementowaną obsługę walidacji, przez co niepoprawnie wypełnione pola podświetlają się na czerwono. Można również tworzyć własne helpery poprzez użycie metod rozszerzających dla klasy `HtmlHelper`, które mogą znacznie usprawnić pracę nad aplikacją. HTML helpery można podzielić na grupy generujące elementy HTML lub generujące linki.

Dostępne HTML helpery dla elementów HTML:

- ◆ `BeginForm()` — otwiera formularz,
- ◆ `EndForm()` — zamyka formularz,
- ◆ `TextArea()` — pole tekstowe — wiele wierszy,
- ◆ `TextBox()` — pole tekstowe — jeden wiersz,
- ◆ `CheckBox()` — pole wyboru wielokrotnego,
- ◆ `RadioButton()` — pole wyboru jednokrotnego,
- ◆ `ListBox()` — lista,
- ◆ `DropDownList()` — lista rozwijana,
- ◆ `Hidden()` — pole typu `hidden`,
- ◆ `Password()` — pole z hasłem.

⁹ [http://msdn.microsoft.com/en-us/library/system.web.mvc.htmlhelper\(v=vs.118\).aspx](http://msdn.microsoft.com/en-us/library/system.web.mvc.htmlhelper(v=vs.118).aspx)

Oto przykładowy kod helpera generującego link:

```
@Html.ActionLink("Usuń", "Usun", new {Id=5})
```

Wygenerowany kod HTML dla przeglądarki wygląda następująco:

```
<a href="/Home/Usun/5">Usuń</a>
```

Aby wygenerować widok *Partial View* (wykonać akcję) na stronie, używa się helpera:

```
@Html.Action("Lista", "Ogloszenia", null)
```

Zwrócony zostanie kod HTML widoku, a nie link do widoku.

Aby zobaczyć kompletną listę dostępnych helperów, wystarczy w pliku z widokiem wpisać @Html, a Visual Studio wyświetli listę dostępnych helperów.

Paczki skryptów i minimalizacja — Script/CSS Bundling and Minification

Minimalizacja oraz tworzenie paczek skryptów CSS i JavaScript to procesy służące zmniejszeniu wielkości plików i ograniczeniu liczby żądań HTTP do minimum. Plik odpowiedzialny za budowanie i minimalizację znajduje się w folderze *App_Start* i nosi nazwę *BundlesConfig.cs*.

Minimalizacja (ang. *Minification*) to proces polegający na zmianie nazw zmiennych na krótsze (zazwyczaj jednoznakowe), usuwaniu białych znaków, komentarzy do kodu oraz przejść do kolejnej linii w celu osiągnięcia jak najmniejszego rozmiaru pliku.

Tworzenie paczek skryptów (ang. *Bundling*) polega na łączeniu kilku osobnych skryptów i plików z kodem w jeden większy plik w celu przesłania do klienta. Operacja ta pozwala zredukować liczbę żądań HTTP wysyłanych do serwera. Rozwiązuje również problem odświeżania skryptów zapisanych u klienta, jeśli zostały wprowadzone zmiany w skrypcie na serwerze. Za tworzenie, rejestrację i konfigurację paczek skryptów odpowiedzialna jest metoda RegisterBundles (listing 5.13)¹⁰.

Listing 5.13. Kod metody RegisterBundles

```
public static void RegisterBundles(BundleCollection bundles)
{
    bundles.Add(new ScriptBundle("~/bundles/jquery")
        .Include("~/Scripts/jquery-{version}.js"));

    bundles.Add(new ScriptBundle("~/bundles/jqueryval")
        .Include("~/Scripts/jquery.unobtrusive*", 
        "~/Scripts/jquery.validate*"));

    bundles.Add(new StyleBundle("~/Content/css")
        .Include("~/Content/site.css"));
```

¹⁰ <http://www.asp.net/mvc/tutorials/mvc-4/bundling-and-minification>

```

bundles.Add(new StyleBundle("~/Content/themes/base/css")
    ➔.Include("~/Content/themes/base/jquery.ui.core.css",
    ➔ "~/Content/themes/base/jquery.ui.resizable.css",
    ➔ "~/Content/themes/base/jquery.ui.theme.css"));
}

```

W ASP.NET MVC zaimplementowanych jest kilka domyślnych rozwiązań:

- ◆ w trybie release wybierane są skrypty z rozszerzeniem *.min* (jeśli istnieją);
- ◆ w trybie debug wybierane są pełne pliki bez rozszerzenia *.min*;
- ◆ aby zapobiec konieczności ciągłych zmian wersji skryptów jQuery, można użyć *{version}*, który automatycznie wybierze wersję stosowaną przez aplikację (podczas aktualizacji zmienia się tylko skrypt jQuery — bez żadnych zmian w kodzie aplikacji):

```

bundles.Add(new ScriptBundle("~/bundles/jquery")
    ➔.Include("~/Scripts/jquery-{version}.js"));

```

Łączenie kilku stylów CSS w jeden (metoda *StyleBundle*) przebiega następująco:

```

bundles.Add(new StyleBundle("~/Content/themes/base/css")
    ➔.Include("~/Content/themes/base/jquery.ui.core.css",
    ➔ "~/Content/themes/base/jquery.ui.resizable.css",
    ➔ "~/Content/themes/base/jquery.ui.theme.css"));

```

Poniżej zaprezentowano łączenie kilku skryptów JS w jeden (metoda *ScriptBundle*):

```

bundles.Add(new ScriptBundle("~/bundles/jqueryval")
    ➔.Include("~/Scripts/jquery.unobtrusive*",
    ➔ "~/Scripts/jquery.validate*"));

```

A tak używa się paczek w widoku:

```

@Scripts.Render("~/bundles/jquery")
@Styles.Render("~/Content/css")

```

Istnieje kilka zasad, których należy przestrzegać podczas ładowania i umieszczania skryptów w kodzie strony, aby umożliwić pozornie szybsze jej ładowanie (struktura dokumentu będzie gotowa, a użytkownik będzie mógł przeglądać stronę bez wiedzy o tym, że strona jest ciągle wczytywana):

- ◆ w sekcji *<head>* umieszcza się tylko skrypty niezbędne do prawidłowego wyświetlenia się strony;
- ◆ w sekcji *<body>* na samym końcu umieszcza się pozostałe skrypty, które nie wymagają, aby były załadowane przed strukturą HTML strony (np. walidacja, efekty graficzne itp.);
- ◆ dla niektórych skryptów nie ma możliwości tworzenia jednej paczki, np. *jQuery validation* potrzebuje podstawowej biblioteki jQuery, przez co skrypty muszą być ładowane osobno w odpowiedniej kolejności.

Sekcje

Sekcje to części kodu, które mogą zostać umieszczone na stronie, lecz nie muszą. W głównym pliku z szablonem strony `_Layout.cshtml` znajduje się kilka sekcji (listing 5.14).

Listing 5.14. Kod pliku `_Layout.cshtml`

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="utf-8" />
    <title>@ViewBag.Title - My ASP.NET MVC Application</title>
    <link href("~/favicon.ico" rel="shortcut icon"
        type="image/x-icon" />
    <meta name="viewport" content="width=device-width" />
    <script src="http://google-code-
        prettify.googlecode.com/svn/trunk/src/prettify.js"
        type="text/javascript" ></script>
    @Styles.Render("~/Content/css")
    @Scripts.Render("~/bundles/modernizr")
</head>
<body onload='prettyPrint()'>
    <header>
        <div class="content-wrapper">
            <div id="head">
                <div class="float-right">
                    <section id="login">
                        @Html.Partial("_LoginPartial")
                    </section>
                </div>
            </div>
            <div id="menu_top">
                <nav>
                    <ul id="menu">
                        <li>@Html.ActionLink("Strona główna",
                            "Index", "Home")</li>
                        <li>@Html.ActionLink("O nas", "ONas",
                            "HomeTabs")</li>
                        <li>@Html.ActionLink("Kontakt", "Kontakt",
                            "HomeTabs")</li>
                        <li>@Html.ActionLink("Współpraca",
                            "Wspolpraca", "HomeTabs")</li>
                    </ul>
                </nav>
            </div>
        </div>
    </header>
    <div id="body">
        @RenderSection("featured", required: false)
        <section class="content-wrapper main-content clear-fix">
            @RenderBody()
        </section>
    </div>
    <footer>
        <div class="content-wrapper">
            <p>@DateTime.Now.Year - ASP.net-MVC.p1</p>
        </div>
    </footer>
```

```

        </footer>

        @Scripts.Render("~/bundles/jquery")
        @RenderSection("scripts", required: false)
    </body>
</html>

```

Treść strony WWW jest umieszczona w sekcji RenderBody() i jest to jedyna wymagana sekcja. Bez tej części kodu nie byłoby wiadomo, w którym miejscu ma się znajdować główna, zmieniająca się część strony. W kodzie (listing 5.15) znajduje się również druga sekcja, która nie jest wymagana (required: false), o nazwie featured.

Listing 5.15. Sekcja niewymagana

```

<div id="body">
    @RenderSection("featured", required: false)
    <section class="content-wrapper main-content clear-fix">
        @RenderBody()
    </section>
</div>

```

Aby sekcja o nazwie featured znalazła się na stronie, w pliku z widokiem trzeba dodać sekcję o tej samej nazwie (listing 5.16), a w niej kod, który ma się znaleźć we wcześniej wybranym miejscu (w pliku *_Layout.cshtml*).

Listing 5.16. Kod sekcji z pliku Index.cshtml dla kontrolera Home

```

@section featured {
    <section class="featured">
        <div class="content-wrapper">
            <hgroup class="title">
                <h1>@ViewBag.Title.</h1>
                <h2>@ViewBag.Message</h2>
            </hgroup>
            <p>
                Miło nam poinformować o powstaniu portalu.
            </p>
        </div>
    </section>
}

```

Kod HTML wygenerowany dla podstrony bez sekcji featured zaprezentowano na listingu 5.17, a wygląd strony przedstawiono na rysunku 5.11.

Listing 5.17. Kod HTML bez sekcji featured

```

<div id="body">
    <section class="content-wrapper main-content clear-fix">
        <!-- Tutaj treść strony
        Zobacz, czego się tutaj dowiesz: ....
        -->
    </section>
</div>

```

Rysunek 5.11.

*Wygląd strony
bez sekcji featured*



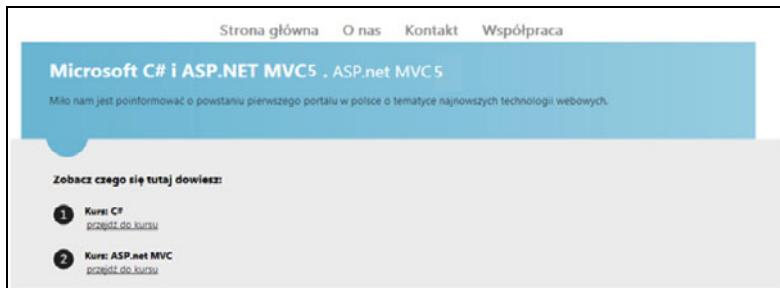
Kod HTML wygenerowany dla podstrony z sekcją featured zaprezentowano na listingu 5.18, a wygląd strony na rysunku 5.12.

Listing 5.18. Kod HTML z sekcją featured

```
<div id="body">
    <section class="featured">
        <div class="content-wrapper">
            <hgroup class="title">
                <h1>Microsoft C# i ASP.NET MVC 5.</h1>
                <h2>ASP.net MVC 5</h2>
            </hgroup>
            <p>Miło nam jest poinformować o powstaniu pierwszego
                ↗ portalu w Polsce o tematyce najnowszych technologii
                ↗ webowych. </p>
        </div>
    </section>
    <section class="content-wrapper main-content clear-fix">
        <!-- Tutaj treść strony
        Zobacz, czego się tutaj dowiesz: ....
        -->
    </section>
</div>
```

Rysunek 5.12.

*Wygląd strony
z sekcją featured*



Na samym końcu kodu (listing 5.14) znajduje się sekcja scripts. Powinny się w niej znajdować wszystkie skrypty JavaScript unikalne dla konkretnych podstron. Sekcja ze skryptami jest ładowana na samym końcu dokumentu, aby nie blokować ładowania się strony WWW. Skrypty wspólne dla każdej podstrony są ładowane w pliku *_Layout.cshtml* za pomocą `@Scripts.Render("~/bundles/jquery")`.

Sekcje są pomocne dla tych części podstron, które mogą wyglądać różnie w zależności od podstrony, a nie znajdują się w miejscu, gdzie ładowana jest treść strony (np. reklamy dostosowane do treści na podstronach).

Aby nie dodawać tego samego *diva* do wszystkich sekcji, można skorzystać z następującego rozwiązania:

```
@if(IsSectionDefined("featured")){
    <div id="jakas_klasa">
        @RenderSection("featured", required: false)
    </div>
}
```

Routing

Routing to tłumaczenie adresu URL na wykonanie pewnej akcji z kontrolera lub zwrócenie pliku bądź obrazu z podanej ścieżki. Aby przetłumaczyć adres podstrony na akcję zrozumiałą dla aplikacji, wykorzystywane są reguły routingu. Kod domyślnej reguły routingu wygląda następująco:

```
routes.MapRoute(
    name: "Default",
    url: "{controller}/{action}/{id}",
    defaults: new { controller = "Home", action = "Index",
        id = UrlParameter.Optional }
);
```

Domyślna reguła routingu przyjmuje trzy parametry. Pierwszy z nich to nazwa kontrolera, który ma obsłużyć żądanie, drugi to akcja z kontrolera, która ma się wykonać, a trzeci to parametr opcjonalny do danej akcji. Domyślne wartości są wykorzystywane przy odwiedzaniu strony głównej i ustawione są na kontroler Home oraz akcję Index.

Zasady routingu znajdują się w statycznej metodzie RegisterRoutes z klasy RouteConfig. Klasa ta znajduje się w folderze *App_Start*. Metoda RegisterRoutes zostaje wywołana przy starcie aplikacji.

Kolejność w routingu

Bardzo ważna jest kolejność dodawania zasad routingu. Zasady powinny być ułożone w odpowiedniej kolejności. Gdy adres URL nie pasuje do pierwszej zasady routingu, sprawdzana jest kolejna zasada. Jeśli zostanie znaleziona odpowiednia zasada, przeszukiwanie się zakończy. Ścieżki z atrybutów są sprawdzane jako pierwsze.

Ignorowanie ścieżek

W domyślnej konfiguracji routingu w klasie RouteConfig na samym początku metody RegisterRoutes() znajduje się następujący kod:

```
routes.IgnoreRoute("{resource}.axd/{*pathInfo}");
```

Metoda IgnoreRoute() służy do oznaczania ścieżek, które nie powinny być tłumaczone ani przechwytywane poprzez zasady routingu.

W przypadku gdy nie chcesz, aby żądanie o favicon przechodziło poprzez mechanizm routingu, trzeba dodać następującą linię:

```
routes.IgnoreRoute("{*favicon}", new { favicon = @"(.*/)?favicon.ico(/.*?)" });
```

Gdy nie chcesz, aby żądania o zdjęcia i inne pliki z folderu *Content* przechodziły poprzez routing, dodaj następującą linię:

```
routes.IgnoreRoute("{Content}/{*pathInfo}");
```

Reguły ignorowania muszą się znajdować na samym początku — przed innymi regułami, ponieważ routing wybiera pierwszą pasującą do żądania regułę.

Ograniczenia

Aby zasady routingu miały sens, niezbędne jest tworzenie ograniczeń. Do ograniczeń stosowane są wyrażenia regularne. Możliwe jest ograniczenie danej zasady tylko do jednego kontrolera, np. *HomeController*. Tworzenie własnej zasady routingu przebiega następująco:

```
routes.MapRoute(  
    "Default",  
    "{controller}/{action}/{id}",  
    new { controller = "Home", action = "Index",  
        id = UrlParameter.Optional },  
    new { controller = "^H.*", action = "^Index$|^About$" }  
    // Ograniczenia  
>);
```

Utworzona zasada odnosi się tylko i wyłącznie do akcji *Index* lub *About* w kontrolerach, których nazwa zaczyna się literą H. W przeciwnym wypadku zostaną wykorzystane inne „routy”. Usunięte zostały nazwy parametrów: *name*, *url* itp. Ścieżki działają tak samo jak z nazwami parametrów, a przy pisaniu dużej liczby reguł jest to znaczące uproszczenie.

Routing na podstawie atrybutów

Routing na podstawie atrybutów (ang. *Attribute Routing*) to nowość wprowadzona w MVC 5. Aby włączyć *Attribute Routing*, należy wywołać metodę *MapMvcAttributeRoutes()* w pliku konfiguracyjnym z regułami routingu.

Oto opcjonalne parametry w atrybutach:

```
[Route("ogloszenia/{id?}")]  
public ActionResult View(string id)
```

A to domyślne wartości parametrów:

```
[Route("ogloszenia/lang/{lang=pl}")]
```

Prefiksy

Prefiks pozwala na ustalenie powtarzającej się części ścieżki dla całego kontrolera w jednym miejscu. Prefiks dla całego kontrolera:

```
[RoutePrefix("ogloszenie")]
public class OgloszenieController : Controller
{
    //Np. /ogloszenie
    [Route]
    public ActionResult Index() { ... }
    //Np. /ogloszenie/2
    [Route("{ogId}")]
    public ActionResult Szczegoly(int ogId) { ... }
    //Np. /ogloszenie/2/edytuj
    [Route("{ogId}/edit")]
    public ActionResult Edytuj(int ogId) { ... }
}
```

oznacza to samo co:

```
public class OgloszenieController : Controller
{
    //Np. /ogloszenie
    [Route("{ogloszenie}")]
    public ActionResult Index() { ... }
    //Np. /ogloszenie/2
    [Route("{ogloszenie}/{ogId}")]
    public ActionResult Szczegoly(int ogId) { ... }
    //Np. /ogloszenie/2/edytuj
    [Route("{ogloszenie}/{ogId}/edit")]
    public ActionResult Edytuj(int ogId) { ... }
}
```

Tak przebiega nadpisywanie prefiku:

```
[RoutePrefix("ogloszenie")]
public class OgloszenieController : Controller
{
    //Np. /polecane-ogloszenie
    [Route("~/polecane-ogloszenie")]
    public ActionResult PokazPolecane() { ... }
    ...
}
```

Domyślna ścieżka dla kontrolera to:

```
[RoutePrefix("ogloszenie")]
[Route("{action=index1}")]
public class OgloszenieController : Controller
{
    //Np. /ogloszenie
    public ActionResult Index1() { ... }
}
```

Ograniczenia

Poniżej zaprezentowano składnię ograniczeń w *Attribute Routing* (tabela 5.1):

```
{parametr:constraint}
```

A to przykład zastosowania ograniczeń:

```
[Route("users/{id:int}")]
public ActionResult GetUserById(int id) { ... }

// Np. users/user
[Route("users/{name}")]
public ActionResult GetUserByName(string name) { ... }
```

Tabela 5.1. Tabela ograniczeń

Ograniczenie	Opis	Przykład użycia
datetime	Wartości DateTime	{x:datetime}
decimal	Wartości decimal	{x:decimal}
double	Wartości 64-bitowe floating-point	{x:double}
Float	Wartości 32-bitowe floating-point	{x:float}
Guid	Wartości GUID	{x:guid}
Int	Wartości 32-bitowe integer	{x:int}
Length	Wartości string o określonej długości lub w określonym przedziale długości	{x:length(6)} {x:length(6,12)}
Long	Wartości 64-bitowe integer	{x:long}
Max	Wartości integer z maksimum	{x:max(10)}
maxlength	Wartości string z maksimum	{x:maxlength(10)}
Min	Wartości integer z minimum	{x:min(10)}
minlength	Wartości string z minimum	{x:minlength(10)}
Range	Wartości integer w określonym przedziale wartości	{x:range(10,50)}
Regex	Wyrażenia regularne	{x:regex(^\\d{} - \\d{3}\$)}

Nazywanie ścieżek i generowanie linków po nazwie ścieżki

Poniżej pokazano przykład nazywania ścieżek:

```
[Route("Ogloszenie", Name = "Ogloszenia")]
public ActionResult Ogloszenia() { ... }
```

Tworzenie linku na podstawie nazwy ścieżki przebiega następująco:

```
<a href="@Url.RouteUrl("Ogloszenia")>Ogloszenia</a>
```

Obszary

Obszary (ang. *Areas*) w *Attribute Routing* pozwalają na odseparowanie części aplikacji w osobnym folderze (projekt w projekcie). Wykorzystywane są przy dużych aplikacjach, aby uporządkować kod. Zazwyczaj panel administratora zostaje przeniesiony

do osobnego obszaru. Ponieważ kontrolery i widoki mogą mieć takie same nazwy, co może powodować problemy podczas wyboru kontrolera, należy skonfigurować routing dla danego obszaru.

Ustawienie obszaru dla ścieżki przebiega następująco:

```
[RouteArea("Admin")]
[RoutePrefix("ogloszenie")]
[Route("{action}")]
public class OgloszenieController : Controller
{
    // Np. /admin/ogloszenie/login
    public ActionResult Login() { ... }

    // Np. /admin/ogloszenie/pokaz-opcje
    [Route("pokaz-opcje")]
    public ActionResult PokazOpcje() { ... }

    // Np. /statystyki
    [Route("~/statystyki")]
    public ActionResult Statystyki() { ... }
}
```

Kod generujący linki z wykorzystaniem obszarów:

```
Url.Action("PokazOpcje", "Ogloszenie", new { Area = "Admin" })
```

wygeneruje link:

```
/Admin/Ogloszenie/pokaz-opcje
```

Oto prefiksy dla obszarów:

```
[RouteArea("PanelAdmina", AreaPrefix = "panel-admina")]
```

Aby włączyć obsługę obszarów, należy wywołać następującą metodę w metodzie Application_Start() z pliku global.asax:

```
AreaRegistration.RegisterAllAreas();
```

Model

Za warstwę modelu odpowiada zazwyczaj framework ORM. Dla ASP.NET MVC domyślnym ORM-em jest Entity Framework.

ViewModel

ViewModel to klasa modelu tworzona na potrzeby widoku. Jeśli masz np. listę ofert przejazdów bądź kursów dodawanych przez użytkowników i chcesz na liście z ofertami przejazdów wyświetlić również podstawowe informacje o dodającym, musisz mieć w bazie danych dwie tabele (kurs i user) połączone relacjami. Do widoku można przekazać tylko jeden model domenowy. Może to być model z klasą Kurs lub klasą User

(każda tabela posiada własną klasę z modelem). W takiej sytuacji niezbędne jest utworzenie klasy KursUserViewModel (listing 5.21) łączącej potrzebne dane z klasy Kurs (listing 5.19) i klasy User (listing 5.20). Klasa ViewModel nie zawiera wszystkich pól, jakie są dostępne w klasach, które łączy — powinna zawierać tylko te dane, które są potrzebne do widoku. Klasa ViewModel niekoniecznie musi łączyć kilka tabel z bazy danych, może również obejmować tylko część pól z podstawowego modelu, np. tylko imię i nazwisko, przy czym w tabeli istnieje większa liczba pól. Dla klas ViewModel należy określić reguły validacji, podobnie jak dla zwykłych klas z modelem.

Listing 5.19. Przykładowy kod klasy Kurs

```
public partial class Kurs
{
    public Kurs()
    {
        this.KursUser = new HashSet<KursUser>();
    }

    public int Id { get; set; }
    public bool Typ { get; set; }
    public string Skad { get; set; }
    public string Dokad { get; set; }
    public System.DateTime DataWyjazdu { get; set; }
    public System.DateTime DataDodania { get; set; }

    public virtual ICollection<KursUser> KursUser { get; set; }
}
```

Listing 5.20. Przykładowy kod klasy User

```
public class User
{
    public User()
    {
        this.KursUser = new HashSet<KursUser>();
    }

    public int UserId { get; set; }

    [EmailAddress]
    [DisplayName("Nazwa użytkownika")]
    public string UserName { get; set; }

    [Phone]
    public string Telefon { get; set; }

    public System.DateTime DataRej { get; set; }

    public virtual ICollection<KursUser> KursUser { get; set; }
}
```

Listing 5.21. Przykładowy kod klasy KursUserViewModel

```
public class KursUserViewModel
{
    [EmailAddress]
    public string UserName { get; set; }

    [Required]
    public string Skad { get; set; }

    [Required]
    public string Dokad { get; set; }

    [DataType(DataType.Date)]
    public System.DateTime DataWyjazdu { get; set; }
}
```

Klasa przedstawiona na listingu 5.21 łączy tylko niektóre pola z dwóch klas. W tym przypadku klasy nie są na tyle rozbudowane, aby była konieczność wybiorcze pobierania określonych pól. Dlatego można wykorzystać inne podejście. Klasa KursUserViewModel będzie łączyć wszystkie pola z jednej i drugiej klasy. W tym celu ViewModel po prostu będzie się składał z dwóch poprzednich modeli domenowych (Kurs i User) (listing 5.22).

Listing 5.22. Kod klasy KursUserViewModel łączącej dwie klasy

```
public class KursUserViewModel
{
    public Kurs kurs { get; set; }
    public User user { get; set; }
}
```

Aby pobrać dane do KursUserViewModel, trzeba pobrać listę kursów. Kursy pobiera się za pomocą metody pomocniczej GetLast() wykorzystującej zapytanie LINQ. Po pobraniu kursów w pętli foreach dla każdego kursu pobiera się odpowiadającego mu użytkownika i dodaje do listy obiektów z ViewModel (listing 5.23). Dzięki takiej operacji otrzymujemy listę obiektów KursUserViewModel, których dane są połączeniem dwóch tabel w bazie danych. Kod widoku dla utworzonego ViewModel prezentuje listing 5.24.

Listing 5.23. Pobieranie danych do ViewModel

```
public IQueryble<Kurs> GetLast(int naStrone, int strona)
{
    return _db.Set<Kurs>().OrderByDescending(x => x.DataDodania);
}

public IList<KursUserViewModel> GetKursUserViewList(int naStrone, int strona)
{
    List<KursUserViewModel> lista = new List<KursUserViewModel>();
    IList<Kurs> kursy = GetLast(naStrone, strona).ToList<Kurs>();
    foreach (var k in kursy)
    {
        KursUserViewModel vm = new KursUserViewModel();
        vm.kurs = k;
```

```
        vm.user = (from u in _db.Set<User>()
            ↗join c in _db.Set<KursUser>() on u.UserId equals
            ↗c.UserId where c.KursId == k.Id
            ↗select u).FirstOrDefault();
        lista.Add(vm);
    }
    return lista;
}
```

Listing 5.24. Kod widoku

```
@model IEnumerable<CoreModel.ViewModels.KursUserViewModel>
 @{
    ViewBag.Title = "Lista ofert";
}
<h2>Lista ofert</h2>
<p>
    @Html.ActionLink("Dodaj nową trasę", "Dodaj")
</p>
<table>
<tr>
    <th>
        @Html.DisplayNameFor(model => model.user.UserName)
    </th>
    <th>
        @Html.DisplayNameFor(model => model.kurs.Skad)
    </th>
    <th>
        @Html.DisplayNameFor(model => model.kurs.Dokad)
    </th>
    <th>
        @Html.DisplayNameFor(model => model.kurs.DataWyjazdu)
    </th>
</tr>
@foreach (var item in Model) {
    <tr>
        <td>
            @Html.DisplayFor(modelItem => item.user.UserName)
        </td>
        <td>
            @Html.DisplayFor(modelItem => item.kurs.Skad)
        </td>
        <td>
            @Html.DisplayFor(modelItem => item.kurs.Dokad)
        </td>
        <td>
            @Html.DisplayFor(modelItem => item.kurs.DataWyjazdu)
        </td>
    </tr>
}
</table>
```

Walidacja

Walidacja danych jest domyślnie zaimplementowana jako adnotacje (atrybuty) do pól w klasach z modelem. Walidacja odbywa się po stronie serwera lub po stronie klienta za pomocą biblioteki jQuery Unobtrusive. Walidacja po stronie serwera wymaga odświeżenia strony, a walidacja po stronie klienta nie. Walidacja po stronie serwera może zostać również zaimplementowana przy użyciu technologii AJAX, co pozwala uniknąć konieczności odświeżania strony, jednak podczas sprawdzania poprawności będą wysyłane osobne żądania do serwera dla każdego pola — z tego względu lepszym rozwiązaniem jest walidacja po stronie klienta. Walidacja po stronie serwera i tak zawsze zostanie wywołana, bez względu na to, czy została uruchomiona walidacja po stronie klienta.

Aby uruchomić walidację po stronie klienta, należy umieścić następujący wpis w pliku *web.config*:

```
<add key="ClientValidationEnabled" value="true" />
```

Oto przykładowe adnotacje do walidacji:

```
[Required(ErrorMessage = "Pole wymagane")]
[Range(0.01, 100.00, ErrorMessage =
    →"Wymagana wartość pomiędzy 0.01 a 100")]
[RegularExpression(@"^[A-Z]+[a-zA-Z'-`\s]*$")]
[Compare("NewPassword", ErrorMessage = "Hasła nie są takie same.")]
[DataType(DataType.ImageUrl)]
[StringLength(1024)]
```

Przykład walidacji dla pola *E-mail*:

```
[Required]
[EmailAddress(ErrorMessage="Wartość w polu E-mail nie jest
    →prawidłowym adresem e-mail")]
[Display(Name = "E-mail")]
public string Email { get; set; }
```

Kod widoku odpowiedzialny za wyświetlenie formularza oraz walidację:

```
@Html.TextBoxFor(model => model.Email)
@Html.ValidationMessageFor(model => model.Email)
```

Komunikat wyświetlany po wpisaniu nieprawidłowych danych w polu *E-mail* zaprezentowano na rysunku 5.13. Treść komunikatu to wartość pola *ErrorMessage* z atrybutu.

Rysunek 5.13.

Komunikat o błędzie



MVC Scaffolding

MVC Scaffolding to bardzo pomocne narzędzie automatycznie generujące kod na podstawie klas z modelem. Nie ma potrzeby pisania kodu, który może zostać automatycznie wygenerowany przez Visual Studio. Do generowania wykorzystywane są szablony. Szablony można samodzielnie modyfikować według własnych potrzeb.

Aby nadpisać standardowe szablony T4 wykorzystywane przy generowaniu kontrolerów i widoków, należy umieścić w projekcie folder *CodeTemplates*, w którym znajdują się wszystkie pliki T4. Folder *CodeTemplates* znajduje się w lokalizacji: *C:\Program Files (x86)\Microsoft Visual Studio 12.0\Common7\IDE\Extensions\Microsoft\Web\Mvc\Scaffolding\Templates*.

Generowanie kontrolerów

Na podstawie modelu można wygenerować kontroler z domyślnie zaimplementowanymi akcjami odpowiadającymi za operacje CRUD (ang. *Create, Read, Update, Delete*).

Oto kod klasy z modelem, na podstawie której będzie generowany kontroler:

```
[Bind(Exclude = "DataDodania")]
[Table("Kurs", Schema = "dbo")]
public partial class Kurs
{
    public Kurs()
    {
        this.KursUser = new HashSet<KursUser>();
    }
    [Key]
    [ScaffoldColumn(false)]
    [DatabaseGeneratedAttribute
     ↳(DatabaseGeneratedOption.Identity)]
    public int Id { get; set; }

    [DisplayName("Osoba prywatna:")]
    public bool Typ { get; set; }

    [Required]
    [DisplayName("Z:")]
    public string Skad { get; set; }

    [Required]
    [DisplayName("Do:")]
    public string Dokad { get; set; }

    [DataType(DataType.Date)]
    [DisplayName("Kiedy:")]
    public System.DateTime DataWyjazdu { get; set; }

    [DataType(DataType.Date)]
    [DisplayName("Data dodania:")]
    [ScaffoldColumn(false)]
```

```

public System.DateTime DataDodania { get; set; }

public virtual ICollection<KursUser> KursUser { get; set; }
}

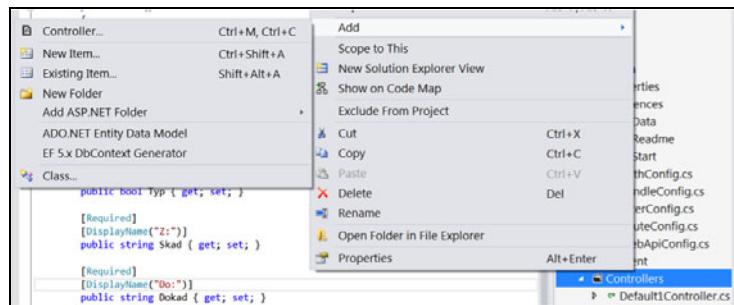
```

Aby dodać kontroler, kliknij folder prawym przyciskiem myszy i z menu kontekstowego wybierz opcję *Add/Controller...* (rysunek 5.14).

Otworzy się okno, w którym trzeba podać nazwę kontrolera oraz szablon, na podstawie którego ma być wygenerowany kontroler.

Rysunek 5.14.

Dodawanie kontrolera



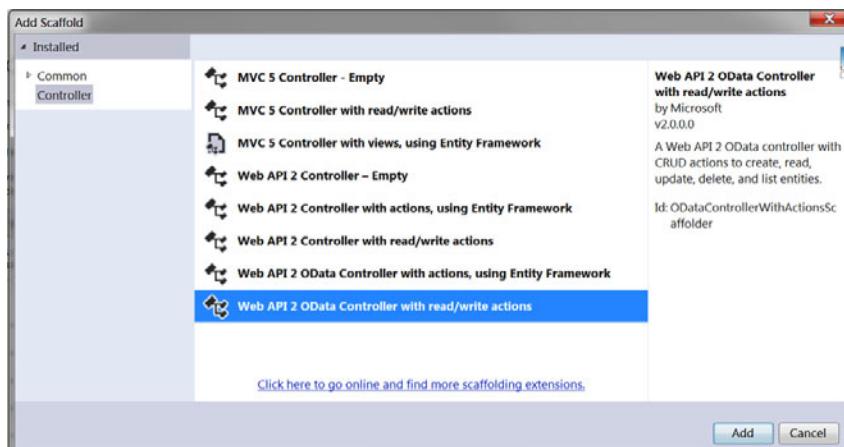
Do wyboru są szablony (rysunek 5.15):

- ◆ *MVC 5 Controller Empty* — pusty kontroler;
- ◆ *MVC 5 Controller with read/write actions* — kontroler ze szkieletami funkcji, bez implementacji funkcjonalności;
- ◆ *MVC 5 Controller with Views, using Entity Framework* — kontroler zaimplementowanymi operacjami CRUD;
- ◆ *Web API 2 Controller Empty* — pusty kontroler dla Web API;
- ◆ *Web API 2 Controller with actions, using Entity Framework* — kontroler Web API z zaimplementowanymi operacjami za pomocą EF;
- ◆ *Web API 2 Controller with read/write actions* — kontroler Web API ze szkieletami funkcji, bez implementacji funkcjonalności;
- ◆ *Web API 2 OData Controller with actions, using Entity Framework* — kontroler Web API z zaimplementowanymi operacjami za pomocą EF, korzystający z OData;
- ◆ *Web API 2 OData Controller with read/write actions* — kontroler Web API ze szkieletami funkcji, bez implementacji funkcjonalności, korzystający z OData.

Wybierz opcję *MVC 5 kontroler z zaimplementowanymi operacjami CRUD* za pomocą EF. Aby mechanizm Scaffoldingu wiedział, dla której klasy z modelem ma wygenerować kontroler, wybierz go z listy rozwijanej *Model class*. Kolejną opcję do wyboru jest obiekt dostępu do bazy danych — kontekst, z którego ma korzystać kontroler (rysunek 5.16).

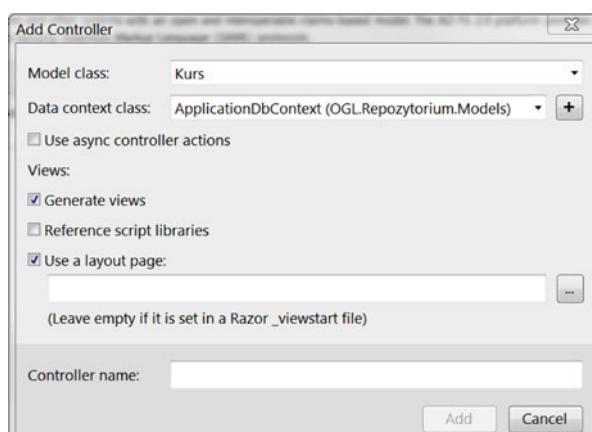
Można zaznaczyć opcję, która utworzy asynchroniczne metody w kontrolerze oraz pozwoli wybrać, czy podczas generowania mają zostać utworzone widoki dla każdej akcji.

Kliknij przycisk *Add*, aby wygenerować kontroler i widoki. Nowo wygenerowany kontroler zawiera zaimplementowane metody przedstawione na rysunku 5.17.

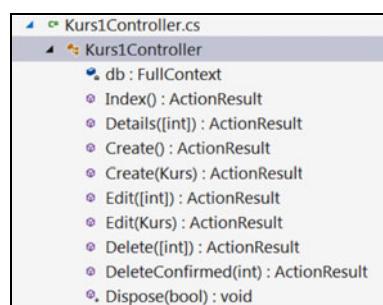


Rysunek 5.15. Scaffold templates — szablony

Rysunek 5.16.
Wybór klasy modelu
oraz kontekstu



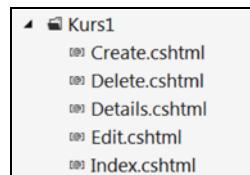
Rysunek 5.17.
Schemat
wygenerowanego
kontrolera



W folderze *Views* zostaje utworzony folder o tej samej nazwie co kontroler *Kurs1* i pięć widoków dla operacji CRUD (rysunek 5.18).

Rysunek 5.18.

Wygenerowane widoki



Podczas generowania utworzone zostały kontroler oraz widoki w folderze, którego nazwa jest taka sama jak nazwa kontrolera (listing 5.25).

Listing 5.25. Wygenerowany kod widoku Index

```
@model IEnumerable<CoreModel.Models.Kurs>

 @{
    ViewBag.Title = "Index";
}
<h2>Index</h2>
<p>
    @Html.ActionLink("Create New", "Create")
</p>
<table>
    <tr>
        <th>
            @Html.DisplayNameFor(model => model.Typ)
        </th>
        <th>
            @Html.DisplayNameFor(model => model.Skad)
        </th>
        <th>
            @Html.DisplayNameFor(model => model.Dokad)
        </th>
        <th>
            @Html.DisplayNameFor(model => model.DataWyjazdu)
        </th>
        <th></th>
    </tr>
    @foreach (var item in Model) {
        <tr>
            <td>
                @Html.DisplayFor(modelItem => item.Typ)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.Skad)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.Dokad)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.DataWyjazdu)
            </td>
            <td>
```

```

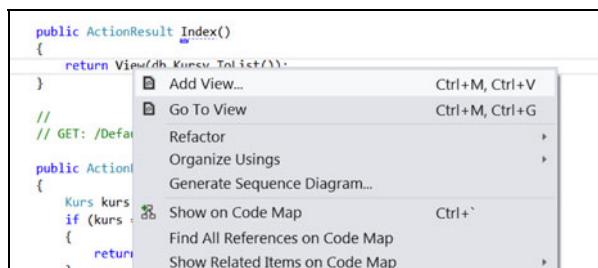
@Html.ActionLink("Edit", "Edit", new { id=item.Id }) |
@Html.ActionLink("Details", "Details",
    new { id=item.Id }) |
@Html.ActionLink("Delete", "Delete", new { id=item.Id })
</td>
</tr>
}
</table>

```

Generowanie widoków

W momencie generowania kontrolera na podstawie modelu automatycznie wygenerowane zostały również widoki. Jeśli masz kontroler i chcesz dodać tylko jeden widok określonego typu, kliknij prawym przyciskiem myszy w oknie z kontrolerem i wybierz opcję *Add View...* (rysunek 5.19).

Rysunek 5.19.
Dodawanie widoku



Zostanie otwarte okno dialogowe, w którym wpisz nazwę widoku, po czym wybierz silnik widoku — Razor. Następnie jest do wyboru opcja *strongly-typed view*, która pozwala na wygenerowanie widoku dla konkretnego modelu. Wybierz klasę z modelem (*Kurs1*). Kolejnym krokiem będzie wybranie szablonu dla widoku (*Scaffold template*).

Do wyboru jest sześć szablonów widoków (rysunek 5.20):

- ◆ *Create* — widok dla operacji dodawania;
- ◆ *Delete* — widok dla operacji usuwania;
- ◆ *Details* — widok wyświetlający szczegóły o kursie;
- ◆ *Edit* — widok dla operacji edytowania;
- ◆ *Empty* — pusty widok;
- ◆ *List* — lista wszystkich kursów z modelem.

Po wyborze szablonu do zaznaczenia pozostaje opcja *Reference script libraries*. Określa ona, czy do widoku mają zostać dołączone skrypty jQuery do walidacji. Jeśli skrypty są ładowane w sekcji `<head>` w widoku `_Layout.cshtml`, należy usunąć zaznaczenie tej opcji.

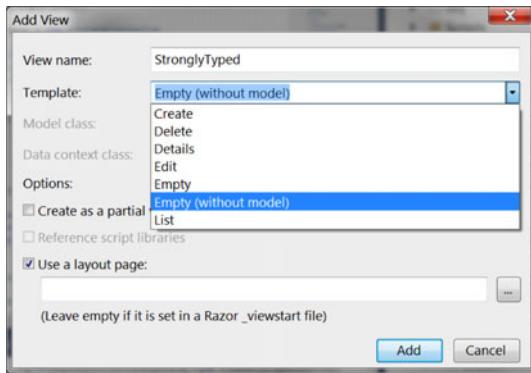
Oto kod dodany do widoku po zaznaczeniu opcji *Reference script libraries*:

```
<script src="@Url.Content("~/Scripts/jquery.validate.min.js")"
    type="text/javascript"></script>

<script src="@Url.Content("~/Scripts/jquery.validate.unobtrusive.min.js")" type="text/javascript"></script>
```

Rysunek 5.20.

Okno wyboru szablonu widoku

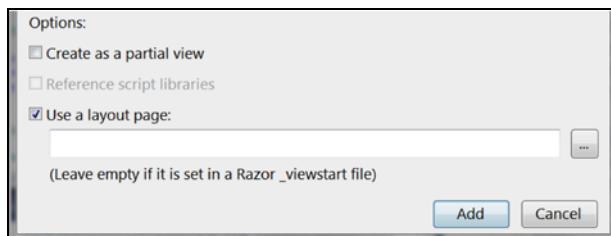


Na samym końcu wybiera się jedną z dwóch opcji (rysunek 5.21):

- ◆ *Create as a partial view* — tworzy widok częściowy (*Partial*), czyli dla danego widoku nie zostanie załadowany szablon strony i zostanie zwrócona tylko część z danego widoku;
- ◆ *Use a layout page* — wybiera się, jaki szablon strony ma być załadowany (domyślnie jest to *_Layout.cshtml*).

Rysunek 5.21.

Okno wyboru typu widoku



Wygląd widoku wygenerowanego jako *Partial View* prezentuje rysunek 5.22. Ten sam widok wygenerowany jako zwykły widok pokazano na rysunku 5.23.

Metody synchroniczne i asynchroniczne w MVC

W .NET Framework do każdego żądania ASP.NET jest przypisywany jeden wątek. Jeśli żądanie jest przetwarzane synchronicznie, to wątek zostaje zablokowany do czasu wykonania tego żądania. Liczba wątków w puli jest ograniczona i w .NET 4.5 jest to

Lista ofert

[Dodaj nową tase](#)

Nazwa Użytkownika:	Z:	Do:	Kiedy:
	wszystko@gmail.com Lutoryż, Polska	Lubcza, Polska	2013-09-29 Szczegóły
	BVNW, Overland Park, KS	BN, Italia	2013-09-21 Szczegóły
	Nva Presidente Riesco, Santiago, Chile	Vbl, Lucerne, Suisse	2013-09-28 Szczegóły
	Coventry CV3, United Kingdom	Coventry CV1, United Kingdom	2013-09-15 Szczegóły
bl@bl.pl	Gfomergasse, 1060 Wiedeń, Austria	Głowiackiego 111, 37-400 Nisko, Polska	2013-09-21 Szczegóły
bl@bl.pl	GFF, Chemin de Crèvecœur, Saint-Denis, France	GF Strong Rehabilitation Centre, Laurel Street, Vancouver, BC, Canada	2013-09-15 Szczegóły
	Calle Haendel, 1-3, 29004 Málaga, Prowincja Malaga, Hiszpania	Barriada Molina, 39, 29130 Alhaurín de la Torre, Prowincja Malaga, Hiszpania	2013-09-15 Szczegóły
	A-366, 29109 Tolox, Prowincja Malaga, Hiszpania	MA-415, 29591 Málaga, Prowincja Malaga, Hiszpania	2013-09-07 Szczegóły
	BVG, Potsdamer Straße, Schöneberg, Deutschland	VB, Italia	2013-09-21 Szczegóły
	H.J.E. Wenckebachweg, De Omval, Amsterdam, Nederland	Ghana	2013-09-08 Szczegóły

Rysunek 5.22. Widok Partial View

Szukam-Transportu.pl lub [S-T.pl](#)
[Zarejestruj się] lub [Zaloguj]

Szukaj transportu
Dodaj przejazd
Kontakt
O Nas
Przeglądaj Mapę
Najnowsze oferty

Lista ofert

[Dodaj nową tase](#)

Nazwa Użytkownika:	Z:	Do:	Kiedy:
	wszystko@gmail.com Lutoryż, Polska	Lubcza, Polska	2013-09-29 Szczegóły
	BVNW, Overland Park, KS	BN, Italia	2013-09-21 Szczegóły
	Nva Presidente Riesco, Santiago, Chile	Vbl, Lucerne, Suisse	2013-09-28 Szczegóły
	Coventry CV3, United Kingdom	Coventry CV1, United Kingdom	2013-09-15 Szczegóły
bl@bl.pl	Gfomergasse, 1060 Wiedeń, Austria	Głowiackiego 111, 37-400 Nisko, Polska	2013-09-21 Szczegóły
bl@bl.pl	GFF, Chemin de Crèvecœur, Saint-Denis, France	GF Strong Rehabilitation Centre, Laurel Street, Vancouver, BC, Canada	2013-09-15 Szczegóły
	Calle Haendel, 1-3, 29004 Málaga, Prowincja Malaga, Hiszpania	Barriada Molina, 39, 29130 Alhaurín de la Torre, Prowincja Malaga, Hiszpania	2013-09-15 Szczegóły
	A-366, 29109 Tolox, Prowincja Malaga, Hiszpania	MA-415, 29591 Málaga, Prowincja Malaga, Hiszpania	2013-09-07 Szczegóły
	BVG, Potsdamer Straße, Schöneberg, Deutschland	VB, Italia	2013-09-14 Szczegóły

Rysunek 5.23. Widok zwykły razem z layoutem

maksymalnie 5000. Każdy wątek zajmuje ok. 1 MB pamięci, co przy 5000 daje 5 GB na samo uruchomienie wątków. Wątek jest blokowany aż do momentu, gdy otrzyma odpowiedź na żądanie. Gdy klient nie odpowiada, wątek czeka na odpowiedź i blokuje wykonanie innych żądań. Aby zabezpieczyć aplikację przed blokowaniem wątków,



należy używać metod asynchronicznych. Jeśli żądanie jest asynchroniczne, jeden wątek może obsługiwać wiele żądań równocześnie.

Aby maksymalnie uprościć korzystanie z metod asynchronicznych, w .NET 4.5 zostały wprowadzone dwa nowe słowa kluczowe: `async` i `await`.

Metody synchroniczne stosuje się:

- ◆ dla prostych, szybkich operacji, które nie powodują blokowania wątków;
- ◆ gdy prostota jest ważniejsza niż efektywność;
- ◆ dla operacji wykorzystujących w większym stopniu procesor (metody asynchroniczne nie obniżają zużycia procesora, wręcz przeciwnie — zwiększą jego zużycie).

Metody asynchroniczne stosuje się:

- ◆ gdy brakuje wolnych wątków;
- ◆ gdy aplikacja będzie szybciej/lepiej działać przy użyciu metod asynchronicznych;
- ◆ gdy operacja opiera się w głównej mierze na operacjach I/O (zapis/odczyt) na dysku lub na operacjach sieciowych, które mogą zablokować wątek poprzez wolne działanie;
- ◆ chce się zaimplementować mechanizm zatrzymywania lub przerwania długich żądań (np. przesyłanie plików).

Słowa kluczowe — Async, Await, Task

`Async` — tym słowem kluczowym oznacza się metodę asynchroniczną. Dzięki temu kompilator wie, że będzie to metoda asynchroniczna. Metoda asynchroniczna według konwencji powinna mieć na końcu nazwy słowo `Async`.

`Task` — jest to typ danych zwracanych przez metodę asynchroniczną. Typ `Task<typ_zwracany>` informuje kompilator o danych zwracanych asynchronicznie.

`Await` — informuje o metodzie, która czeka na dane zwracane w sposób asynchroniczny. Słowo `Await` może być użyte tylko w metodzie asynchronicznej oznaczonej słowem `Async`. Dzięki słowu `Await` kompilator nie blokuje wątku ani wykonywania innych części kodu na czas pobierania danych. Gdy dane z metody oznaczonej jako `Await` zostaną pobrane, wykonywane są kolejne operacje, które czekały na te dane. Pozostałe operacje były wykonywane równocześnie z pobieraniem danych. Przykładem może być wyświetlenie i załadowanie danych do formularza. Gdy metoda jest synchroniczna, dane są ładowane, a dopiero później uruchamiane jest okno z formularzem. Gdy metoda jest asynchroniczna, okno zostaje załadowane i jest responsywne (można np. zmieniać wielkość okna), podczas gdy w tle są pobierane dane. Po zakończeniu pobierania danych zostanie wykonana pozostała część kodu odpowiedzialna za wyświetlenie danych w formularzu¹¹.

¹¹ <http://www.asp.net/mvc/tutorials/mvc-4/using-asynchronous-methods-in-aspnet-mvc-4>

Cache

Cache w tłumaczeniu na język polski oznacza „pamięć przechowującą dane po to, by dostęp do nich był szybszy”. Cache służy do przechowywania danych w celu ich późniejszego wykorzystania bez konieczności obliczania lub przesyłania. W zależności od miejsca, w którym dane są przetwarzane, można wyróżnić dwie główne techniki cachowania: po stronie klienta w przeglądarce oraz po stronie serwera w pamięci.

Cachowanie po stronie serwera — Server Side Caching

Cachowanie po stronie serwera ma na celu jak najszybsze zwrócenie danych do klienta przy jak najmniejszym koszcie zużycia zasobów serwera (pamięci i procesora).

Cache po stronie serwera można podzielić na zakresy:

- ◆ cache dla żądania HTTP — dane zapisuje się na czas żądania:

```
HttpContext.Items["PierwszeZadanie"] = true;
```
- ◆ cache dla użytkownika — dane są zapisywane w sesji i są dostępne tylko dla jednego użytkownika:

```
HttpContext.Session["username"] = "Login";
```
- ◆ cache dla aplikacji — dane dostępne dla całej aplikacji:

```
Application["DataUruchomienia"] = System.DateTime.Now;
```

Metody te są na najniższym poziomie i są rzadko stosowane. W ASP.NET istnieje biblioteka *System.Web.Cache* ułatwiająca pracę z cache. Pozwala ona na zarządzanie:

- ◆ czasem żywotności — jak długo dane zapisane w cache będą aktualne, np. 20 sekund;
- ◆ zależnościami — pomiędzy danymi w cache a innymi elementami aplikacji, np. plikami;
- ◆ pamięcią — wykorzystywaną na cache (gdy brakuje pamięci, część najrzadziej wykorzystywanych danych zostaje usunięta z cache).

Dostępne rodzaje zależności:

- ◆ *Aggregate* — usuwa dany element z cache, gdy zajdą zmiany w powiązanych elementach;
- ◆ *Custom* — usuwa z cache w zależności od własnej klasy;
- ◆ *File* — usuwa powiązany element z cache, gdy zajdą zmiany w pliku lub plik zostanie skasowany;
- ◆ *Key* — usuwa element z cache, gdy zostanie skasowany inny element z cache;

- ◆ *SQL* — usuwa element z cache, gdy zostaną wprowadzone zmiany w określonej tabeli w bazie danych.

Atrybut OutputCache

OutputCache to najczęściej stosowana technika cachowania opierająca się na zapisywaniu wyniku żądania, czyli najczęściej widoku (kodu HTML). W ASP.NET MVC cachowanie wyjściowe dostępne jest poprzez wspomniany przy okazji kontrolerów filtr akcji OutputCache. Aby użyć OutputCache, dodaje się atrybut przed akcją w kontrolerze:

```
[OutputCache(Duration = 10, VaryByParam = "none")]
public ActionResult Index()
{
    return View();
}
```

Atrybut OutputCache posiada następujące parametry:

- ◆ CacheProfile — wykorzystuje profil cachowania, czyli pewną ogólną zasadę cachowania zdefiniowaną w pliku *web.config* (listing 5.26) — pozwala zmienić np. czas cachowania wielu akcji dzięki zmianie tylko jednej wartości w pliku konfiguracyjnym:

```
[OutputCache(CacheProfile = "NazwaProfilu")]
public ActionResult Index()
{
    return View();
}
```

Listing 5.26. Tworzenie profilu w *web.config*

```
<caching>
    <outputCacheSettings>
        <outputCacheProfiles>
            <add name="NazwaProfilu" duration="3600"
                varyByParam="id"/>
        </outputCacheProfiles>
    </outputCacheSettings>
</caching>
```

- ◆ Duration — czas, przez jaki dane mają być cachowane:

```
[OutputCache(Duration = 1)]
public ActionResult Index()
{
    return View();
}
```

- ◆ NoStore — brak cachowania (np. wyłączenie cachowania dla jednej akcji).
- ◆ Location — określa, gdzie dane mają być cachowane. Dostępne są następujące opcje:

- ◆ Client — po stronie klienta:

```
[OutputCache(Location=OutputCacheLocation.Client)]
public ActionResult Index()
```

```
{  
    return View();  
}  
◆ Any — automatycznie wybrana opcja,  
◆ Server — po stronie serwera,  
◆ Server and Client — po stronie serwera i klienta,  
◆ Downstream — gdzie indziej niż serwer aplikacji,  
◆ None — brak cachowania.  
◆ SqlDependency — cachowanie zależne od zmian w bazie danych.  
◆ VaryByContentEncoding — cachowanie zależne od kodowania.  
◆ VaryByParam — cachowanie zależne od parametru w akcji:
```

```
[OutputCache(VaryByParam = "id")]  
public ActionResult Index(int id)  
{  
    return View();  
}
```

```
◆ VaryByCustom — cachowanie zależne od indywidualnej implementacji  
zależności.  
◆ VaryByHeader — cachowanie zależne od nagłówka HTTP.
```

Cachowanie częściowe

Cachowanie częściowe:

```
◆ DonutCaching — zapisuje w cache tylko część strony, która się nie zmienia  
dla różnych użytkowników, np. cała strona bez części logowania (inne imię  
lub nazwisko dla każdego użytkownika powodowałoby tworzenie dla każdego  
osobnych danych w cache). W ASP.NET MVC należy zainstalować rozszerzenie  
MvcDonutCaching, aby korzystanie z tej formy cachowania było możliwe.  
◆ DonutHoleCaching — zapisuje te części, które nie mogą być cachowane  
za pomocą DonutCaching (są to tzw. dziury). Za pomocą tego cachowania może  
być zapisywana np. lista kategorii, która zostanie wczytana z cache; reszta  
strony będzie standardowo ładowana lub zostanie wczytana z innego cache.  
Ta forma cachowania jest dostępna przy użyciu atrybutu ChildActionOnly:
```

```
[ChildActionOnly]  
public ActionResult Index()  
{  
    return View();  
}
```

Cachowanie rozproszone

W dużych aplikacjach, gdy dane są przechowywane na więcej niż jednym serwerze, niezbędne jest użycie innego, bardziej wydajnego sposobu cachowania. Tworzy się osobną, wspólną warstwę przeznaczoną do cachowania, do której dostęp posiadają wszystkie

serwery. Takie rozwiązanie zapewnia lepszą wydajność, skalowalność oraz redundancję (odporność na awarie pojedynczych serwerów). Do cachowania rozproszonego służy narzędzie Microsoft Velocity.

Cachowanie po stronie klienta

— Client Side Caching

Cachowanie po stronie klienta ma na celu zmniejszenie do minimum liczby żądań wysyłanych do serwera. Przy każdym odświeżeniu strony przeglądarka sprawdza, czy nie posiada plików, które są częścią ładowanej strony. Jeśli są to pierwsze odwiedziny strony, to przeglądarka pobiera całą zawartość, dla kolejnych żądań pobiera tylko pliki i zdjęcia, które się zmieniają lub są nowe (nie były jeszcze pobierane).

Podstawowe mechanizmy kontroli cache zdefiniowane w protokole HTTP, przekazywane w nagłówku HTTP, to:

- ◆ *freshness* — nie wymaga sprawdzenia na serwerze, czy dana zawartość jest jeszcze aktualna, może być kontrolowana zarówno przez serwer, jak i klienta (np. nagłówek `Expires` określa datę, do której dana treść jest aktualna);
- ◆ *validation* — służy do sprawdzenia, czy dana treść jest nadal aktualna (np. nagłówek `Last-Modified` w celu sprawdzenia, czy dane nie zostały zmodyfikowane);
- ◆ *invalidation* — służy do usunięcia danych z cache.

Cachowanie w HTML 5

HTML 5 Application Cache

Dzięki *HTML 5 Application Cache* możliwe jest przeglądanie zawartości witryny bez konieczności pobierania danych z serwera. Do wyboru plików, które mają być zapisywane po stronie klienta, wykorzystywany jest specjalny plik tekstowy, który zaczyna się od linii `CACHE MANIFEST`. Przeglądanie offline jest możliwe tylko wtedy, gdy już wcześniej użytkownik odwiedzał daną podstronę, a niezbędne pliki są zapisane w pamięci cache.

HTML 5 WebStorage

WebStorage to nowy sposób zapisywania danych po stronie użytkownika. Można wyróżnić typy *LocalStorage* i *SessionStorage*. *LocalStorage* jest powiązane z domeną, dane mogą być współdzielone pomiędzy zakładkami. *SessionStorage* jest powiązane tylko z jedną zakładką. Po zamknięciu zakładki dane sesji są usuwane (jeśli przeglądarka ich nie zapamiętuje).

Code First Data annotations

Data annotations to atrybuty dodawane przed polami w klasie z modelem. Pozwalają na określenie struktury danych i powiązań, jakie mają zostać utworzone w bazie danych¹².

Dostępne atrybuty *Data annotations* to:

- ◆ [Required] — pole wymagane;
- ◆ [Key] — klucz główny (jeśli nazwa klucza głównego nie jest zgodna z konwencją);
- ◆ [MaxLength(10), MinLength(5)] — ograniczenia długości danych do validacji;
- ◆ [NotMapped] — pola, które nie są zapisywane w bazie danych, np. pole łączące Imię i Nazwisko generowane na podstawie danych z pól Imię i Nazwisko;
- ◆ [ComplexType] — typy złożone, np. klasa z kilkoma polami jako część modelu;
- ◆ [Table("NazwaTabeli")] — określa nazwę tabeli w bazie danych;
- ◆ [Column("NazwaKolumny", TypeName="TypDanych")] — określa nazwę kolumny oraz typ danych;
- ◆ [DatabaseGenerated(DatabaseGenerationOption.Computed)] — pola generowane przez bazę danych (np. czas, id);
- ◆ [InverseProperty("NazwaPola")] — stosuje się przy wielokrotnych powiązaniach pomiędzy tabelami;
- ◆ [ForeignKey("FkId")] — klucz obcy;
- ◆ [ConcurrencyCheck] — podczas edycji danych w metodzie POST sprawdzane jest nie tylko id, ale też pola oznaczone atrybutem ConcurrencyCheck — jeśli pola nie były zmienione od czasu odczytu, to operacja kończy się powodzeniem; jeśli odczytane wcześniej dane były w międzyczasie zmienione przez kogoś innego, zostaje rzucony wyjątek;
- ◆ [Timestamp] — pole mówiące o ostatnich zmianach, działa podobnie jak [ConcurrencyCheck] — jeśli od czasu odczytu zostały wprowadzone zmiany i pole Timestamp się nie zgadza, zostaje rzucony wyjątek.

Bezpieczeństwo

Zabezpieczenie aplikacji jest jednym z najważniejszych elementów. Istnieje wiele różnych metod uzyskania dostępu do danych poufnych lub danych, do których określony użytkownik nie powinien mieć dostępu. Najpopularniejsze rodzaje ataków to: *SQL Injection*, *Cross-Site Request Forgery* oraz *Cross-Site Scripting*.

¹² <http://msdn.microsoft.com/en-us/data/gg193958.aspx>

SQL Injection

Ataki typu *SQL Injection* polegają na dołączeniu dodatkowych parametrów do zapytania wykonywanego na bazie danych. Występuje zazwyczaj w przypadku przekazywania parametrów w adresie URL. Najpopularniejszym sposobem jest dodanie do wyrażenia where `id=1` dodatkowej części `or 1=1`. Powoduje to, że wyrażenie jest zawsze prawdziwe, więc zamiast otrzymać tylko jeden wynik dla konkretnego `id`, otrzymuje się wszystkie dane. Nowoczesne ORM-y, jak Entity Framework, są zabezpieczone przed tego typu atakami. Dane przekazywane w parametrach muszą być konkretnego typu, co blokuje możliwość zastosowania *SQL Injection*. Nie oznacza to jednak pełnego bezpieczeństwa — niebezpieczeństwo nadal występuje przy zwykłych zapytaniach SQL oraz procedurach składowanych, które należy zabezpieczać w inny sposób.

Cross-Site Request Forgery

CSRF (ang. *Cross-Site Request Forgery*) to atak mający na celu wykorzystanie uprawnień użytkownika do wykonywania określonych operacji. Użytkownik loguje się do portalu i posiada uprawnienia do dodawania, usuwania lub edycji określonych danych. Osoba atakująca chce wykorzystać fakt, że użytkownik jest zalogowany, i przesyła odnośnik, który po otwarciu wysyła określone żądanie do serwera. Ponieważ użytkownik jest zalogowany, operacja wygląda prawidłowo i nie ma przeszkołów, aby ją wykonać. Przygotowanie prowizorycznej strony jest bardzo proste — wystarczy skopiować kod autentycznej strony i ustawić wartości w formularzu. Po otwarciu fałszywego odnośnika w przeglądarce dane za pomocą skryptu JavaScript zostają automatycznie wysłane bez klikania jakiegokolwiek przycisku.

Aby zapobiec tego typu atakom, stosuje się jednorazowe znaczniki (ang. *Token*) zabezpieczające, które są przesyłane razem z formularzem. Przeglądarka nie pozwala (istnieje możliwość wykorzystania luki w przeglądarce, aby uzyskać dane z innej witryny, jednak obecne przeglądarki są zabezpieczone przed tego typu atakami) na odczyt danych z innych zakładek. Aby żądanie było zaakceptowane przez serwer, musi zawierać wcześniej wysłany *token*.

Aby zabezpieczyć metodę w ASP.NET MVC przed atakiem CSRF, należy dodać przed metodą w kontrolerze atrybut `[ValidateAntiForgeryToken]`. Jest to możliwe tylko przy metodzie POST (listing 5.27). Metoda GET nie powinna i raczej nie jest używana do operacji na danych.

Listing 5.27. Przykład użycia atrybutu `[ValidateAntiForgeryToken]`

```
[HttpPost]
[ValidateAntiForgeryToken]
public ActionResult Zarejestruj(RegisterModel model)
{
    // Ciało metody
}
```

Aby przekazać *token* w formularzu (listing 5.28), należy wykorzystać HTML helper: `@Html.AntiForgeryToken()`.

Listing 5.28. Przykładowy plik widoku

```
@using (Html.BeginForm()) {  
    @Html.AntiForgeryToken()  
  
    <fieldset>  
        <legend>Formularz rejestracji:</legend>  
        // Ciało formularza  
        <input type="submit" value="Załóż konto" />  
    </fieldset>  
}
```

Ograniczenia AntiForgeryToken:

- ◆ wymaga włączonych plików *cookies* (ciasteczek) w przeglądarce;
- ◆ tylko dla żądań POST;
- ◆ łatwe do złamania, gdy są inne luki w aplikacji (XSS).

Cross-Site Scripting

XSS (ang. *Cross-Site Scripting*) polega na osadzeniu w treści strony niepożdanego kodu (najczęściej JavaScript). Aby uchronić aplikacje przed tego typu atakami, należy śledzić treści wprowadzane przez użytkowników. W polach tekstowych wśród tekstu mogą się również znaleźć kody JavaScript, które po zapisie do bazy danych i odczytanie przez innego użytkownika zostają uruchomione. Aby kod został uruchomiony, musi być traktowany jako kod HTML. ASP.NET MVC ma domyślnie zaimplementowaną ochronę przed XSS — nie ma możliwości podawania kodu HTML jako tekstu. Aby umożliwić użytkownikom dodanie kodu HTML, należy użyć atrybutu [AllowHTML], jednak jest to bardzo ryzykowne. Aby zapobiec atakom, należy zainstalować rozszerzenie AntiXSS poprzez platformę NuGet. Nawet jeśli kod zostanie przekazany do bazy danych, AntiXSS przeanalizuje kod wynikowy przesyłany do klienta. Jeśli dany tekst zawiera złośliwy kod, to automatycznie usuwa szkodliwą część. Ponieważ złośliwy kod nie jest wysłany do klienta, nie ma możliwości, aby się wykonał.

Over-Posting — parametr binding

W przypadku gdy jako argument w żądaniu POST przekazuje się obiekt, przesyłane są tylko wymagane dane. Istnieje możliwość podania większej liczby parametrów, które nie są pożądane. Z pomocą przychodzi parametr binding i atrybut [Bind], pozwalające na określenie, jakiego parametru nie chce się używać, nawet gdy będzie przekazany z formularza. Określa się, który parametr ma być ignorowany. Przykładowo przekazuje się obiekt typu kurs, który zawiera pola: skad, dokad, kiedy oraz czy_zaakceptowany. Pole o akceptacji może zmieniać tylko administrator, dlatego należy je zignorować w metodzie wykorzystywanej przez użytkownika. Jako parametr wykorzystywany jest model kurs, czyli wszystkie pola łącznie z polem czy_zaakceptowany. Aby zignorować dane pole, należy użyć atrybutu Bind — Exclude (listing 5.29).

Listing 5.29. Przykład użycia Bind

```
[HttpPost]
[Bind(Exclude="czy_zaakceptowany")]
public ActionResult Dodaj(Kurs kurs)
{
    // Ciało metody
}
```

Obsługa, śledzenie i logowanie wyjątków w MVC

Obsługę wyjątków lub logowania komunikatów można realizować na poziomie lokalnym (w pojedynczej metodzie bądź kontrolerze) lub na poziomie globalnym (dla całej aplikacji w jednym miejscu).

Lokalne zarządzanie wyjątkami

Blok try-catch

Zwrócenie widoku dla błędu w bloku catch działa tylko dla tej jednej operacji (listing 5.30).

Listing 5.30. Blok try-catch

```
public ActionResult TestMethod()
{
    try
    {
        //...
        return View();
    }
    catch (Exception e)
    {
        // Handle Exception;
        return View("Error");
    }
}
```

Nadpisywanie metody OnException() w kontrolerze

Dzięki nadpisaniu metody OnException() dla wszystkich wyjątków w całym kontrolerze zostanie zwrócony odpowiedni widok błędu (listing 5.31).

Listing 5.31. Nadpisywanie metody OnException()

```
protected override void OnException(ExceptionContext filterContext)
{
    Exception e = filterContext.Exception;
```

```
// Log Exception e
filterContext.ExceptionHandled=true;
filterContext.Result = new ViewResult()
{
    ViewName = "Error"
};
}
```

Globalne zarządzanie wyjątkami

Aby włączyć globalne zarządzanie wyjątkami, w pliku *Web.Config* należy ustawić parametr *CustomErrors* na wartość *On*:

```
<customErrors mode="On" defaultRedirect="Error.htm"/>
```

Klasa FilterConfig

W klasie *FilterConfig* znajduje się metoda *RegisterGlobalFilters()*, w której zostaje dodany filtr *HandleErrorAttribute()*. Metoda *RegisterGlobalFilters()* jest uruchamiana w pliku *Global.asax* podczas startu aplikacji. Filtr *HandleErrorAttribute()* będzie zwracał widok *Error.cshtml* z folderu *Views/Shared* dla wszystkich akcji w kontrolerach podczas wystąpienia błędów (listing 5.32).

Listing 5.32. Klasa *FilterConfig*

```
public class FilterConfig
{
    public static void RegisterGlobalFilters
        (GlobalFilterCollection filters)
    {
        filters.Add(new HandleErrorAttribute());
    }
}
```

HandleError na poziomie kontrolerów i akcji

Aby uruchomić obsługę wyjątków dla jednego kontrolera, należy dodać atrybut *[HandleError()]* na poziomie kontrolera i usunąć kod z klasy *FilterConfig*, która włącza obsługę globalną:

```
[HandleError()]
public class KategoriaController : Controller
```

Aby uruchomić obsługę wyjątków dla jednej akcji z kontrolera, należy dodać atrybut przed akcją:

```
public class KategoriaController : Controller
{
    [HandleError()]
    public ActionResult Index()
    {
        return View(db.Kategorie.ToList());
    }
}
```

Zwracanie widoków dostosowanych do konkretnych typów wyjątków

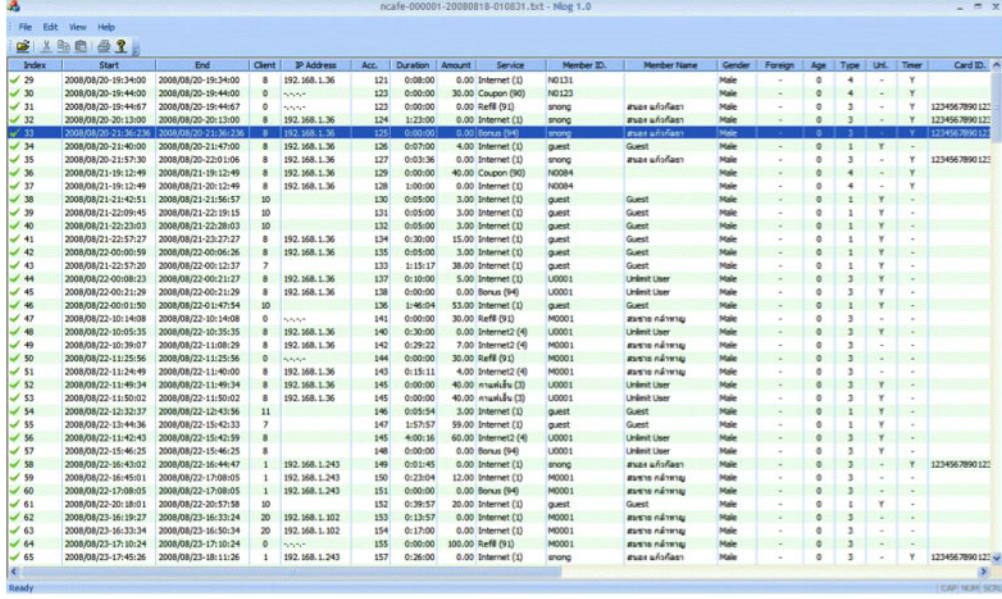
Domyślny atrybut HandleError obsługuje tylko wyjątki wywołane w kontrolerach i służą jedynie do wyświetlania stron błędów. Nie ma możliwości zapisywania ani logowania wyjątków. Aby obsługiwać wyjątki w indywidualny sposób (np. zapisywać w bazie danych), można rozszerzyć implementację atrybutu HandleErrorAttribute:

```
[HandleError(ExceptionType=typeof(DivideByZeroException),
    ↪View="DivideZeroError"]
[HandleError(ExceptionType=typeof(NullReferenceException),
    ↪View="NullRefError"]
public ActionResult Index()
{
    return View(db.Kategorie.ToList());
}
```

Logowanie globalne za pomocą osobnych narzędzi

Istnieje również inny sposób zarządzania wyjątkami (i nie tylko wyjątkami). Służą do tego narzędzia pozwalające na zapisywanie logów. Wyjątki niewyławane przez atrybut HandleError można przechwytywać za pomocą zewnętrznych narzędzi. Aby wszystkie wyjątki były obsługiwane poprzez zewnętrzne narzędzia, należy całkowicie usunąć lub wyłączyć obsługę wyjątków w aplikacji za pomocą atrybutów.

Jednym z narzędzi jest NLOG. Za pomocą NLOG można zapisywać nie tylko wyjątki, ale również wywołania, logi informacyjne lub ostrzeżenia (rysunek 5.24).



Rysunek 5.24. Okno NLOG

Moduł Elmah umożliwia logowanie błędów do plików XML lub do bazy danych SQLite. Dodatkowo trzeba utworzyć katalog *App_Data* w głównym katalogu i nadać mu prawa do zapisu. Po skonfigurowaniu modułu można przeglądać błędy w bardzo przejrzystej formie (rysunek 5.25) pod adresem *twoja_domena/Elmah.axd*.

Rysunek 5.25.

Okno z błędami modułu Elmah

KANAŁ RSS	KANAŁ RSS Z PODZIAŁEM NA DNI	ŚCIĄGNIJ ZAPISANE BŁĘDY	POMOC	O MODULE
Błędy od 1 do 15 z 44 (strona 1 z 3). Wyświetl 10 , 15 , 20 , 25 , 30 , 50 lub 100 błędów na stronę.				
Code	Type	Error	User	Date
404	Http	The controller for path '/wwwroot/Elmah.axd' was not found or does not implement IController. Details...		2012-01-13 16:30
0	InvalidOperationException	Nullable object must have a value. Details...		2011-12-17 04:00
0	InvalidOperationException	Nullable object must have a value. Details...		2011-12-17 01:54
0	InvalidOperationException	Nullable object must have a value. Details...	admin3	2011-12-15 16:48
0	InvalidOperationException	Nullable object must have a value. Details...	admin3	2011-12-15 16:48
0	InvalidOperationException	Nullable object must have a value. Details...	admin3	2011-12-15 16:48
0	DbEntityValidation	Validation failed for one or more entities. See 'EntityTypeErrors' property for more details. Details...	admin3	2011-12-15 14:43

Identyfikacja, uwierzytelnianie i autoryzacja w MVC 5

Identyfikacja

Identyfikacja to inaczej przedstawienie się użytkownika, np. imię, nazwisko, adres zamieszkania (miasto, ulica). Strona odbierająca te dane jest stroną ufającą, że użytkownik podał prawdziwe dane. Przykładem identyfikacji jest zakładanie konta na portalu. Identyfikacja to proces jednorazowy (dla pojedynczego użytkownika).

Uwierzytelnianie

Uwierzytelnianie (ang. *Authentication*) polega na weryfikacji tożsamości przez stronę ufającą (serwer). Metoda uwierzytelnienia zależy od serwera. Zazwyczaj w celu uwierzytelnienia należy podać hasło i login wpisane podczas identyfikacji. Innymi słowy, jest to logowanie na portalu. Jeśli dane zgadzają się z tymi podanymi podczas identyfikacji, użytkownik jest uwierzytelniany i posługuje się danymi wpisanymi w trakcie identyfikacji. Aby uwierzytelnienie było możliwe, najpierw musi wystąpić proces identyfikacji. Uwierzytelnienie to proces, który może być wykonywany wielokrotnie (wielokrotne logowanie na to samo konto).

Autoryzacja

Autoryzacja (ang. *Authorization*) polega na potwierdzeniu, czy dany podmiot jest uprawniony do uzyskania dostępu do żądanego zasobu. Wymagane jest wcześniejsze uwierzytelnienie, aby potwierdzić tożsamość. Najpopularniejszym mechanizmem zarządzania dostępem do zasobów jest użycie ról. Autoryzacja może polegać również na sprawdzeniu, czy dany użytkownik jest autorem bądź właścicielem danego tekstu lub zdjęcia. Role są bardziej ogólnym pojęciem i pozwalają na dostęp grupie użytkowników do tego samego zasobu bądź funkcjonalności. Jeśli użytkownik posiada przypisaną rolę administratora, może korzystać z portalu jako administrator i ma dostęp do większej liczby funkcji.

Role w MVC

W ASP.NET MVC domyślnie zaimplementowany został mechanizm autoryzacji dostępu do treści bądź części aplikacji poprzez role. Aby zarządzać dostępem do poszczególnych akcji w kontrolerze dla użytkowników przypisanych do konkretnych ról, wykorzystywane są atrybuty autoryzacji (listing 5.33). Sprawdzanie, czy dany użytkownik należy do danej roli, jest przeprowadzane automatycznie dzięki ASP.NET Identity.

Listing 5.33. Przykład użycia atrybutu autoryzacji, w którym dostęp do akcji posiadają tylko użytkownicy mający rolę administratora

```
[Authorize(Roles = "Administrator")]
//GET: Ogloszenie/Details/5
public ActionResult Details(int? id)
{
    if (id == null)
    {
        return new HttpStatusCodeResult(HttpStatusCode.BadRequest);
    }
    Ogloszenie ogloszenie = db.Ogloszenia.Find(id.Value);
    if (ogloszenie == null)
    {
        return HttpNotFound();
    }
    return View(ogloszenie);
}
```

Stan aplikacji, sesje i ciasteczka

Stan aplikacji

Stan aplikacji to aktualnie przechowywane informacje o użytkowniku. Tego typu dane są zazwyczaj przechowywane krótki czas i początkowo część z nich jest odczytywana z bazy danych (np. nazwa użytkownika, imię lub login). Ponieważ protokół HTTP używany w aplikacjach internetowych jest bezstanowy, zostały opracowane sposoby na tymczasowe zapisanie potrzebnych danych (np. na pewien czas po zalogowaniu). Dane stanu mogą być zapisywane po stronie serwera (sesje) lub po stronie klienta (ciasteczka), lub jednocześnie na serwerze i w ciasteczkach.

Ciasteczka

Ciasteczka (ang. *cookies*) to dane (informacje na temat stanu) przechowywane w przeglądarce. Podczas logowania serwer generuje ciasteczkę i przesyła je do przeglądarki. Serwer wysyła informacje o ciasteczku za pomocą Set-Cookie (w jednym pakiecie HTTP może być ich wiele). Ciasteczka są przesyłane przy każdym żądaniu zarówno do przeglądarki, jak i od przeglądarki do serwera. Przeglądarki posiadają opcję wyłączania ciasteczek, co powoduje, że nie ma możliwości zapisania danych. Aby pomimo to przekazać dane na temat stanu aplikacji, korzysta się z dopisywania ciasteczka jako parametru do adresu URL lub przekazuje w polu ukrytym w kodzie HTML: <input type="hidden">. Są to bardzo rzadkie przypadki i stosuje się je w ostateczności.

Budowa ciasteczka:

- ◆ name — nazwa ciasteczka (może być ich wiele);
- ◆ expires — data wygaśnięcia; po tym czasie ciasteczkę zostaje skasowane i dane zostają utracone;
- ◆ domain (domena) — atrybut określa, do których serwerów przeglądarka będzie mogła wysłać cookie (widoczność ciasteczka);
- ◆ path (ścieżka) — umożliwia określenie ścieżki na serwerze, w której (wliczając podkatalogi) ciasteczkę będzie widoczne;
- ◆ secure (bezpieczny) — określa, czy konieczne jest połączenie HTTPS (opcjonalne).

Do ciasteczka powinno się dodawać flagę `HttpOnly`, która oznacza, że przeglądarka nie powinna pozwalać na manipulację wartością poprzez języki działające po stronie klienta, np. JavaScript. Dane ciasteczka w takim wypadku mogą być zmieniane tylko poprzez serwer.

Sesje

Sesje to dane (informacje na temat stanu) przechowywane po stronie serwera. Aby było wiadomo, która sesja należy do którego użytkownika, po stronie klienta przechowywana jest wartość SessionId odpowiadająca określonej sesji z serwera. SessionId przechowywane jest w ciasteczku. Różnica pomiędzy sesją przechowywaną po stronie serwera a danymi w ciasteczkach polega na tym, że w ciasteczkach wszystkie dane są przesyłane przy każdym żądaniu HTTP. W przypadku sesji przesyłana jest tylko wartość SessionId. Sesje wpływają na gorszą wydajność aplikacji, ponieważ serwer jest dodatkowo obciążony poprzez przechowywanie danych sesyjnych. Sesje dają potencjalnie większe bezpieczeństwo, gdyż po stronie klienta dane mogą zostać zmienione przez użytkownika, a po stronie serwera nie. Ponieważ sesja jest odczytywana na podstawie id, które jest zapisane w ciasteczku, wystarczy tylko przekopiować wartość ciasteczka i już mamy dostęp do sesji. W zależności od wymogów można używać innego sposobu zapisywania stanu aplikacji, jednak obecnie zalecane jest użycie ciasteczek (pomimo większej ilości danych do przesyłania), ponieważ sesje powodują problemy natury architektonicznej w aplikacji. Jeśli dane sesji są przechowywane w pamięci operacyjnej, to

gdy serwis posiada dwa serwery, jeden z nich nie będzie miał dostępu do sesji drugiego, co może spowodować utratę sesji i wylogowanie użytkownika, jeśli trafi na serwer, w którym nie ma jego sesji.

Istnieje kilka sposobów na przechowywanie sesji:

- ◆ W pamięci operacyjnej procesu (tryb *InProc*) — dane sesyjne są dostępne tylko na jednym serwerze, w razie awarii dane zostają utracone. Najszybsze rozwiązanie, ponieważ dane są odczytywane z pamięci.
- ◆ W bazie danych — dane będą dostępne dla wszystkich serwerów w klastrze, jednak w bardzo dużym stopniu obciąży to bazę danych. To bezpieczne rozwiązanie, gdyż w razie awarii dane pozostają w bazie. Jest też jednak wolne, ponieważ wymaga odczytu i zapisu na dysku — operacje I/O.
- ◆ Na osobnym serwerze (ang. *State Server*) — odpowiedzialnym tylko i wyłącznie za przechowywanie danych sesyjnych. *State Server* jest najlepszym rozwiązaniem, ponieważ nie obciąża bazy danych aplikacji, a dane sesyjne są dostępne dla wszystkich serwerów w klastrze, na których uruchomiona jest aplikacja. Jednak rozwiązanie to jest sensowne i opłacalne dopiero w przypadku dużych serwisów.

OWIN

W poprzednich wersjach MVC (przed MVC 5) do uwierzytelniania za pomocą ciasteczek używany był mechanizm *Forms Authentication* wykorzystujący uwierzytelnianie na podstawie formularzy. W wersji MVC 5 zostało wprowadzone nowe, nowoczesne rozwiązanie o nazwie *OWIN cookie authentication middleware*. OWIN nie różni się pod względem funkcjonalności od *Forms Authentication* — zapisuje ciasteczka po stronie klienta i przy każdym żądaniu dekoduje i waliduje dane z ciasteczka. Zdekodowane dane są dostępne w aplikacji (w obiekcie `HttpContext.User`) bez konieczności odczytu ich z bazy danych. OWIN — w przeciwieństwie do *Forms Authentication* — posiada wsparcie dla *claimów* (pojedynczych informacji o użytkowniku — stwierzeń). Dane w postaci *claimów* są dopisywane do ciasteczek jako znacznik. Nie należy mylić OWIN z sesjami, ponieważ jest on wykorzystywany tylko do ciasteczek. OWIN bazuje na projekcie Katana i może być uruchamiany jako samodzielna aplikacja, co oznacza, że nie wymaga serwera IIS. ASP.NET Web API 2 również posiada możliwość samodzielnego hostowania. ASP.NET MVC ma umożliwiać samodzielne hostowanie w kolejnej wersji — MVC 6.

ASP.NET Identity

ASP.NET Identity to framework służący do identyfikacji (zakładania konta) i uwierzytelniania (logowania) użytkowników. Pozwala w prosty sposób uruchomić logowanie poprzez zewnętrzne serwisy (np. Google, Facebook, Twitter, Microsoft) bez konieczności implementacji własnych rozwiązań. ASP.NET Identity jest wspólny dla różnych typów aplikacji: ASP.NET MVC, Web Forms, Web Pages, Web API i SignalR. Proces

uwierzytelniania polega na pobraniu danych od użytkownika (login i hasło) i sprawdzeniu zgodności podanych danych z tymi znajdującymi się w bazie danych. ASP.NET Identity dzieli się na bloki:

- ♦ *authentication manager*, klasa `UserManager<TUser>` — część odpowiedzialna za uwierzytelnienie użytkowników, logowanie bądź wylogowanie; korzysta z OWIN, który zajmuje się tworzeniem i sprawdzaniem ciasteczek;
- ♦ *store manager*, klasa `UserStore<TUser>` — część odpowiedzialna za zarządzanie i przechowywanie danych w dowolnym źródle, np. w bazie danych.

Zalety ASP.NET Identity:

- ♦ wspólny dla wszystkich rodzajów aplikacji opartych na ASP.NET;
- ♦ pozwala użytkownikowi wybrać miejsce składowania danych za pomocą odpowiednich providerów (bazy NoSQL, Azure, SQL Server, MySQL);
- ♦ pozwala na prostą kontrolę i rozszerzanie ilości danych o użytkowniku, które chce się przechowywać w tabeli o użytkowniku;
- ♦ zaimplementowany *Role Provider*;
- ♦ wsparcie dla testów jednostkowych (ang. *Unit Test*);
- ♦ wsparcie dla *claimów*;
- ♦ posiada wbudowane providery do logowania poprzez zewnętrzne serwisy, takie jak Microsoft Account, Facebook, Twitter, Google;
- ♦ pozwala na logowanie poprzez Windows Azure Active Directory;
- ♦ integracja z OWIN;
- ♦ dystrybuowany w formie pakietu NuGet, co pozwala na prostsze rozpowszechnianie i aktualizację.

WIF i uwierzytelnianie za pomocą claimów

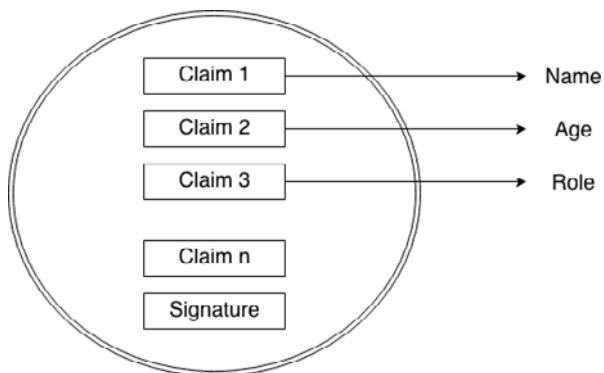
Claims-based Authentication polega na oddelegowaniu logiki uwierzytelniania do innej warstwy lub aplikacji, która ukrywa wszystkie sposoby uwierzytelniania i zwraca wniosek, czyli informację, czy użytkownik jest uwierzytelniony, czy nie, oraz niektóre informacje o użytkowniku (*claimy*). Na podstawie tych informacji aplikacja podejmuje decyzję o autoryzacji (dostępie do określonych zasobów lub części aplikacji).

WIF (ang. *Windows Identity Foundation*) to framework Microsoftu wykorzystujący *Claims-based Authentication*. Posiada zestaw klas upraszczających proces uwierzytelniania bazujący na *claimach*. Wykorzystuje między innymi STS (ang. *Secure Token Service*) do generowania tokenów. WIF jest jedną z części Microsoft Federated Identity, która składa się z trzech części: WIF, ACS (ang. *Windows Azure Access Control Services*) i AD FS (ang. *Active Directory Federation Services*). WIF korzysta ze standardów WS-Federation, czyli rozszerzeń do protokołu SOAP, które dodają autoryzację bądź uwierzytelnienie oraz zabezpieczenie informacji przesyłanych przez ten protokół.

Token to ciąg bajtów przesyłanych poprzez sieć do użytkownika. Posiada on listę informacji o użytkowniku w formie *claimów* (w formacie słownikowym, np. Imię: "Marek"). Każdy *claim* ma pojedynczą informację o użytkowniku (rysunek 5.26).

Rysunek 5.26.

Budowa tokena



Identity Provider, STS

Identity Provider uwierzytelnia użytkownika i tworzy token zawierający dane z *claimów* oraz podpis elektroniczny potrzebny w celu późniejszej weryfikacji autentyczności. *Identity Provider* jest nazywany STS (rysunek 5.27).

Rysunek 5.27.

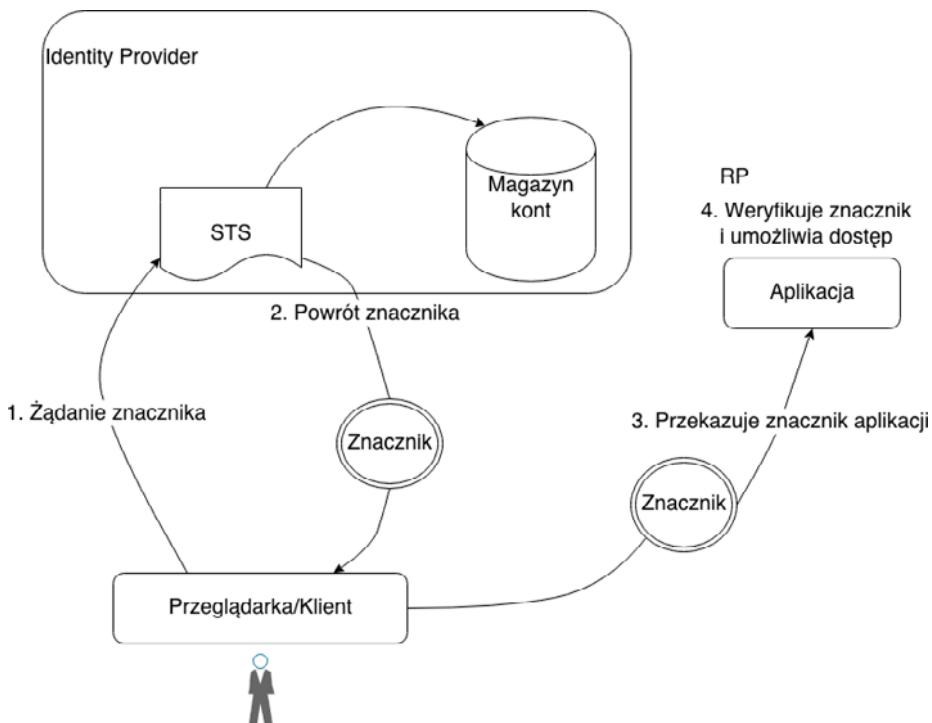
Pobieranie tokena
z STS



Strona ufająca — Relying Party

RP (ang. *Relying Party*) to aplikacje, które używają *Identity Provider* (STS) do uwierzytelnienia. RP musi budować zaufaną relację z STS, aby zweryfikować autentyczność danych oraz przekazać informacje, jakie dane są potrzebne dla danego użytkownika.

Po prawidłowym uwierzytelnieniu użytkownika za pomocą STS RP tworzy ciasteczkę, dzięki czemu przy kolejnym żądaniu nie musi ponownie przechodzić procesu uwierzytelniania (rysunek 5.28).

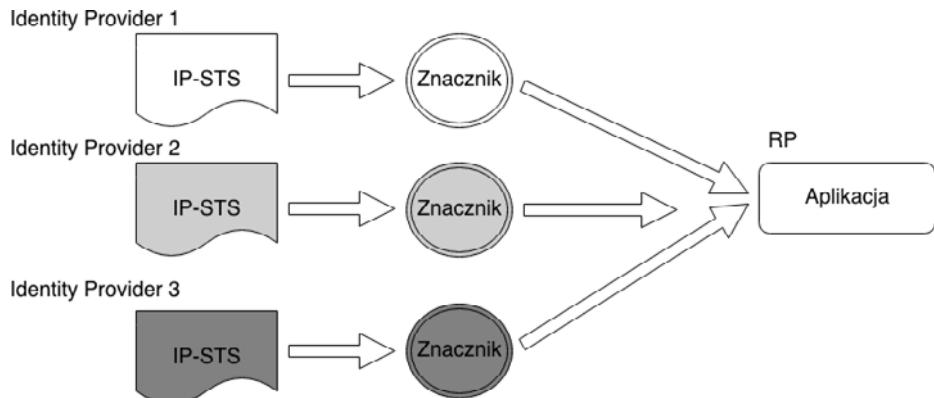


Rysunek 5.28. Kompletny proces Claim-based Authentication

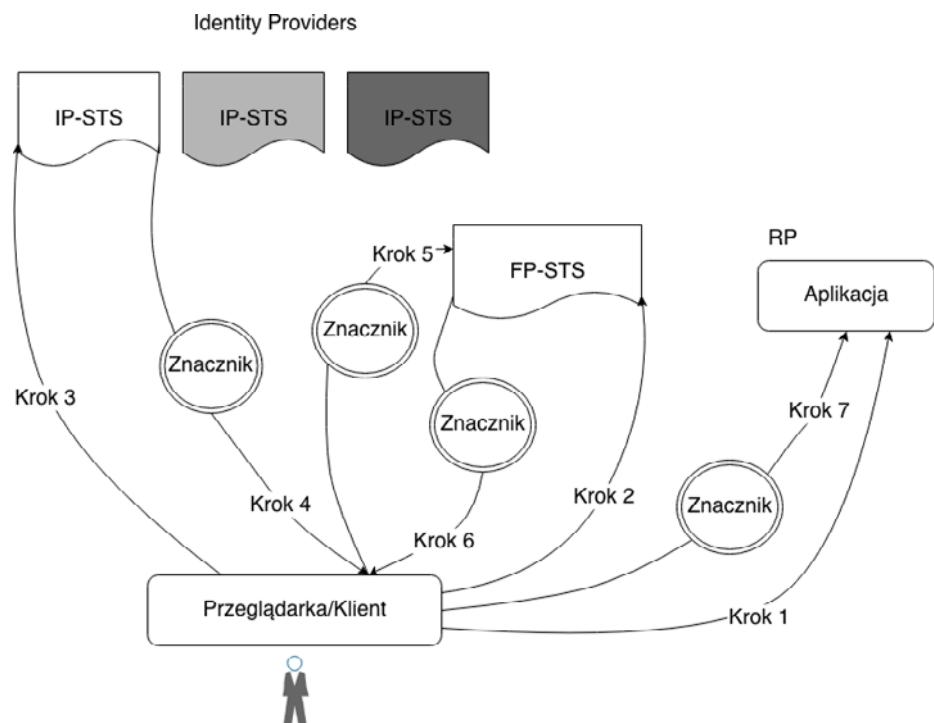
Federated Authentication

Aplikacja (RP) musi umieć rozpoznawać różne tokeny dla różnych STS (np. Facebook, Google, Microsoft itp.) — to jej słaby punkt. Każdy *Identity Provider* może używać innego protokołu i dla każdego STS trzeba tworzyć osobny kod odpowiedzialny za jego obsługę. Za każdym razem konieczna jest zmiana kodu aplikacji, co jest bardzo złym rozwiązaniem (rysunek 5.29).

Aby rozwiązać ten problem, zostało wprowadzone *Federated Identity*, które polega na utworzeniu *Federated Provider*. Potrafi on rozpoznać znaczniki ze wszystkich *Identity Provider* i na ich podstawie generuje jeden uniwersalny znacznik dla aplikacji. Aplikacja nie musi nic wiedzieć o serwisach, na podstawie których użytkownik został uwierzytelniony, musi tylko i wyłącznie umieć rozpoznać znacznik z *Federated Provider* (rysunek 5.30). Odpowiedzialność za uwierzytelnianie jest przenoszona na inną warstwę w aplikacji.



Rysunek 5.29. Dodatkowy kod do obsługi różnych tokenów

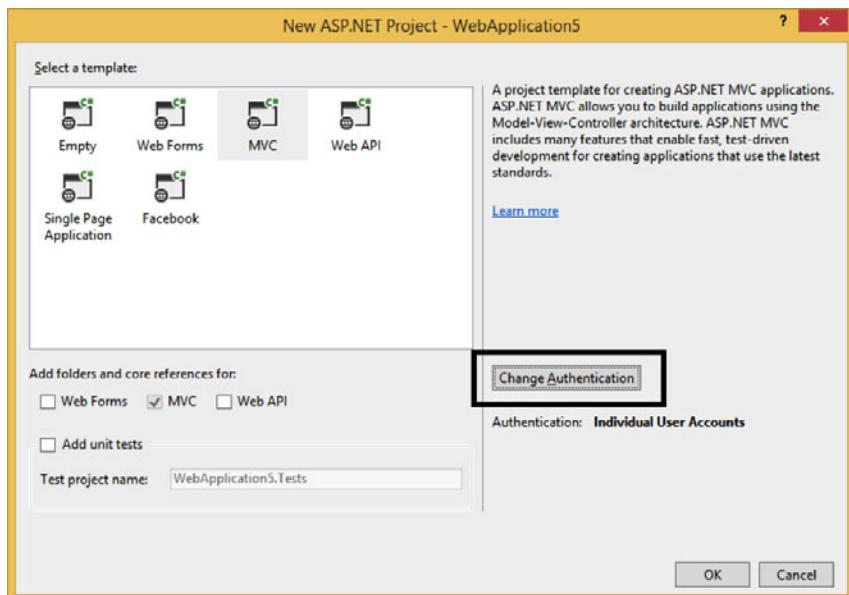


Rysunek 5.30. Schemat dla Federated Authentication

Windows ACS

Windows ACS (ang. *Access Control Service*) to chmurowy serwis Microsoftu, który w prosty sposób pozwala na uwierzytelnianie i autoryzację użytkowników poza kodem aplikacji dzięki *Federated Authentication*. Visual Studio umożliwia uruchomienie

w prosty sposób uwierzytelniania przez Windows ACS. Podczas tworzenia projektu wystarczy kliknąć przycisk *Change Authentication* (rysunek 5.31), a następnie wybrać opcję *Organizational Accounts*.



Rysunek 5.31. Uwierzytelnianie przez Windows ACS

OpenId i OpenAuth

OpenId

OpenId to uwierzytelnianie użytkownika za pomocą kont założonych na zewnętrznych serwisach. Pozwala na dostęp do danych z aplikacji (np. *portal.pl*) dzięki uwierzytelnianiu poprzez inną aplikację (np. Facebook). Nie ma potrzeby zakładania osobnego konta na portalu *portal.pl*, aby móc korzystać z jego funkcjonalności. Na portalu zostaje zapisany zazwyczaj adres e-mail użytkownika i certyfikat, aby możliwe było późniejsze zalogowanie na to samo konto. Hasło nie jest przesyłane.

Przykładowe kroki logowania przez *OpenId*:

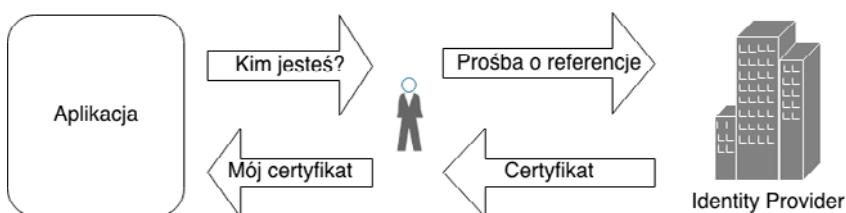
- ◆ użytkownik chce się zalogować na *portal.pl*;
- ◆ *portal.pl* (*Relying Party* w *OpenId*) pyta użytkownika o jego *OpenId*;
- ◆ użytkownik podaje swoje *OpenId*, czyli wybiera portal, na którym posiada konto;
- ◆ *portal.pl* przekierowuje użytkownika do providera *OpenId*;
- ◆ użytkownik przechodzi proces uwierzytelniania (loguje się, jeśli nie jest wcześniej zalogowany);
- ◆ *OpenId* przekierowuje użytkownika z powrotem na *portal.pl*;

- ◆ *portal.pl* pozwala użytkownikowi na dostęp do swojego konta na portalu, które zostało wcześniej powiązane z tym providerem (np. użytkownikiem Facebooka), lub zakłada nowe konto na *portal.pl* (powiązane z kontem na Facebooku) na podstawie danych przesyłanych z zewnętrznego usługodawcy.

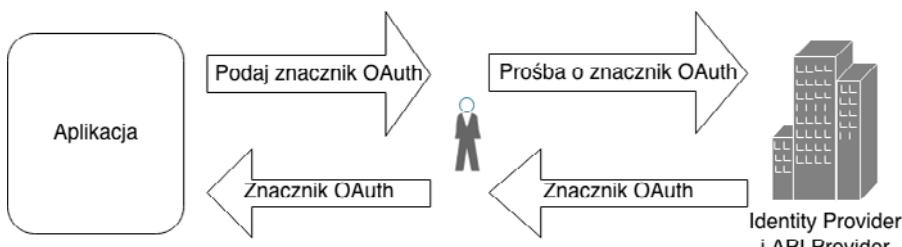
OpenAuth

OpenAuth polega na autoryzacji dostępu do zasobów lub funkcjonalności z innego portalu bez konieczności przesyłania danych potrzebnych do uwierzytelniania (przesyłane są tokeny). *OpenAuth* pozwala na dostęp do API różnych serwisów, aby pobrać takie dane jak np. zdjęcia bez konieczności podawania loginu i hasła do tego serwisu. *OpenAuth* 2.0 umożliwia również uwierzytelnianie użytkowników, które korzysta z *OpenId Connect*. *OpenId* w najnowszej wersji pozwala na autoryzację, jednak nadal podstawa zasada działania pozostaje różna (*OpenId* — uwierzytelnienie, *OpenAuth* — autoryzacja) i w zależności od wymogów bardziej odpowiedni będzie inny protokół.

Porównanie uwierzytelniania w *OpenId* i pseudouwierzytelniania w *OpenAuth* przedstawiają rysunki 5.32 i 5.33.



Rysunek 5.32. Uwierzytelnianie OpenId



Rysunek 5.33. Pseudouwierzytelnianie OpenAuth

Rozdział 6.

Web serwisy i ASP.NET Web API 2

ASP.NET Web API (wprowadzone wraz z MVC 4) to interfejs komunikacyjny korzystający z protokołu HTTP i formatu XML lub JSON w celu przesłania danych. Web API pozwala na zdalny dostęp do zasobów serwisu. Zwraca tylko dane w wybranym formacie, a nie — jak w ASP.NET MVC — całą stronę WWW razem z odpowiednio wyświetlonymi i sformatowanymi danymi. Web API pozwala na prostą i szybką wymianę informacji z zewnętrznymi serwisami lub aplikacjami mobilnymi bez konieczności generowania widoków. W kontrolerze ASP.NET MVC również można zwrócić dane za pomocą odpowiedniego typu ActionResult, jednak dobrą praktyką jest przeniesienie metod zwracających dane w formacie XML lub JSON do Web API. Web API może być osobną aplikacją lub częścią projektu MVC. Posiada osobny katalog w strukturze projektu, w którym znajdują się kontrolery Web API. Kontroler Web API musi dziedziczyć po klasie ApiController. Routing dla Web API jest konfigurowany w pliku *WebApiConfig* i różni się tylko przedrostkiem *api/* w adresie URL.

Domyślna reguła routingu dla Web API jest następująca:

```
config.Routes.MapHttpRoute(  
    name: "DefaultApi",  
    routeTemplate: "api/{controller}/{id}",  
    defaults: new { id = RouteParameter.Optional }  
)
```

Web API 2

Nowości w Web API 2 (od wersji MVC 5):

- ◆ routing na podstawie atrybutów (ang. *Attribute Routing*);
- ◆ mechanizm ignorowania ścieżek routingu (ang. *Ignore Routes*);
- ◆ CORS;

- ◆ autentykacja za pomocą *OAuth* (ang. *Web API Authentication using OAuth-OpenID*);
- ◆ samodzielne hostowanie przy użyciu OWIN (ang. *Open Web Interface for .NET*);
- ◆ poprawki i rozszerzenia w Web API *OData* (`$expand`, `$value`);
- ◆ interfejs `IHttpActionResult` — pozwala na wstrzykiwanie własnych, niestandardowych implementacji typów rezultatów;
- ◆ `HttpRequestContext` — dostęp do szczegółowych informacji na temat żądania;
- ◆ asynchroniczne filtry (ang. *Async Filters*);
- ◆ uproszczenia i rozszerzenia możliwości testowania aplikacji Web API;
- ◆ nadpisywanie filtrów (ang. *Filter Overrides*);
- ◆ formatowanie danych w formacie BSON (ang. *Binary JSON*);
- ◆ globalne zarządzanie wyjątkami (ang. *Global Error Handling*).

Web API a ASP.NET MVC

Web API zostało utworzone z myślą o zwracaniu danych, a nie całych widoków lub stron WWW, dlatego nie zwraca się typu `ActionResult`. Zwracane są tylko listy lub wybrane typy wartości, które są automatycznie zamieniane na dane w formacie XML lub JSON. Nie trzeba przy każdej akcji określać formatu zwracanych danych, ponieważ jest on ustalany ogólnie dla całej aplikacji Web API. Można zmienić domyślny typ zwracanych danych z JSON na XML w jednym miejscu w konfiguracji. Web API wspiera mechanizm negocjacji typów (ang. *Content Negotiation*). Polega on na tym, że jeśli domyślnie zwracane są dane w formacie XML, a ktoś zażąda danych w formacie JSON, to Web API zwróci dane w formacie JSON. W Web API 2.1 zostało dodane wsparcie dla formatu BSON (binarnego JSON). Reguły routingu działają bardzo podobnie jak w aplikacjach MVC. Web API wspiera również routing na podstawie atrybutów (ang. *Attribute Routing*), jednak akcje z kontrolera są wybierane na innej zasadzie. W Web API nazwy akcji są wybierane na podstawie typu żądania. Kontrolery w Web API dziedziczą po klasie `ApiController` (listing 6.2), a nie — tak jak w MVC — po klasie `Controller` (listing 6.1). Web API wspiera samodzielne hostowanie bez serwera IIS. Serwis działa wtedy jako samodzielna aplikacja. Traci się jednak wszystkie zalety serwera IIS i nie można np. śledzić zużycia zasobów.

Listing 6.1. Przykład w ASP.NET MVC

```
public class OgloszenieController : Controller
{
    // GET: /Ogloszenia/
    [HttpGet]
    public ActionResult Index() {
        return Json(Ogloszenia.GetData(), JsonRequestBehavior.AllowGet);
    }
}
```

Listing 6.2. Przykład w ASP.NET Web API

```
public class OgloszenieController : ApiController
{
    // GET: /Api/Ogloszenia/
    public List<Ogloszenie> Get() {
        return Twitter.GetData();
    }
}
```

Web serwis, REST, SOAP i OData

Web serwis (ang. *Web Service*) to nic innego jak pośrednik pomiędzy danymi pobieranymi z różnych źródeł, najczęściej z bazy danych jednego serwisu, i udostępnianymi w sieci dla innych aplikacji nieposiadających bezpośredniego dostępu do bazy danych, z której pochodzą dane. Poprzez Web serwis udostępnia się tylko wybrane dane. Web serwisy pozwalają na wymianę danych pomiędzy firmami, oddziałami firm lub rozproszonymi częściami aplikacji. Istnieje kilka technologii pozwalających na tworzenie Web serwisów, czyli warstwy komunikacji sieciowej. Obecnie najpopularniejsze są serwisy w stylu REST lub oparte na protokole SOAP. SOAP jest starszą technologią z największymi możliwościami, jednak wymaga przesyłania większej ilości danych. REST jest nową technologią, nie jest protokołem i opiera się głównie na żądaniach HTTP, dzięki czemu bardzo dobrze wpisuje się w specyfikację nowoczesnych witryn internetowych i aplikacji mobilnych. Web serwis to pojęcie ogólne, jednak jeszcze do niedawna wskazywało na użycie technologii SOAP i przesyłanie danych w formacie XML.

SOAP

SOAP (ang. *Simple Object Access Protocol*) to protokół zdalnego dostępu do obiektów. Opiera się na komunikacji protokołami HTTP lub RPC (najczęściej) oraz przesyła dane w formacie XML.

Dokument SOAP składa się z trzech części:

- ◆ koperty (ang. *Envelope*) — szkielet opisujący, co znajduje się w komunikacie i jak go przetwarzać,
- ◆ zbioru reguł kodujących, potrzebnych do rozszyfrowania typów danych (również złożonych) zdefiniowanych wewnętrz aplikacji,
- ◆ reguł dotyczących wywoływania zdalnych metod i odczytu odpowiedzi.

REST

REST jest stylem architektonicznym, a nie specyfikacją lub standardem. Większość ludzi określa aktualnie mianem REST wszystko, co nie jest SOAP i działa z użyciem protokołu HTTP. REST nie jest związany z żadnym protokołem komunikacyjnym, jednak

najpopularniejszy protokół używany wraz z REST to HTTP. HTTP posiada predefiniowany zestaw metod, takich jak POST, GET, PUT i DELETE, które same w sobie niosą znaczenie. REST wymaga określenia jawnych zasad działania na zasobach, a właśnie to dają metody HTTP. REST nie określa formatu zwracanych danych, jednak najpopularniejsze formaty to XML i JSON.

Standardowe odwołanie do zasobów wygląda następująco:

```
/api/ogloszenie/get/ // HTTP GET
```

Odwołanie w REST:

```
/api/ogloszenie/ // HTTP GET
```

OData

OData (ang. *Open Data Protocol*) to protokół pozwalający na dostęp, odczytywanie i aktualizację danych w ten sam sposób dla różnych typów aplikacji. Oznacza to, że można wywołać zapytania na źródle danych za pomocą wywołania zwykłego adresu URL. *OData* bazuje na protokołach HTTP i Atom oraz korzysta z zasad REST i zwraca dane w formacie XML lub JSON. Serwis *OData* musi posiadać model danych EDM (ang. *Entity Data Model*) podobny do tego, którego używa Entity Framework. Model danych jest niezbędny, aby było możliwe wykonywanie zapytań na źródle danych *OData*. EDM w *OData* przechowuje kolekcje wpisów (ang. *Collection* — odpowiednik tabeli) i relacje (ang. *Link*) pomiędzy wpisami — jeden wpis (ang. *Entry* — odpowiednik wiersza w tabeli) może mieć wiele właściwości (ang. *Property* — odpowiednik pojedynczej wartości z wiersza), gdzie właściwość zawiera część wpisu. Dane w kolekcjach mogą pochodzić z różnych źródeł (ang. *Feed*) i są udostępniane w sieci jako zasoby (ang. *Resource*).

Adres serwisu *OData* składa się z części:

- ◆ adres serwisu: <http://services.odata.org/OData/OData.svc>,
- ◆ ścieżka do zasobu: *Kategorie(1)/Produkty*,
- ◆ operatory zapytań: *top=3&\$orderby=Nazwa*.

Ostateczny adres, który zwróci wiersze posortowane po nazwie¹, wygląda następująco:

```
http://services.odata.org/OData/OData.svc/Kategorie(1)/  
Produkty?$top=3&$orderby=Nazwa
```

Dostępne operacje zapytań to: \$orderby (sortowanie), \$top (liczba zwróconych wpisów), \$filter (filtrowanie danych), \$format (format danych), \$select (właściwości, które chce się pobrać), \$expand (pozwala na pobieranie danych z powiązanych tabel — operacja join), \$value (zwraca pojedynczą wartość).

¹ Znak ? poprzedza część zapytania, znak \$ — każdą operację zapytania.

CORS i JSONP

CORS i JSONP pozwalają na pobieranie danych z Web serwisów, które są umieszczone w innej domenie niż ta, która jest aktualnie odwiedzana.

JSONP

Za pomocą żądań AJAX można pobierać dane tylko z tej samej domeny, na której działa strona. Żądania z innych domen są blokowane przez przeglądarkę ze względu na bezpieczeństwo. Aby pobierać dane przy użyciu JavaScriptu ze źródeł umieszczonych w innych domenach, można użyć technologii JSONP (ang. *JSON with Padding*). JSONP korzysta z luki (możliwości) w przeglądarkach, która pozwala na pobieranie skryptów zamieszczonych na innych domenach. Kod musi się znajdować w tagu `<script>` (listing 6.3). JSONP jest wspierany przez większość przeglądarek — zarówno tych starszych, jak i najnowszych. W JSONP istnieje możliwość ataków XSS polegających na wstrzyknięciu złośliwego kodu.

Listing 6.3. Przykładowy kod standardowego wywołania JSONP oraz za pomocą jQuery:

```
//Standardowe wywołanie
<script type="application/javascript"
src="http://serwer.przykładowy.pl/Users/1234?jsonp=parseResponse">
</script>

//Wywołanie za pomocą jQuery i AJAX
$.ajax({
  url: "←http://serwer.przykładowy.pl/Users/1234?jsonp=parseResponse",
  dataType: "jsonp",
  success: function( response ) { console.log( response ); }
});
```

CORS

JSONP wspiera tylko żądania GET, czyli pobieranie danych z innego serwera. Nową technologią pozwalającą na komunikację z innymi domenami niż ta, na której znajduje się odwiedzana strona, jest CORS (ang. *Cross-Origin Resource Sharing*). CORS wspiera różne żądania HTTP — zarówno GET, jak i POST, korzysta z technologii AJAX i używa obiektu XMLHttpRequest. CORS określa, czy i w jaki sposób przeglądarka (określona strona) może się komunikować z serwerem dostępnym pod inną domeną. CORS jest bezpieczniejszą technologią niż JSONP, ponieważ pozwala walidować dane przesłane przez serwer znajdujący się w innej domenie.

Zasada działania CORS:

- ◆ Przeglądarka wysyła nagłówek do serwera Origin HTTP header zawierającego nazwę domeny, w której zamieszczona jest strona:

Origin: http://www.czyszczanie-dywianow.pl

- ◆ Jeśli serwer pozwala na kontakt z tą domeną, odsyła w odpowiedzi nagłówek ACAO (ang. *Access-Control-Allow-Origin*), zawierający listę domen, które mogą się z nim komunikować. Jeśli w odpowiedzi nie ma odpowiedniej nazwy domeny, żądanie zostanie zablokowane i do witryny (domeny) zostanie zwrócony błąd:

```
Access-Control-Allow-Origin: http://www.czyszczanie-dywanow.pl
```

W starszych przeglądarkach, które nie posiadają wsparcia dla CORS, wykorzystywany jest JSONP.

Uruchamianie CORS w Web API

Atrybut `EnableCors` posiada parametry określające, dla jakich żądań ma zezwolić na dostęp do zasobów: `origins` — lista domen, które mają zezwolenie na dostęp, `headers` — akceptowane nagłówki żądań HTTP (np. `accept`, `content-type`, `origin`, `x-my-header`), i `methods` — dostępne metody (np. `GET`, `POST`)².

Włączanie CORS dla jednej metody przebiega następująco:

```
[EnableCors(origins: "http://www.example.com", headers: "*",
            methods: "*")]
public HttpResponseMessage GetOgloszenie(int id) { ... }
```

A tak wygląda włączanie CORS dla jednego kontrolera:

```
[EnableCors(origins: "http://www.example.com", headers: "*",
            methods: "*")]
public class OgloszenieController : ApiController
```

Poniżej zaprezentowano sposób włączania CORS globalnie:

```
public static class WebApiConfig
{
    public static void Register(HttpConfiguration config)
    {
        var cors = new EnableCorsAttribute
                    ("www.example.com", "*", "*");
        config.EnableCors(cors);
        //...
    }
}
```

Istnieje możliwość zdefiniowania własnego atrybutu służącego do uruchamiania CORS (listing 6.4). Atrybut musi dziedziczyć po klasie `Attribute` i implementować interfejs `ICorsPolicyProvider`. Na listingu 6.5 zaprezentowano jego wykorzystanie.

Listing 6.4. Przykład implementacji własnego atrybutu

```
[AttributeUsage(AttributeTargets.Method |
                AttributeTargets.Class, AllowMultiple = false)]
public class MyCorsPolicyAttribute : Attribute, ICorsPolicyProvider
{
```

² <http://www.asp.net/web-api/overview/security/enabling-cross-origin-requests-in-web-api>

```
private CorsPolicy _policy;

public MyCorsPolicyAttribute()
{
    // Tworzenie polityki CORS
    _policy = new CorsPolicy
    {
        AllowAnyMethod = true,
        AllowAnyHeader = true
    };

    // Dodawanie dozwolonych domen
    _policy.Origins.Add("http://aspnetmvc.pl");
    _policy.Origins.Add("http://JakaStrona.pl");
}

public Task<CorsPolicy> GetCorsPolicyAsync
    =>(HttpRequestMessage request)
{
    return Task.FromResult(_policy);
}
```

Listing 6.5. Przykład wykorzystania własnego atrybutu

```
[MyCorsPolicy]
public class TestController : ApiController
{ }
```

Routing w Web API

Routing w Web API nie różni się znacząco od routingu w MVC. Web API 2 wspiera routing na podstawie atrybutów i wszystkie pozostałe mechanizmy routingu z MVC. Domyślnie akcje wybierane są na podstawie konwencji, która mówi, że nazwa metody musi się zaczynać nazwą metody HTTP z nagłówka żądania. Aby oznaczyć metody, które nie mają być dostępne jako API, został wprowadzony atrybut [NonAction]:

```
[NonAction]
public int PrzeliczWartosc(int wart1, int wart2);
```

Mapowanie żądań na akcje bądź metody w kontrolerze Web API

Mapowanie na podstawie nazwy akcji:

- ♦ nazwa metody taka sama jak nazwa akcji z żądania (adresu URL):

```
public HttpResponseMessage Ogloszenia();
```

- ◆ poprzez dodanie nazwy typu żądania HTTP na początku nazwy akcji:

```
public HttpResponseMessage GetOgloszenie (int id);
```

- ◆ poprzez użycie atrybutu [ActionName]:

```
[HttpGet]
[ActionName("Ogłoszenia")]
public HttpResponseMessage ListaOgłoszeń(int id);

[HttpPost]
[ActionName("Ogłoszenia")]
public void AddOgłoszenie(int id);
```

Mapowanie po metodzie HTTP:

- ◆ poprzez dodanie nazwy typu żądania HTTP na początku nazwy akcji:

```
public IEnumerable<Ogłoszenie> GetOgłoszenieById(int id) { ... }
public HttpResponseMessage DeleteOgłoszenie(int id){ ... }
```

- ◆ poprzez użycie atrybutu [AcceptVerbs]:

```
[AcceptVerbs("GET", "HEAD")]
public Ogłoszenie ZnajdzOgłoszenie(id) { ... }
```

- ◆ poprzez dodanie atrybutu z ActionMethodSelectorAttribute, np. [HttpGet]:

```
[HttpGet]
public Ogłoszenie ZnajdzOgłoszenie (id) { ... }
```

Atrybuty dostępne dla metod HTTP to: HttpDelete, HttpGet, HttpHead, HttpOptions, HttpPatch, HttpPost oraz HttpPut.

Web API a Entity Framework i warstwa modelu

Zarówno tworzenie bazy danych, jak i migracje działają identycznie jak w przypadku MVC. Warstwa modelu może być wspólna dla Web API i MVC. Metody przeniesione do warstwy repozytorium mogą zostać wykorzystane zarówno w kontrolerach Web API, jak i w kontrolerach MVC. Entity Framework pozwala (tak samo jak w MVC) na podejście *Code First*, *Model First* i *Database First*.

Typy rezultatu w Web API

W Web API istnieją typy rezultatu (ang. *Action Results*), które można zwrócić z kontrolera: void, HttpResponseMessage, IHttpActionResult, a także inny typ rezultatu (np. lista produktów).

Typ void

Metoda z kontrolera zwracająca typ void (listing 6.6) po prostu nic nie zwraca. Zwracony status HTTP to 204 (listing 6.7).

Listing 6.6. Typ rezultatu void

```
public void Post()
{
}
```

Listing 6.7. Odpowiedź HTTP

```
HTTP/1.1 204 No Content
Server: Microsoft-IIS/8.0
Date: Mon, 12 JUN 2014 12:18:46 GMT
```

HttpResponseMessage

Metoda z kontrolera zwracająca typ HttpResponseMessage (listing 6.8) daje pełną kontrolę nad odpowiedzią i zwraca skonfigurowaną odpowiedź HTTP (listing 6.9).

Listing 6.8. Typ rezultatu HttpResponseMessage

```
public class ValuesController : ApiController
{
    public HttpResponseMessage Get()
    {
        HttpResponseMessage response
            => Request.CreateResponse(HttpStatusCode.OK, "value");
        response.Content = new StringContent("Tresc", Encoding.Unicode);
        response.Headers.CacheControl = new CacheControlHeaderValue()
        {
            MaxAge = TimeSpan.FromMinutes(20)
        };
        return response;
    }
}
```

Listing 6.9. Odpowiedź HTTP

```
HTTP/1.1 200 OK
Cache-Control: max-age=1200
Content-Length: 10
Content-Type: text/plain; charset=utf-16
Server: Microsoft-IIS/8.0
Date: Mon, 12 JUN 2014 12:18:46 GMT
```

Tresc

IHttpActionResult

IHttpActionResult to interfejs wprowadzony w Web API 2, który posiada jedną metodę ExecuteAsync, tworzącą asynchronicznie instancje klasy HttpResponseMessage. Dzięki zastosowaniu interfejsu pozwala za implementację własnego typu ActionResult. Aby utworzyć własny typ rezultatu (listing 6.10), należy zaimplementować interfejs IHttpActionResult, a w nim asynchroniczną metodę ExecuteAsync.

Listing 6.10. Przykład wykorzystania własnego typu Action Result w kontrolerze

```
public class JakisController : ApiController
{
    public IHttpActionResult Get()
    {
        return new WlasnyTypRezultatu("hello", Request);
    }
}
```

Inny dowolny typ z aplikacji

Istnieje możliwość zwracania dowolnego typu z aplikacji (listing 6.11), np. może to być lista ogłoszeń (listing 6.12).

Listing 6.11. Dowolny typ rezultatu

```
public class OgloszenieController : ApiController
{
    public IEnumerable<Ogloszenie> Get()
    {
        return PobierzWszystkieOgłoszeniaZBazy();
    }
}
```

Listing 6.12. Odpowiedź HTTP

Odpowiedź HTTP:
HTTP/1.1 200 OK
Content-Type: application/json; charset=utf-8
Server: Microsoft-IIS/8.0
Date: Mon, 27 Jan 2014 08:53:35 GMT
Content-Length: 56

[{"Id":1,"Tytul":"Ogłoszenie o pracę","Kategoria":"Budownictwo","Cena":9.99}]

Pobieranie danych z Web API

Dane z Web API mogą być pobierane za pomocą dowolnych klientów. Klientami mogą być inne aplikacje, urządzenia mobilne lub nawet ta sama rozproszona aplikacja. W aplikacji internetowej można wyróżnić dwa sposoby pobierania danych: po stronie serwera, a więc jeszcze zanim zostanie przesłany kod HTML do klienta, oraz po stronie klienta (przeglądarki).

Pobieranie danych po stronie serwera (.NET, C#)

Aby pobrać dane z Web API po stronie serwera (w kodzie C# wykonywanym na serwerze, a nie w przeglądarce), należy użyć klienta HTTP. Przykładowy kod pobierający dane ogłoszeń za pomocą klienta HTTP został zamieszczony na listingu 6.13.

Listing 6.13. Pobieranie danych po stronie serwera — .NET, C#

```
using (var client = new HttpClient())
{
    client.BaseAddress = new Uri("http://aspnetmvc.p1/");
    client.DefaultRequestHeaders.Accept.Clear();
    client.DefaultRequestHeaders.Accept
        ↳.Add(new MediaTypeWithQualityHeaderValue("application/json"));

    HttpResponseMessage response =
        ↳await client.GetAsync("api/ogloszenia/1");
    if (response.IsSuccessStatusCode)
    {
        Ogloszenie ogloszenie = await
            ↳response.Content.ReadAsAsync<Ogloszenie>();
        Console.WriteLine("{0}\t{1}", ogloszenie.Tytul,
            ↳ogloszenie.Tresc);
    }
}
```

Pobieranie danych po stronie klienta (JavaScript, jQuery, AJAX)

Istnieje również możliwość pobierania danych po stronie klienta (listing 6.14). Przeglądarka internetowa łączy się z serwisem Web API przy użyciu języka JavaScript lub biblioteki jQuery, wykorzystując asynchroniczne żądania AJAX niewymagające odświeżania strony WWW.

Listing 6.14. Pobieranie danych po stronie klienta (przeglądarki) — jQuery i AJAX

```
<script type="text/javascript">
    function GetOgloszenia() {
        jQuery.support.cors = true;
        $.ajax({
            ↳url: 'http://aspnetmvc.p1/api/Ogloszenia',
            ↳type: 'GET'.
```

```
    dataType: 'json',
    success: function (data) {
        WyswietlDane(data);
    },
    error: function () {...}
});
</script>
```

Wersjonowanie w Web API

Najpopularniejszym i najbardziej poprawnym sposobem na wykorzystanie kilku wersji Web API w jednej aplikacji jest udostępnienie różnych wersji pod innym adresem:

- ◆ <http://NaszPortal.pl/api/v1/Ogloszenia/>,
- ◆ <http://NaszPortal.pl/api/v2/Ogloszenia/>.

Drugim podejściem jest utworzenie metod z dopiskiem wersji, którą obsługują:

- ◆ <http://NaszPortal.pl/api/Ogloszenia-v1/>,
- ◆ <http://NaszPortal.pl/api/Ogloszenia-v2/>.

Trzecie podejście to podawanie wersji jako parametru — w tej samej metodzie w zależności od wersji zostaje wykonany inny kod:

- ◆ <http://NaszPortal.pl/api/Ogloszenia?v=1>,
- ◆ <http://NaszPortal.pl/api/Ogloszenia?v=2>.

Rozdział 7.

Narzędzia, licencje i ceny

Poza elementami programistycznymi warto również zwrócić uwagę na stosowane narzędzia. Ich użycie jest związane z posiadaniem odpowiednich licencji oraz kosztami, które trzeba ponieść.

Serwer IIS

IIS (ang. *Internet Information Services*) to wydajny serwer sieci Web. IIS jest konkurencją dla takich serwerów jak Apache oraz nginx. Służy do hostingu aplikacji i usług. Posiada modularną budowę, co sprawia, że można łatwo instalować i usuwać dodatkowe moduły. Im większa liczba zainstalowanych modułów, tym większa podatność na ataki, dlatego instaluje się tylko te moduły, które są potrzebne¹.

Kategorie dla modułów dostępnych w IIS

Moduły IIS dzielą się na kilka podstawowych grup:

- ◆ *HTTP,*
- ◆ *Security,*
- ◆ *Content,*
- ◆ *Compression,*
- ◆ *Cache,*
- ◆ *Logging and Diagnostics,*
- ◆ *Managed support.*

¹ <http://www.iis.net/learn/get-started/introduction-to-iis/introduction-to-iis-architecture>

Pule aplikacji w IIS

Pula aplikacji to grupa zawierająca jedną lub więcej aplikacji korzystających z tych samych zasobów. Na jednym serwerze IIS można tworzyć wiele pól aplikacji. Do każdej puli aplikacji przypisuje się określone zasoby. W przypadku gdy aplikacji z innej puli zacznie brakować zasobów, nie wpłynie to na wydajność aplikacji z innych pul. Jeśli aplikacje znajdują się w jednej puli, zasoby są dzielone pomiędzy nimi i jedna aplikacja może blokować drugą, wykorzystując 100% zasobów przeznaczonych dla danej puli. Każda pula działa w osobnym procesie, co powoduje, że pule aplikacji zwiększą odporność na nieoczekiwane błędy. W przypadku gdy aplikacja z jednej puli przerwie działanie lub wystąpi jakiś problem, aplikacje w innych pulach nadal działają normalnie. Pule aplikacji zwiększą zatem również bezpieczeństwo, ponieważ aplikacje z jednej puli nie mają dostępu do danych aplikacji w innych pulach.

Przetwarzanie żądań w IIS

Pula aplikacji może pracować w jednym z dwóch trybów: *Classic Mode* lub *Integrated Pipeline*. Każda aplikacja ASP.NET MVC musi działać w trybie *Integrated Pipeline*, który zapobiega dwukrotnemu uruchamianiu tych samych modułów na serwerze IIS i w aplikacji ASP.NET. Przykładem takiego zachowania jest np. autentykacja użytkownika. W trybie *Classic Mode* autentykacja odbyłaby się dwukrotnie: na serwerze IIS i w aplikacji ASP.NET. W trybie *Integrated Pipeline* ścieżka wywołań kolejnych modułów jest wspólna dla serwera IIS i aplikacji ASP.NET. Pozwala to również na zarządzanie wszystkimi modułami z jednego miejsca w konfiguracji aplikacji lub na serwerze IIS.

Microsoft SQL Server 2014

Microsoft SQL Server to system zarządzania bazą danych, wspierany i rozpowszechniany przez korporację Microsoft. Jest to główny produkt bazodanowy tej firmy, który charakteryzuje się tym, iż jako język zapytań jest używany przede wszystkim T-SQL (ang. *Transact-SQL*), który stanowi rozszerzenie języka SQL. T-SQL pozwala na tworzenie pętli, instrukcji warunkowych i zmiennych. Za jego pomocą tworzone są także wyzwalacze, procedury i funkcje składowane².

Pełna wersja Microsoft SQL Server jest płatna, jednak Microsoft udostępnił okrojoną darmową wersję Microsoft SQL Server Express Edition, posiadającą pewne ograniczenia i mniejszą liczbę funkcjonalności. Ograniczenia SQL Server 2014 Express Edition:

- ◆ maksymalny rozmiar bazy danych — 10 GB,
- ◆ maksymalna ilość pamięci RAM — 1 GB (w serwerze może być zainstalowane więcej pamięci, ale baza danych będzie korzystała tylko z 1 GB),
- ◆ maksymalnie 1 procesor,

² <http://msdn.microsoft.com/en-us/library/cc645993.aspx>

- ♦ maksymalnie 4 rdzenie,
- ♦ brak wyszukiwania pełnotekstowego.

Licencjonowanie SQL Server 2014

W wersji pełnej (płatnej) SQL Server 2014 dostępne są dwa modele licencjonowania. Pierwszy opiera się na mocy obliczeniowej (łącznej liczbie rdzeni), a drugi na liczbie użytkowników i urządzeń. Wersje SQL Server i dostępne modele licencjonowania przedstawiono w tabeli 8.1³.

Tabela 8.1. Modele licencjonowania SQL Server

Trzy główne edycje	Modele licencjonowania	
	Server + CAL (ang. Client Access License)	Model „na rdzeń”
<i>Standard</i>	+	+
<i>Business Intelligence</i>	+	
<i>Enterprise</i>		+

Model licencjonowania „na rdzeń” jest dostępny w wersjach *Standard* i *Enterprise*. Licencje są sprzedawane w pakietach na dwa rdzenie. Nie ma możliwości zakupu licencji na jeden rdzeń. Dodatkowo obowiązuje zasada, że procesor musi mieć minimum cztery rdzenie. Jeśli ma się procesor dwurdzeniowy, istnieje konieczność wykupienia licencji na cztery rdzenie. Przy dwóch procesorach po dwa rdzenie trzeba wykupić licencję na osiem rdzeni. Jeśli np. masz dwa procesory czterordzeniowe, koszt licencji będzie taki sam jak w przypadku dwóch procesorów dwurdzeniowych.

W modelu licencjonowania *Server + CAL* na każdy serwer potrzebna jest licencja na serwer oraz po jednej licencji *CAL user* dla każdego użytkownika łączącego się z serwerem SQL lub licencji *CAL device* dla urządzeń łączących się z serwerem SQL. Licencja SQL Server zawiera w sobie licencję *CAL*, dlatego do połączenia dwóch serwerów nie są potrzebne licencje *CAL*. Licencja *CAL* pozwala na łączenie się z dowolną liczbą serwerów SQL z tą samą wersją lub starszą od wersji *CAL*. Przykładowo z licencją *CAL 2008* nie można się połączyć z SQL Server 2012 oraz z wersją 2014, jednak z licencją *CAL 2014* uda się połączyć z każdym serwerem SQL, łącznie z wersją 2014.

Ceny licencji SQL Server 2014

Ceny dla wersji *Standard*:

- ♦ *CAL + Server*:
 - ◆ licencja *CAL user* (359-06098) — ok. 1000 zł,
 - ◆ licencja *CAL device* (359-06096) — ok. 1000 zł,
 - ◆ licencja na serwer (228-10344) — ok. 4300 zł.

³ <http://www.microsoft.com/licensing/about-licensing/sql2014.aspx>

◆ *Core:*

- ◆ 2 licencje — 4 rdzenie (7NQ-00563) — ok. 17 000 zł.

Ceny dla wersji *Bussines Intelligence*:

◆ *CAL + Server:*

- ◆ licencja *CAL user* (359-06098) — ok. 1000 zł,
- ◆ licencja *CAL device* (359-06096) — ok. 1000 zł,
- ◆ licencja na serwer (D2M-00667) — ok. 40 000 zł.

Ceny dla wersji *Enterprise*:

◆ *Core:*

- ◆ 2 licencje — 4 rdzenie (7JQ-00751) — ok. 65 000 zł.

Nowości w SQL Server 2014

SQL Server 2014 wprowadza bardzo dużą liczbę nowości zarówno w silniku bazodanowym, jak i w integracji z chmurą Windows Azure. Najbardziej znaczącą nowością jest wsparcie dla In-Memory OLTP (ang. *OnLine Transaction Processing*), czyli przechowywania danych w postaci prostszych struktur w pamięci operacyjnej, co przyspiesza pracę bazy danych nawet 30-krotnie. Lista najważniejszych nowości w SQL Server 2014:

- ◆ przechowywanie danych w pamięci operacyjnej (ang. *In-Memory OLTP Engine*);
- ◆ nowości w *AlwaysOn Failover Cluster Instances* i *Availability Groups* — ulepszenia w obszarze replikacji i zabezpieczania danych na wielu serwerach;
- ◆ narzędzie do łatwego backupu danych do chmury Windows Azure (ang. *Managed Backup to Azure*);
- ◆ poprawiona integracja z Windows Server 2012;
- ◆ wstawianie, edycja i usuwanie danych indeksu (ang. *Updateable Columnstore Indexes*);
- ◆ ograniczenie zużycia zasobów dla określonych pól zasobów (ang. *Resource Governor Enhancements for Physical IO Control*);
- ◆ możliwość zwracania danych z pamięci jeszcze przed zapisem na dysku (ang. *Delayed Durability*);
- ◆ zapis danych na dysku SSD, gdy brakuje pamięci (ang. *SSD Buffer Pool Extension*).

Windows Server 2012

Microsoft Windows Server to system operacyjny przeznaczony do serwerów. Windows Server 2012 to odpowiednik serwerowy systemu Windows 8 dla komputerów osobistych.

Windows Server 2012 może być instalowany w dwóch trybach:

- ♦ Po instalacji systemu Windows Server 2012 w trybie *Server Core* istnieje dostęp tylko do linii poleceń, bez interfejsu użytkownika. Tryb *Server Core* jest rekommendowaną wersją instalacji. Z linii komend dostępnych jest ok. 2300 komend interpretowanych przez powłokę *PowerShell*. Dzięki tak dużej liczbie komend można wykonać każdą czynność dostępną w wersji z interfejsem użytkownika.
- ♦ Po instalacji systemu Windows Server 2012 w trybie *Server z GUI* otrzymuje się system z interfejsem *Metro* oraz dostępem do linii poleceń.

Wersje Windows Server 2012

Windows Server 2012 dostępny jest w kilku wersjach:

- ♦ *Datacenter* — pełna wersja systemu z nieograniczoną liczbą instancji wirtualnych;
- ♦ *Standard* — pełna wersja systemu z ograniczeniem do dwóch instancji wirtualnych;
- ♦ *Essentials* — ograniczone funkcje systemu, maksymalnie dwa procesory, maksymalnie 25 użytkowników;
- ♦ *Foundation* — ograniczone funkcje systemu, maksymalnie jeden procesor, maksymalnie 15 użytkowników.

Licencjonowanie Windows Server 2012

Licencjonowanie systemów Windows Server 2012 jest oparte na modelu *Server + CAL*. Licencjonowanie to odnosi się tylko do wersji *Standard* i *Datacenter*. Każda licencja na serwer obejmuje maksymalnie dwa procesory. Jeśli np. masz serwer z jednym lub dwoma procesorami, konieczna jest tylko jedna licencja, a jeśli masz trzy lub cztery procesory w jednym serwerze, potrzebne są dwie licencje na jeden serwer. Wersja *Standard* posiada dodatkowo ograniczenie do dwóch instancji wirtualnych na licencję, zatem jeśli ma się więcej instancji wirtualnych na jednym serwerze, należy wykupić kolejne licencje na serwer. Licencje *CAL* działają na takiej samej zasadzie jak w SQL Server. Każdy użytkownik lub urządzenie łączące się z Windows Server musi mieć wykupioną licencję *CAL*.

Wersje *Essentials* i *Foundation* nie wymagają licencji *CAL* ani *Server*. Posiadają jedynie ograniczenie na liczbę użytkowników i procesorów. Po przekroczeniu liczby użytkowników (15 lub 25) konieczne jest przejście na wyższą wersję Windows Server, czyli wersje *Standard* lub *Datacenter*.

Ceny Windows Server 2012

Ceny dla wersji *Datacenter*:

- ◆ *CAL + Server*:
 - ◆ licencja *CAL user* (R18-04281) — ok. 160 zł,
 - ◆ licencja *CAL device* (R18-04277) — ok. 140 zł,
 - ◆ licencja na serwer — 2 rdzenie (P71-07236) — ok. 23 000 zł.

Ceny dla wersji *Standard*:

- ◆ *CAL + Server*:
 - ◆ licencja *CAL user* (R18-04281) — ok. 160 zł,
 - ◆ licencja *CAL device* (R18-04277) — ok. 140 zł,
 - ◆ licencja na serwer — 2 rdzenie (P73-05762) — ok. 4200 zł.

Cena dla wersji *Essentials*:

- ◆ licencja na serwer do 25 użytkowników (G3S-00548) — ok. 2400 zł.

Cena dla wersji *Foundation*:

- ◆ licencja na serwer do 15 użytkowników — ok. 800 zł.

Microsoft Visual Studio 2013 Ultimate⁴

Microsoft Visual Studio to zintegrowane środowisko programistyczne IDE (ang. *Integrated Development Environment*) firmy Microsoft, czyli zestaw aplikacji pozwalających tworzyć, testować i modyfikować oprogramowanie. Umożliwia tworzenie aplikacji konsolowych, a także aplikacji z graficznym interfejsem użytkownika GUI (ang. *Graphical User Interface*), jak WF (ang. *Windows Forms*) lub WPF (ang. *Windows Presentation Foundation*), oraz aplikacji i stron internetowych (ASP.NET). Aplikacje mogą być pisane na platformy: Microsoft Windows, Windows Mobile, Windows CE, .NET Framework oraz Microsoft Silverlight. Visual Studio pozwala na instalowanie rozszerzających jego funkcjonalność wtyczek, zawiera edytor kodu z *IntelliSense* (automatyczne uzupełnianie kodu), a także debugger działający na dwóch poziomach — kodu i maszyny. Obsługuje wiele języków programowania: C/C++, VB.NET, C#, F#, J#, a także wspiera inne języki: M, Python, Ruby, XML/XSLT, HTML/XHTML, JavaScript i CSS. Posiada kilka edytorów wizualnych wspomagających projektowanie oprogramowania, m.in. Windows Forms Designer, WPF Designer, Web Designer/Development, Class Designer, Data Designer oraz Mapping Designer. Pod koniec 2014 roku udostępniona została wersja Visual Studio Community 2013 darmowa dla większości zastosowań.

⁴ Podstawowe środowisko do tworzenia aplikacji ASP.NET.

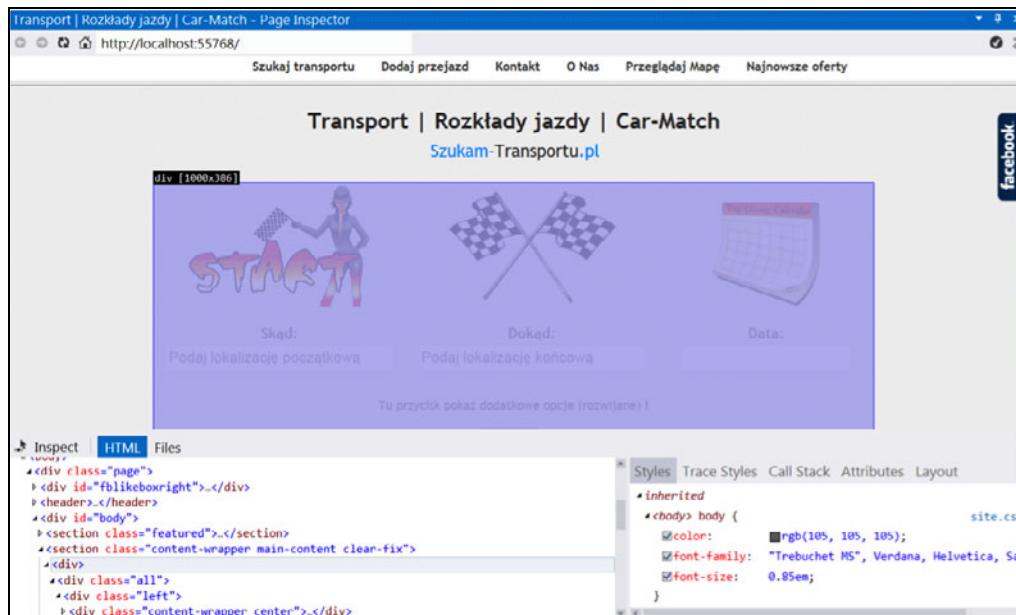
Snippety

Snippety to skróty, po wpisaniu których Visual Studio automatycznie zamienia wpisaną treść na pełną strukturę⁵. Wpisuje się np. słowo do i klika dwukrotnie klawisz *Tab*. Wpisane znaki zostaną rozwinięte w pętlę do while.

Page Inspector

Page Inspector to nowość wprowadzona w Visual Studio 2012. Pozwala na uruchomienie strony WWW bezpośrednio w oknie Visual Studio zamiast w przeglądarce. Po uruchomieniu strony w *Page Inspector* można zmieniać na żywo kod widoku oraz style CSS. Dodatkowo *Page Inspector* działa identycznie jak funkcja *Zbadaj element* w przeglądarkach. Podświetla daną część strony odnoszącą się do określonej części kodu.

Po prawej stronie znajduje się otwarte okno ze stylami CSS, natomiast po lewej kod HTML strony (rysunek 7.1).



Rysunek 7.1. Przykład działania Page Inspectora

Nowości w Visual Studio 2013

Aby móc korzystać ze wszystkich opisanych tutaj nowych narzędzi, należy zainstalować najnowszą wersję Visual Studio razem z dodatkiem Update 2 lub nowszym⁶.

⁵ <http://msdn.microsoft.com/en-us/library/z41h7fat.aspx>

⁶ <http://www.asp.net/visual-studio/overview/2013>

Poprawiony pasek przewijania

W nowym, poprawionym pasku przewijania (ang. *Enhanced Scroll Bar*) można włączyć podgląd kodu całego pliku, który będzie widoczny na pasku. Do wyboru są różne szerokości paska; można też całkowicie wyłączyć podgląd. Podczas poruszania myszką po pasku można włączyć podgląd kodu, który będzie widoczny po lewej stronie paska (rysunek 7.2). Standardowo niebieska linia na pasku oznacza, gdzie jest aktualnie kurSOR, natomiast czerwone kropki wskazują błędy.

Rysunek 7.2.

Przykładowy pasek przewijania



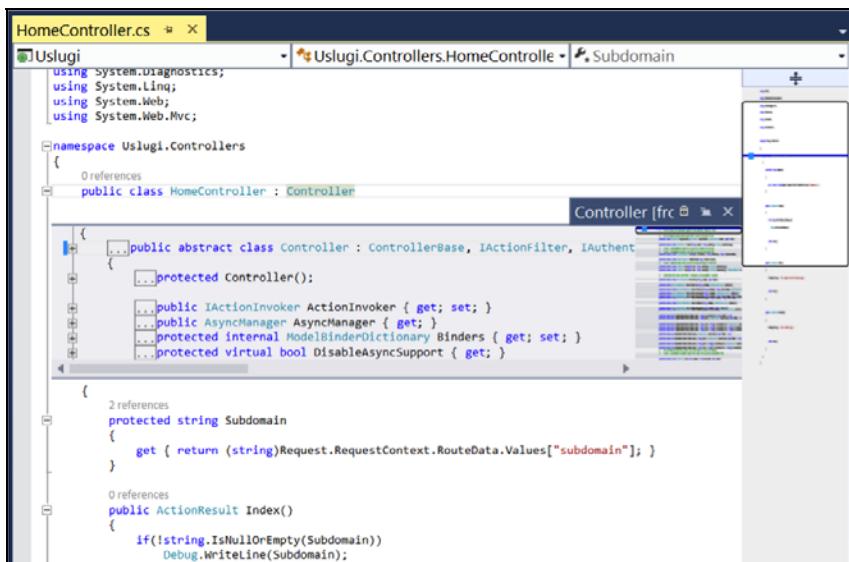
Podgląd definicji

Podgląd definicji (ang. *Peek Definition*) pozwala na szybki podgląd definicji dla wybranej klasy. Działa na identycznej zasadzie jak opcja *Go to Definition*. Jednak *Peek Definition* zamiast otwierać nowe okno w miejscu obecnego, tak jak to robi *Go to Definition*, otwiera w tym samym oknie, na ciemniejszym tle, podgląd klasy bazowej. Uogólniając, jest to okno w oknie (rysunek 7.3).

Browser Link

Browser Link pozwala na połączenie dowolnej liczby przeglądarek z jednym projektem. Działa dzięki wykorzystaniu technologii *SignalR*. *Browser Link* może działać w kilku trybach:

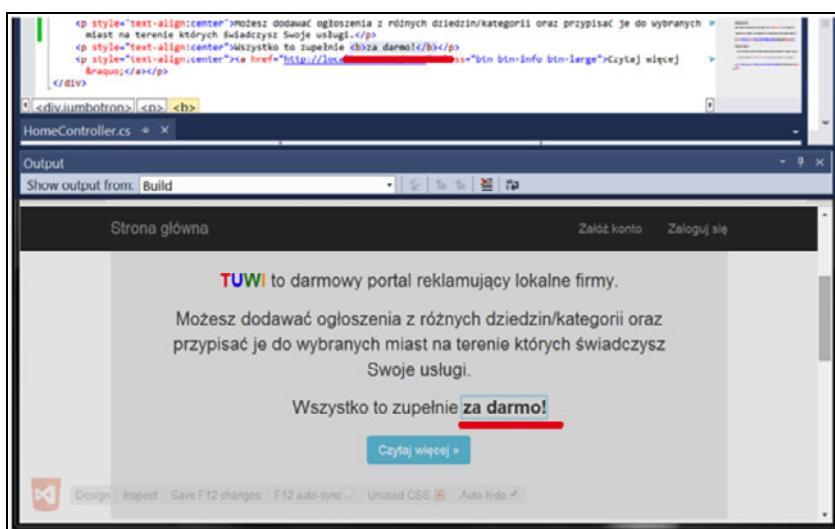
- ◆ *Refresh* — automatyczne odświeżanie strony we wszystkich powiązanych z projektem przeglądarkach;
- ◆ *Design Mode* — pozwala na edycję treści w przeglądarce; dane są automatycznie zmieniane w odpowiednim pliku w Visual Studio;
- ◆ *Inspect Mode* — inspekcje zawartości, jak w rozszerzeniu *Page Inspector*, jednak strona nie musi być otworzona w *Page Inspector* jako okno w Visual Studio; podczas przechodzenia myszką po kolejnych elementach strony w wybranej przeglądarce automatycznie zostaje podświetlony kod w pliku, który jest odpowiedzialny za daną część strony (działa podobnie jak opcja *Zbadaj element* w przeglądarkach internetowych).



Rysunek 7.3. Przykładowe „okno w oknie”

Przykład działania trybu *Inspect Mode* obrazuje automatyczne odnalezienie odpowiedniego pliku i podświetlenie tej samej treści w kodzie aplikacji Visual Studio po nажęciu na tekst *za darmo!* w przeglądarce internetowej (rysunek 7.4).

Rysunek 7.4.
Przykład
działania trybu
Inspect Mode



JSON Editor i JavaScript

W najnowszej wersji Visual Studio zostały poprawione podpowiedzi dla kodu JavaScript oraz koloryzacja składni i walidacja dla danych w formacie JSON.

Powiązanie z Microsoft Azure

W wielu miejscach zostały dodane dodatkowe opcje i okna pozwalające na prostszą integrację aplikacji z chmurą Azure już podczas tworzenia nowego projektu.

Wsparcie dla GIT

W wersji Visual Studio 2013 Update 2 zostały dodane nowe opcje w oknach dialogowych dla wsparcia repozytorium kodu GIT. Zostały dodane takie opcje jak: *amending commits*, *pushing to multiple remotes* i *reverting a commit all*.

Najważniejsze skróty klawiszowe

W poniższych tabelach przedstawiono używane skróty klawiaturowe. W tabeli 7.2 są wymienione skróty ogólne, w tabeli 7.3 — związane z projektami i plikami, w tabeli 7.4 — skróty do nawigacji w tekście, w tabeli 7.5 — skróty służące do zaznaczania tekstu, w tabeli 7.6 — do zarządzania oknami, w tabeli 7.7 — do wyszukiwania i wprowadzania zmian, a w tabeli 7.8 — skróty debuggera.

Tabela 7.2. Skróty ogólne

Skrót	Opis
<i>Ctrl+X</i>	Wycina aktualnie zaznaczone elementy do schowka
<i>Ctrl+C</i>	Kopiuje aktualnie zaznaczone elementy do schowka
<i>Ctrl+V</i>	Wkleja element ze schowka
<i>Ctrl+Z</i>	Wróć
<i>Ctrl+Y</i>	Powtóż
<i>Ctrl+S</i>	Zapisz
<i>Ctrl+Shift+S</i>	Zapisz wszystko (dokumenty i projekty)
<i>Ctrl+P</i>	Wyświetla okno dialogowe drukowania
<i>F7</i>	Zamienia z <i>Design View</i> na <i>Code View</i> w edytorze
<i>Shift+F7</i>	Zamienia z <i>Code View</i> na <i>Design View</i> w edytorze
<i>F8</i>	Przenosi do kolejnego elementu (na przykład w oknie <i>Szukaj</i>)
<i>Shift+F8</i>	Przenosi do poprzedniego elementu (na przykład w oknie <i>Szukaj</i>)
<i>Shift+F12</i>	Wyszukuje referencje do zaznaczonego elementu
<i>Ctrl+Shift+F12</i>	Przechodzi do kolejnego elementu na liście zadań/błędów
<i>Ctrl+. (kropka)</i>	Wyświetla menu podpowiedzi (ang. <i>SmartTag</i>)
<i>Ctrl+spacja</i>	Wywołanie <i>IntelliSense</i>
<i>Ctrl+K+D</i>	Automatycznie formatuje kod
<i>Ctrl+K+C</i>	Komentuje zaznaczony kod
<i>Ctrl+K+U</i>	Usuwa komentarz dla zaznaczonego kodu
<i>F5</i>	Start aplikacji z rozpoczęciem debugowania
<i>Ctrl+F5</i>	Start aplikacji bez debugowania

Tabela 7.2. Skróty ogólne — ciąg dalszy

Skrót	Opis
<i>Ctrl+Shift+B</i>	Kompilacja projektu — budowanie solucji
<i>Ctrl+Break</i>	Zatrzymanie komplikacji projektu

Tabela 7.3. Skróty związane z projektem i plikami

Skrót	Opis
<i>Ctrl+Shift+B</i>	Kompilacja projektu — budowanie solucji
<i>Ctrl+N</i>	Otwiera okno dialogowe <i>Nowy plik</i> (powstały plik nie znajduje się w projekcie)
<i>Ctrl+Shift+N</i>	Otwiera okno dialogowe <i>Nowy projekt</i>
<i>Ctrl+O</i>	Otwiera okno dialogowe <i>Otwórz plik</i>
<i>Ctrl+Shift+O</i>	Otwiera okno dialogowe <i>Otwórz projekt</i>
<i>Shift+Alt+A</i>	Otwiera okno dialogowe <i>Dodaj istniejący</i>
<i>Ctrl+Shift+S</i>	Otwiera okno dialogowe <i>Dodaj nowy</i>

Tabela 7.4. Nawigacja po tekście

Skrót	Opis
<i>Page Down</i>	Przechodzi o ekran w dół
<i>Page Up</i>	Przechodzi o ekran w górę
<i>End</i>	Przechodzi do końca aktualnej linii
<i>Home</i>	Przechodzi do początku aktualnej linii
<i>Ctrl+End</i>	Przechodzi do końca dokumentu
<i>Ctrl+Home</i>	Przechodzi do początku dokumentu
<i>Ctrl+G</i>	Otwiera okno dialogowe <i>Idź do linii nr:</i>
<i>Ctrl+J</i>	Przenosi pomiędzy otwierającymi i zamkającymi klamrami/nawiasami
<i>Ctrl+↓</i>	Przewija dokument o jedną linię w dół bez poruszania kurSORA
<i>Ctrl+↑</i>	Przewija dokument o jedną linię w górę bez poruszania kurSORA
<i>Ctrl+→</i>	Przenosi kurSOR o jedno słowo w prawo
<i>Ctrl+←</i>	Przenosi kurSOR o jedno słowo w lewo
<i>Ctrl+M+L</i>	Ukrywa bądź pokazuje wszystkie sekcje kodu
<i>Ctrl+K+K</i>	Ustawia lub kasuje <i>Dodaj do ulubionych</i> dla aktualnej linii
<i>Ctrl+M+M</i>	Ukrywa bądź pokazuje aktualną sekcję kodu
<i>Ctrl+K+H</i>	Ustawia lub kasuje skrót w liście zadań dla aktualnej linii
<i>Ctrl+R+W</i>	Ukrywa bądź pokazuje wcięcia i odstępy
<i>Ctrl+Delete</i>	Kasuje słowo na prawo od kurSORA
<i>Ctrl+Backspace</i>	Kasuje słowo na lewo od kurSORA
<i>Ctrl+Shift+T</i>	Zamienia kolejność dwóch kolejnych wyrazów po kurSORZE

Tabela 7.5. Zaznaczanie tekstu

Skrót	Opis
<i>Shift</i> +→	Przenosi kurSOR o jeden znak w prawo, rozszerza zaznaczenie
<i>Shift</i> + <i>Alt</i> +→	Przenosi kurSOR o jeden znak w prawo, rozszerza zaznaczenie kolumnowo
<i>Shift</i> +←	Przenosi kurSOR o jeden znak w lewo, rozszerza zaznaczenie
<i>Shift</i> + <i>Alt</i> +←	Przenosi kurSOR o jeden znak w lewo, rozszerza zaznaczenie kolumnowo
<i>Shift</i> +↓	Przenosi kurSOR o jedną linię w dół, rozszerza zaznaczenie
<i>Shift</i> + <i>Alt</i> +↓	Przenosi kurSOR o jedną linię w dół, rozszerza zaznaczenie kolumnowo
<i>Shift</i> +↑	Przenosi kurSOR o jedną linię w góre, rozszerza zaznaczenie
<i>Shift</i> + <i>Alt</i> +↑	Przenosi kurSOR o jedną linię w góre, rozszerza zaznaczenie kolumnowo
<i>Ctrl</i> + <i>Shift</i> + <i>End</i>	Przenosi kurSOR na koniec dokumentu, rozszerza zaznaczenie
<i>Ctrl</i> + <i>Shift</i> + <i>Home</i>	Przenosi kurSOR na początek dokumentu, rozszerza zaznaczenie
<i>Ctrl</i> + <i>Shift</i> +J	Przenosi pomiędzy klamrami/nawiasami, rozszerza zaznaczenie
<i>Shift</i> + <i>End</i>	Przenosi kurSOR na koniec aktualnej linii, rozszerza zaznaczenie
<i>Shift</i> + <i>Alt</i> + <i>End</i>	Przenosi kurSOR na koniec linii, rozszerza zaznaczenie kolumnowo
<i>Shift</i> + <i>Home</i>	Przenosi kurSOR na początek aktualnej linii, rozszerza zaznaczenie
<i>Shift</i> + <i>Alt</i> + <i>Home</i>	Przenosi kurSOR na początek linii, rozszerza zaznaczenie kolumnowo
<i>Shift</i> + <i>Page Down</i>	Rozszerza zaznaczenie o stronę w dół
<i>Shift</i> + <i>Page Up</i>	Rozszerza zaznaczenie o stronę w góre
<i>Ctrl</i> +A	Zaznacza wszystko
<i>Ctrl</i> +W	Zaznacza słowo przy kurSORze lub na prawo od kurSORa
<i>Ctrl</i> + <i>Shift</i> +←	Przenosi kurSOR o jedno słowo w lewo, rozszerza zaznaczenie
<i>Ctrl</i> + <i>Shift</i> +→	Przenosi kurSOR o jedno słowo w prawo, rozszerza zaznaczenie

Tabela 7.6. Zarządzanie oknami

Skrót	Opis
<i>Shift</i> + <i>Alt</i> + <i>Enter</i>	Włącza lub wyłącza tryb pełnoekranowy
<i>Shift</i> + <i>Alt</i> +F6	Przechodzi do ostatnio otwartej zakładki
<i>Ctrl</i> +F6	Przechodzi do następnego okna
<i>Ctrl</i> + <i>Shift</i> + <i>Tab</i>	Przechodzi do następnego okna (otwiera listę okien)
<i>Ctrl</i> +F4	Zamyka aktualnie otwarte okno/zakładkę

Tabela 7.7. Szukaj i zamień

Skrót	Opis
<i>Ctrl</i> +F	Wyświetla okno dialogowe <i>Szukaj</i>
<i>Ctrl</i> + <i>Shift</i> +F	Wyświetla okno dialogowe <i>Szukaj w plikach</i>
F3	Przechodzi do kolejnego wystąpienia wyszukiwanego tekstu
<i>Ctrl</i> +F3	Wyszukuje i przechodzi do kolejnego wystąpienia tekstu spod kurSORa
<i>Shift</i> +F3	Wyszukuje i przechodzi do poprzedniego wystąpienia tekstu spod kurSORa

Tabela 7.7. Szukaj i zamień — ciąg dalszy

Skrót	Opis
<i>Ctrl+I</i>	Szybkie wyszukiwanie w tekście (przechodzi do kolejnego wystąpienia)
<i>Ctrl+Shift+I</i>	Szybkie wyszukiwanie w tekście (przechodzi do poprzedniego wystąpienia)
<i>Ctrl+H</i>	Wyświetla okno dialogowe <i>Zamień</i>
<i>Ctrl+Shift+H</i>	Wyświetla okno dialogowe <i>Zamień w plikach</i>

Tabela 7.8. Debugger

Skrót	Opis
<i>F5</i>	Start aplikacji z rozpoczęciem debugowania
<i>Shift+F5</i>	Zatrzymanie debugowania
<i>Ctrl+Alt+B</i>	Otwiera okno <i>Breakpoints</i>
<i>Ctrl+Shift+F9</i>	Kasuje wszystkie <i>breakpointy</i> (punkty przerwania) w projekcie
<i>F9</i>	Ustawia lub usuwa <i>breakpoint</i>
<i>Ctrl+B</i>	Otwiera okno dialogowe <i>New Breakpoint</i>
<i>Ctrl+F9</i>	Włącza lub wyłącza <i>breakpoint</i> (nie usuwa, tylko deaktywuje)
<i>Ctrl+Alt+W, I</i>	Otwiera okno <i>Watch1</i> do podglądu zmiennych
<i>F11</i>	<i>Step Into</i> — wykonuje kolejną linijkę kodu
<i>Shift+F11</i>	<i>Step Out</i> — wykonuje cały kod metody
<i>F10</i>	<i>Step Over</i> — wykonuje kolejną linijkę kodu, jednak nie wchodzi do funkcji

Rozdział 8.

Aplikacja i wdrożenie

Przed rozpoczęciem tworzenia przykładowej aplikacji kilka słów wstępna na temat architektury aplikacji.

Wzorce projektowe i architektoniczne wykorzystywane w .NET

Repozytorium

Standardowo zapytania do bazy danych znajdują się bezpośrednio w kontrolerze lub w klasach z modelem. Zgodnie z wzorcem repozytorium wszystkie operacje korzystające z bazy danych są przenoszone do innej warstwy w aplikacji (osobnego projektu). W kontrolerze wywołuje się tylko metody z repozytorium. Z kilku kontrolerów można wywoływać tę samą metodę bez konieczności kopiowania lub ponownego pisania. Kontroler nie odpowiada za utworzenie kontekstu (obiektu dostępu do danych) i nie posiada informacji na temat tego, skąd są dane ani jaką strukturę ma baza danych.

Wzorzec IoC

Repozytorium ma jeden duży minus — instancja repozytorium jest tworzona w kontrolerze, co powoduje, że każdy kontroler jest zależny nie od kontekstu, a od repozytorium. Aby tworzenie obiektu repozytorium było niezależne od kontrolera, wykorzystuje się wzorzec IoC (ang. *Inversion of Control*). Dzięki takiemu rozwiązaniu instancja repozytorium jest wstrzykiwana poprzez konstruktor. IoC pozwala na działanie na interfejsie zamiast na określonej klasie. Dzięki temu można w łatwy sposób podmienić implementację repozytorium. W konfiguracji biblioteki odpowiedzialnej za IoC wybiera się, jaka klasa ma być wstrzyknięta dla danego interfejsu. Może być na przykład kilka wersji tego samego repozytorium.

Repozytorium generyczne

Bez względu na rodzaj klasy, na której się działa, istnieje kilka podstawowych metod wykorzystywanych w przypadku większości klas. Takimi metodami są na przykład operacje dodawania lub usuwania. Aby nie pisać tych samych deklaracji metod w każdym repozytorium, należy utworzyć generyczny interfejs repozytorium, po którym będą dziedziczyć wszystkie pozostałe interfejsy lub repozytoria. Dzięki repozytorium generycznemu nie ma potrzeby przepisywania tych samych metod do różnych interfejsów lub klas. Według niektórych wzorzec ten jest uznawany za antywzorzec.

Wzorzec UnitOfWork

Wzorzec UnitOfWork pozwala na grupowanie kilku zadań z różnych repozytoriów w jedną transakcję. Służy do dzielenia pojedynczego kontekstu pomiędzy różne repozytoria. UnitOfWork znajduje zastosowanie, gdy chce się wprowadzić zmiany w wielu tabelach za pomocą wielu repozytoriów na pojedynczym kontekście. W przypadku bardzo wielu pojedynczych, prostych zapytań do bazy danych z różnych repozytoriów tworzone są osobne konteksty, a każde zapytanie jest wysyłane jako pojedyncze żądanie. Korzystając z UnitOfWork, oszczędza się zasoby serwera, ponieważ nie trzeba tworzyć wielu osobnych połączeń z bazą ani wielu kontekstów. Wszystkie operacje korzystają z pojedynczego kontekstu i pojedynczego połączenia.

Przykładowa aplikacja

Przykładową aplikacją będzie serwis z ogłoszeniami utworzony w Visual Studio 2013 Update 3. Każde ogłoszenie będzie przypisane do wielu kategorii. Jedna kategoria będzie miała przypisanych wiele ogłoszeń. Jeden użytkownik będzie dodawał wiele ogłoszeń. Ogłoszenie będzie miało tylko jednego autora bądź właściciela. Takie założenia zostały przyjęte, aby zademonstrować powiązania „jeden do wielu” oraz „wiele do wielu”. W tabeli Kategoria zostaną dodane pola MetaTytuł, MetaOpis oraz SłowaKluczowe, aby zoptymalizować portal pod wyszukiwarkę Google. Do uwierzytelnienia za pośrednictwem zewnętrznych serwisów wykorzystamy Google. Facebook wymaga utworzenia aplikacji na Facebooku, w której podaje się adres portalu, aby istniała możliwość korzystania z logowania za jego pośrednictwem. Ponieważ Google nie wymaga tworzenia oddzielnych aplikacji, a zasada działania jest prawie identyczna, skorzystamy z logowania za pośrednictwem Google. Wystarczy zalogować się na swoje konto Gmail i na tej podstawie użytkownik zostaje uwierzytelniony na portalu.

Wszystkie kody etapów i kroków dostępne są w archiwum na serwerze FTP wydawnictwa Helion pod adresem <ftp://ftp.helion.pl/przyklady/c6mvc5.zip>. Działająca aplikacja i odpowiedzi na pytania czytelników zostały udostępnione pod adresem <AspNetMvc.pl>¹.

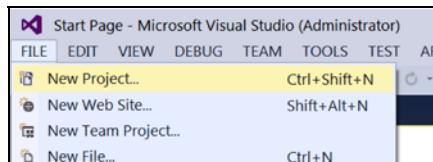
¹ Konto książki na Facebooku: <https://www.facebook.com/CSharp6MVC5>.

Etap 1. Krok 1. Tworzenie nowego projektu i aktualizacja pakietów

Z menu *FILE* w Visual Studio wybierz *New Project...* (rysunek 8.1).

Rysunek 8.1.

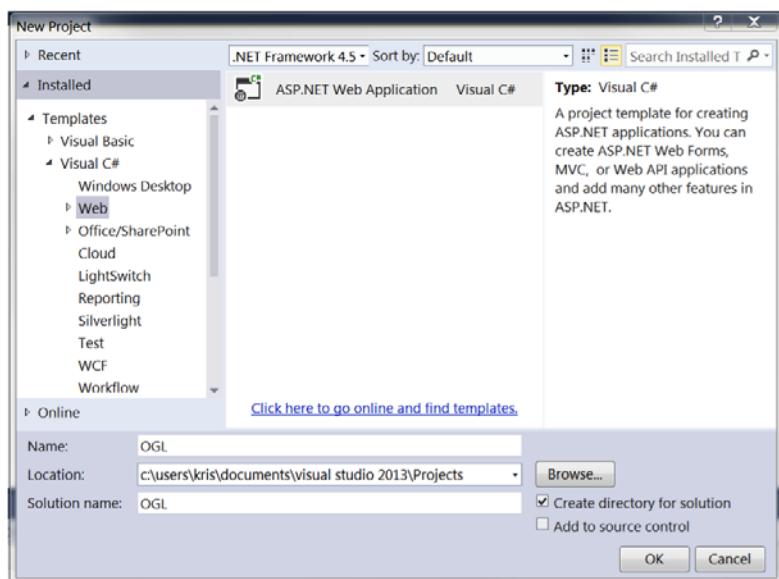
Tworzenie nowego projektu



Pojawi się nowe okno, w którym wybiera się typ tworzonego projektu. Aby utworzyć aplikację ASP.NET MVC, należy wybrać zakładkę *C#*, a następnie *Web*. Na liście aplikacji znajduje się tylko jedna pozycja — *ASP.NET Web Application*. Wybierz ją i wpisz nazwę aplikacji; w tym wypadku aplikacja będzie miała nazwę OGL (rysunek 8.2).

Rysunek 8.2.

Nowy projekt

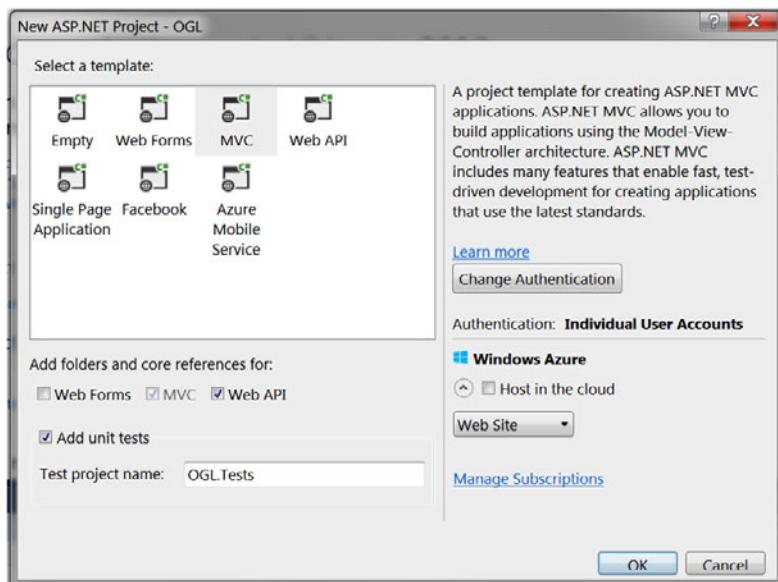


Po wpisaniu nazwy i kliknięciu przycisku *OK* pokaże się kolejne okno, w którym wybierz typ tworzonej aplikacji. W tym przypadku jest to aplikacja *MVC* (rysunek 8.3).

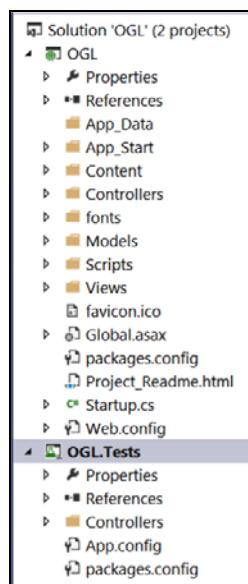
Jeśli zaznaczyś opcję *Add unit tests*, zostanie utworzony dodatkowy projekt z testami. Struktura utworzonych projektów została zaprezentowana na rysunku 8.4.

Możesz usunąć plik *Project_Readme.html*, który nie pełni żadnej istotnej funkcji. Kolejnym krokiem jest aktualizacja wszystkich bibliotek i narzędzi używanych w projekcie. Do instalacji i aktualizacji bibliotek służy narzędzie o nazwie NuGet. Aby zaktualizować wszystkie pakiety, zaznacz projekt, kliknij prawym przyciskiem myszy i wybierz z menu kontekstowego opcję *Manage NuGet Packages...* (rysunek 8.5).

Rysunek 8.3.
Wybieranie typu aplikacji



Rysunek 8.4.
Struktura projektów — okno Solution Explorer

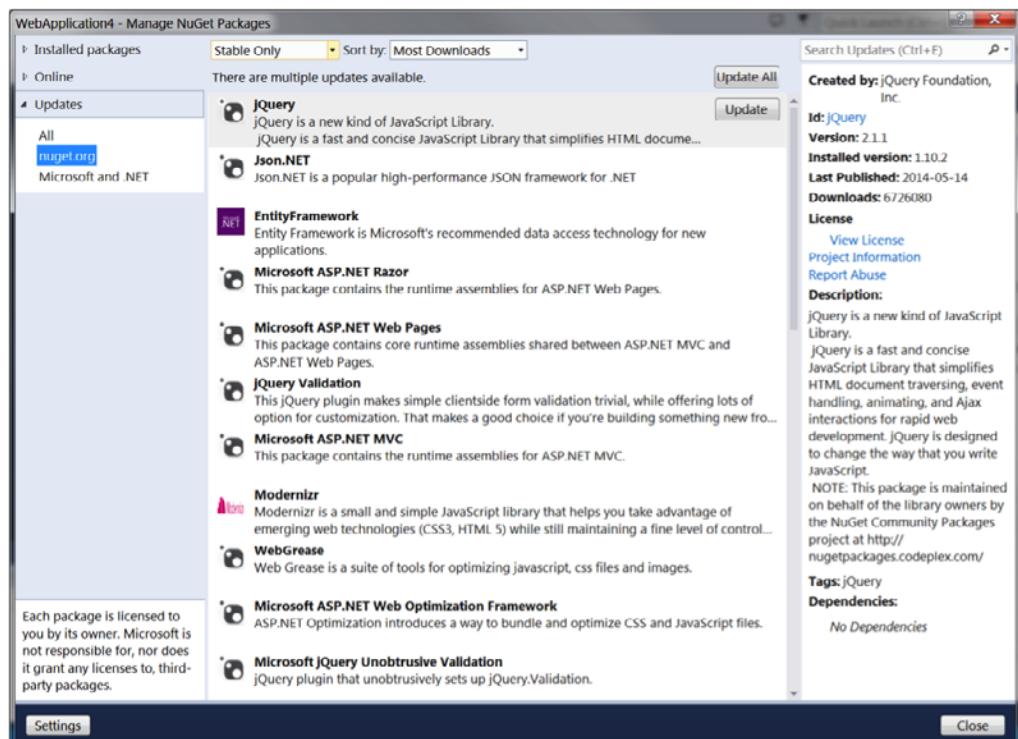
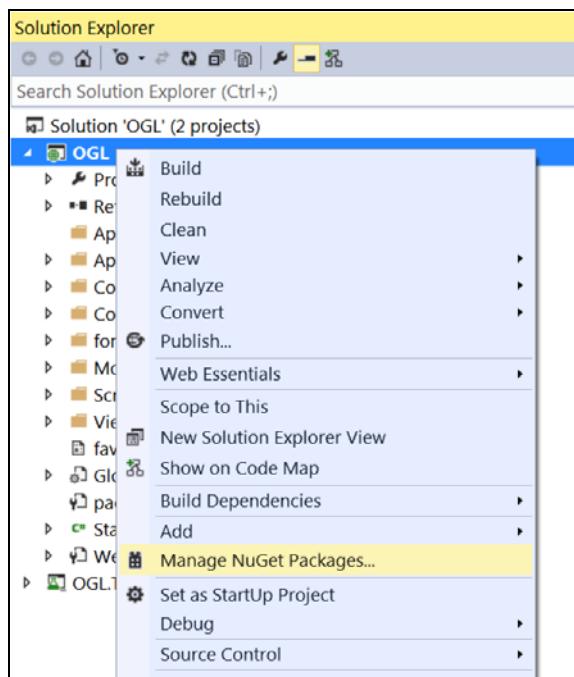


Po otwarciu okna NuGet przejdź na zakładkę *Updates/nuget.org* pokazaną na rysunku 8.6, a następnie kliknij przycisk *Update All*, dzięki czemu zaktualizujesz wszystkie zainstalowane biblioteki. Możesz również kliknąć przycisk *Update* dla każdego pakietu osobno.

Pełną listę domyślnie zainstalowanych pakietów zaprezentowano na rysunku 8.7.

Po aktualizacji będzie konieczny restart Visual Studio. Kliknij przycisk *Restart Now*, a następnie w oknie odpowiedzialnym za zapis projektu przycisk *Yes*. Visual Studio

Rysunek 8.5.
Włączenie NuGet



Rysunek 8.6. Aktualizacja bibliotek

EntityFramework Uninstall

Entity Framework is Microsoft's recommended data access technology for...

jQuery Validation

This jQuery plugin makes simple clientside form validation trivial, while offering lots of option for custo...

jQuery

jQuery is a new kind of JavaScript Library. jQuery is a fast and concise JavaScript Library that simp...

Json.NET

Json.NET is a popular high-performance JSON framework for .NET

Microsoft ASP.NET Identity Core

Core interfaces for ASP.NET Identity.

Microsoft ASP.NET Identity EntityFramework

ASP.NET Identity providers that use Entity Framework.

Microsoft ASP.NET Identity Owin

Owin implementation for ASP.NET Identity.

Microsoft ASP.NET MVC

This package contains the runtime assemblies for ASP.NET MVC.

Microsoft ASP.NET Razor

This package contains the runtime assemblies for ASP.NET Web Pages.

Microsoft ASP.NET Web API 2.1 Client Libraries

This package adds support for formatting and content negotiation to System.Net.Http.

Microsoft ASP.NET Web API 2.1 Core Libraries

This package contains the core runtime assemblies for ASP.NET Web API.

Microsoft ASP.NET Web API 2.1 Web Host

This package contains everything you need to host ASP.NET Web API on IIS.

Microsoft ASP.NET Web API 2.1

This package contains everything you need to host ASP.NET Web API on IIS.

Microsoft ASP.NET Web Optimization Framework

ASP.NET Optimization introduces a way to bundle and optimize CSS and JavaScript files.

Microsoft ASP.NET Web Pages

This package contains core runtime assemblies shared between ASP.NET MVC and ASP.NET Web Pages.

Microsoft jQuery Unobtrusive Validation

jQuery plugin that unobtrusively sets up jQuery.Validation.

Microsoft.Owin.Host.SystemWeb

OWIN server that enables OWIN-based applications to run on IIS using the ASP.NET request pipeline.

Microsoft.Owin.Security.Cookies

Middleware that enables an application to use cookie based authentication, similar to ASP.NET's forms authen...

Microsoft.Owin.Security.Facebook

Middleware that enables an application to support Facebook's OAuth 2.0 authentication workflow.

Microsoft.Owin.Security.Google

Contains middlewares to support Google's OpenId and OAuth 2.0 authentication workflows.

Microsoft.Owin.Security.MicrosoftAccount

Middleware that enables an application to support the Microsoft Account authentication workflow.

Microsoft.Owin.Security.OAuth

Middleware that enables an application to support any standard OAuth 2.0 authentication workflow.

Microsoft.Owin.Security.Twitter

Middleware that enables an application to support Twitter's OAuth 2.0 authentication workflow.

Microsoft.Owin.Security

Common types which are shared by the various authentication middleware components.

Microsoft.Owin

Provides a set of helper types and abstractions for simplifying the creation of OWIN components.

Microsoft.Web.Infrastructure

This package contains the Microsoft.Web.Infrastructure assembly that lets you dynamically register HTTP modul...

Modernizr

Modernizr is a small and simple JavaScript library that helps you take advantage of emerging web technologie...

OWIN

OWIN IAppBuilder startup interface

Respond JS

A fast & lightweight polyfill for min/max-width CSS3 Media Queries (for IE 6-8, and more)

WebGrease

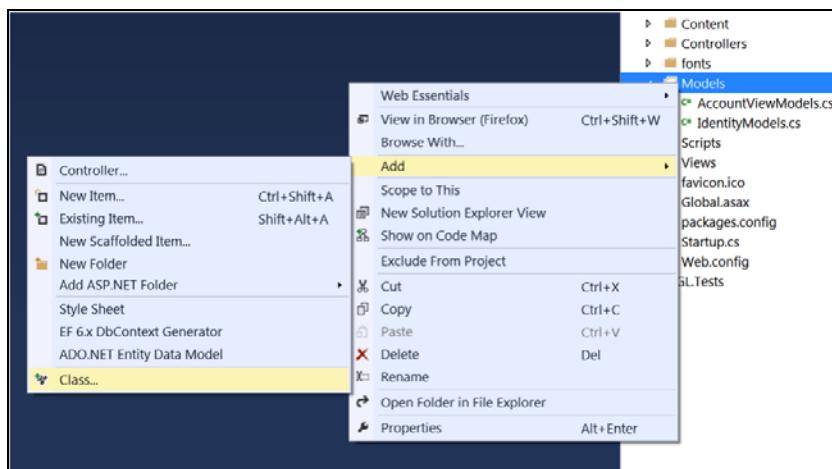
Web Grease is a suite of tools for optimizing javascript, css files and images.

Rysunek 8.7. Zainstalowane pakiety

zostanie zamknięte, a następnie automatycznie zostanie otworzona solucja OGL. W dalszej części książki korzystamy z Visual Studio 2013 Update 3, MVC 5.2 i ASP.NET Identity 2.1. Nowsze wersje, mogą posiadać inną strukturę plików, dlatego zalecane jest otworzenie solucji z folderu: *Etap1/Krok1/Po/OGL* z paczki plików udostępnionych wraz z książką i kontynuowanie pracy nad projektem od kroku nr 2.

Etap 1. Krok 2. Utworzenie modelu danych

Rozpocznij od utworzenia modelu danych. Każda aplikacja przeprowadza operacje na danych, więc jest to swego rodzaju „fundament”. Zostanie utworzona aplikacja w podejściu *Code First*, a więc będą pisane klasy z modelem. Aby dodać plik (klasę) do folderu *Models*, kliknij go prawym przyciskiem myszy i wybierz *Add/Class...* (rysunek 8.8).

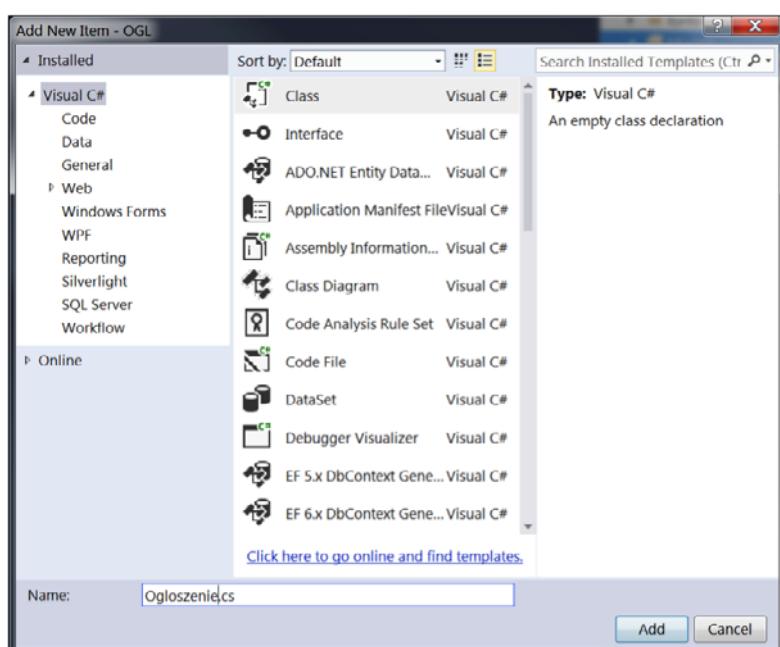


Rysunek 8.8. Dodawanie pliku do folderu

Na początku zostanie dodana klasa *Ogłoszenie*. Wpisz nazwę klasy i kliknij *OK* (rysunek 8.9).

Rysunek 8.9.

Dodawanie nowej klasy



Po kliknięciu przycisku *OK* zostanie wygenerowany następujący kod:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;

namespace OGL.Models
{
    public class Ogloszenie
    {
    }
}
```

W przestrzeni OGL.Models znajduje się klasa o nazwie Ogloszenie.

W miejsce kodu klasy Ogloszenie:

```
public class Ogloszenie
{
}
```

wpisz następujący kod:

```
public class Ogloszenie
{
    public Ogloszenie()
    {
        this.Ogloszenie_Kategoria =
            ↳new HashSet<Ogloszenie_Kategoria>();
    }
    [Display(Name = "Id")]
    //using System.ComponentModel.DataAnnotations;
    public int Id { get; set; }

    [Display(Name = "Treść ogłoszenia")]
    [MaxLength(500)]
    public string Tresc { get; set; }

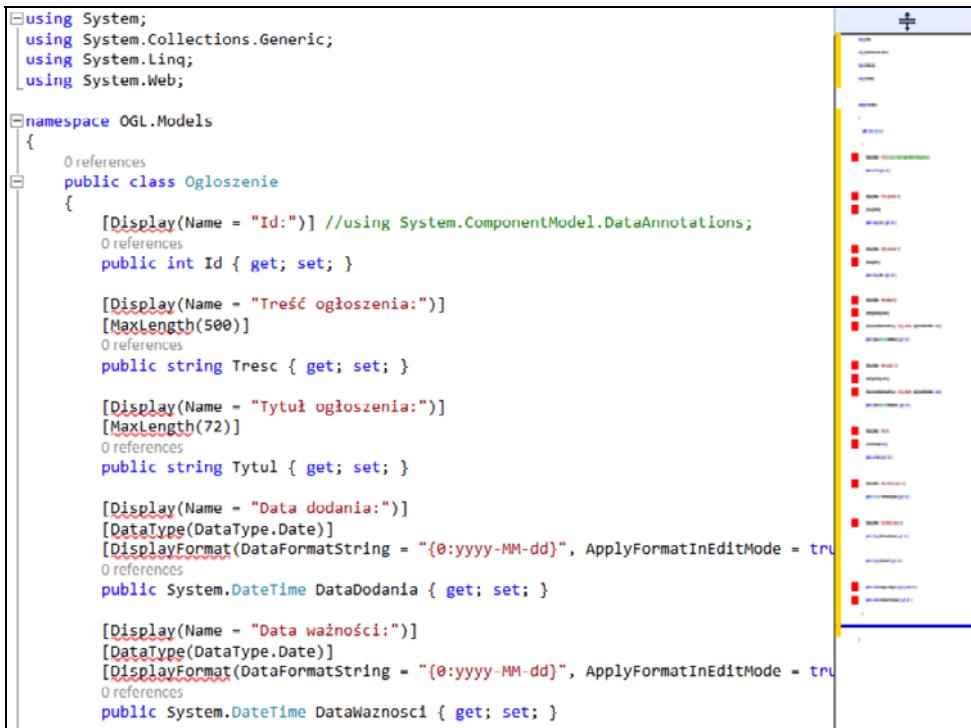
    [Display(Name = "Tytuł ogłoszenia")]
    [MaxLength(72)]
    public string Tytul { get; set; }

    [Display(Name = "Data dodania")]
    [DataType(DataType.Date)]
    [DisplayFormat(DataFormatString = "{0:yyyy-MM-dd}",
                  ApplyFormatInEditMode = true)]
    public System.DateTime DataDodania { get; set; }

    public string UzytkownikId { get; set; }

    public virtual ICollection<Ogloszenie_Kategoria>
        ↳Ogloszenie_Kategoria { get; set; }
    public virtual Uzytkownik Uzytkownik { get; set; }
}
```

Po wklejeniu kodu większość linii zostanie podświetlona na czerwono (rysunek 8.10), ponieważ nie ma zainportowanych bibliotek (przy użyciu słowa *using*), a aplikacja nie rozpoznaje użytych typów, metod ani atrybutów.



```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;

namespace OGL.Models
{
    public class Ogloszenie
    {
        [Display(Name = "Id:")]
        public int Id { get; set; }

        [Display(Name = "Treść ogłoszenia:")]
        [MaxLength(500)]
        public string Tresc { get; set; }

        [Display(Name = "Tytuł ogłoszenia:")]
        [MaxLength(72)]
        public string Tytul { get; set; }

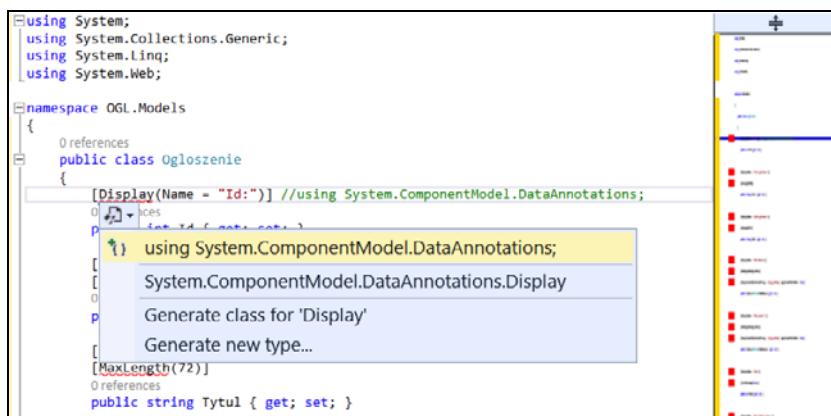
        [Display(Name = "Data dodania:")]
        [DataType(DataType.Date)]
        [DisplayFormat(DataFormatString = "{0:yyyy-MM-dd}", ApplyFormatInEditMode = true)]
        public System.DateTime DataDodania { get; set; }

        [Display(Name = "Data ważności:")]
        [DataType(DataType.Date)]
        [DisplayFormat(DataFormatString = "{0:yyyy-MM-dd}", ApplyFormatInEditMode = true)]
        public System.DateTime DataWaznosci { get; set; }
    }
}

```

Rysunek 8.10. Klasa Ogloszenie

Aby zaimportować biblioteki, kliknij podkreślony kod ([Display(Name = "Id:"))]. Pokażę się wtedy strzałka w dół, jak na rysunku 8.11. Wybierz opcję ze słowem using na początku (using System.ComponentModel.DataAnnotations;). Jeśli wybierzesz drugą opcję, czyli bez using, w wybranym miejscu w kodzie do klasy zostanie dopisana ścieżka, a kolejne wystąpienia atrybutów z tej samej klasy będą nadal podświetlane na czerwono (możesz sprawdzić, jak to działa, ponieważ nie spowoduje to żadnych problemów z działaniem aplikacji).



```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;

namespace OGL.Models
{
    public class Ogloszenie
    {
        [Display(Name = "Id:")]
        public int Id { get; set; }

        [Display(Name = "Treść ogłoszenia:")]
        [MaxLength(500)]
        public string Tresc { get; set; }

        [Display(Name = "Tytuł ogłoszenia:")]
        [MaxLength(72)]
        public string Tytul { get; set; }

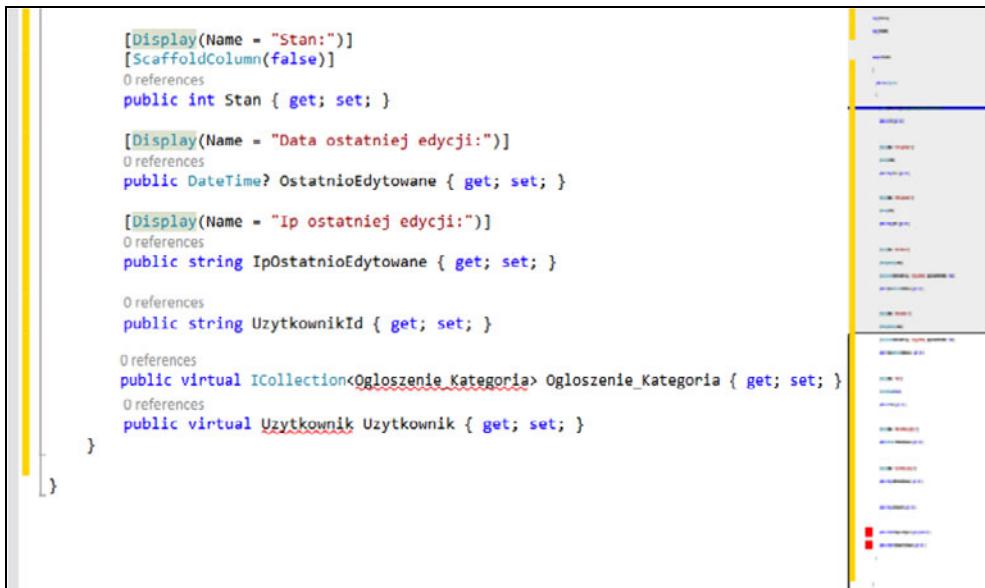
        [Display(Name = "Data dodania:")]
        [DataType(DataType.Date)]
        [DisplayFormat(DataFormatString = "{0:yyyy-MM-dd}", ApplyFormatInEditMode = true)]
        public System.DateTime DataDodania { get; set; }

        [Display(Name = "Data ważności:")]
        [DataType(DataType.Date)]
        [DisplayFormat(DataFormatString = "{0:yyyy-MM-dd}", ApplyFormatInEditMode = true)]
        public System.DateTime DataWaznosci { get; set; }
    }
}

```

Rysunek 8.11. Dodanie dyrektywy using

Po wykonanym importie dla kolejnych typów zniknie czerwone podkreślenie (rysunek 8.12) dla wszystkich pól poza tymi odnoszącymi się do klas, które zostaną dodane w kolejnym kroku (Uzytkownik i Ogloszenie_Kategoria). Zapisz plik za pomocą skrótu *Ctrl+S*.



```
[Display(Name = "Stan")]
[ScaffoldColumn(false)]
0 references
public int Stan { get; set; }

[Display(Name = "Data ostatniej edycji")]
0 references
public DateTime? OstatnioEdytowane { get; set; }

[Display(Name = "Ip ostatniej edycji")]
0 references
public string IpOstatnioEdytowane { get; set; }

0 references
public string UzytkownikId { get; set; }

0 references
public virtual ICollection<Ogloszenie_Kategoria> Ogloszenie_Kategoria { get; set; }
0 references
public virtual Uzytkownik Uzytkownik { get; set; }
}
```

Rysunek 8.12. Poprawne zimportowanie bibliotek

Na zrzutach ekranu po prawej stronie znajduje się rozszerzony pasek przewijania, na którym widać strukturę kodu oraz oznaczone są miejsca, w których jest podkreślony kod. Aby dostać się do ustawień paska, należy kliknąć pasek prawym przyciskiem myszy i wybrać opcję *Scroll Bar Options*.

Klasa Kategoria

Jest ona dodawana tak jak klasa Ogloszenie — utworzymy nowy plik o nazwie *Kategoria* w folderze *Models*. W miejsce pustej klasy Kategoria dopisz kod:

```
public class Kategoria
{
    public Kategoria()
    {
        this.Ogloszenie_Kategoria =
            new HashSet<Ogloszenie_Kategoria>();
    }

    [Key]
    [Display(Name = "Id kategorii")]
    public int Id { get; set; }

    [Display(Name = "Nazwa kategorii")]
    [Required]
    public string Nazwa { get; set; }
```

```
[Display(Name = "Id rodzica:")]
[Required]
public int ParentId { get; set; }

#region SEO

[Display(Name = "Tytuł w Google:")]
[MaxLength(72)]
public string MetaTytul { get; set; }

[Display(Name = "Opis strony w Google:")]
[MaxLength(160)]
public string MetaOpis { get; set; }

[Display(Name = "Słowa kluczowe Google:")]
[MaxLength(160)]
public string MetaSlowa { get; set; }

[Display(Name = "Treść strony:")]
[MaxLength(500)]
public string Tresc { get; set; }

#endregion

public ICollection<Ogloszenie_Kategoria> Ogloszenie_Kategoria
{
    get; set;
}
```

Zainportuj biblioteki, podobnie jak dla klasy Ogloszenie, i zapisz plik.

Klasa Ogloszenie_Kategoria

Dodaj kolejny plik o nazwie *Ogloszenie_Kategoria* w folderze *Models* i dopisz odpowiedni kod:

```
public class Ogloszenie_Kategoria
{
    public Ogloszenie_Kategoria()
    {

    }

    public int Id { get; set; }
    public int KategoriaId { get; set; }
    public int OgloszenieId { get; set; }

    public virtual Kategoria Kategoria { get; set; }
    public virtual Ogloszenie Ogloszenie { get; set; }
}
```

Klasa Uzytkownik

W nowo otwartym projekcie w pliku *IdentityModels* znajduje się klasa *ApplicationUser*, która dziedziczy po *IdentityUser* (rysunek 8.13).

Dla uporządkowania aplikacji należy zmienić w kodzie nazwę klasy z *ApplicationUser* na *Uzytkownik*.

```

namespace OGL.Models
{
    // You can add profile data for the user by adding more properties to your ApplicationUser class.
    public class ApplicationUser : IdentityUser
    {
        // Note: the authenticationType must match the one defined in CookieAuthenticationOptions.AuthenticationType
        public async Task<ClaimsIdentity> GenerateUserIdentityAsync(UserManager<ApplicationUser> manager)
        {
            // Add custom user claims here
            var userIdentity = await manager.CreateIdentityAsync(this, DefaultAuthenticationType);
            return userIdentity;
        }
    }

    public class ApplicationDbContext : IdentityDbContext<ApplicationUser>
    {
        public ApplicationDbContext()
            : base("DefaultConnection", throwIfV1Schema: false)
        {
        }

        public static ApplicationDbContext Create()
        {
            return new ApplicationDbContext();
        }
    }
}

```

Rysunek 8.13. Klasa ApplicationUser

Poniższą linijkę:

```
public class ApplicationUser : IdentityUser
```

zamień na:

```
public class Uzytkownik : IdentityUser
```

Teraz we wszystkich plikach w projekcie należy zmienić nazwy z ApplicationUser na Uzytkownik. Aby zrobić to automatycznie, wystarczy skorzystać ze skrótu *Ctrl+F* i w wyświetlnym oknie kliknąć strzałkę w dół (rysunek 8.14) lub skrót *Ctrl+H*. Można również zmienić nazwę — pokaże się wówczas czerwone podkreślenie, za pomocą którego wybieramy opcję *Rename ' ApplicationUser ' to 'Uzytkownik '*. W przykładzie skorzystamy z pierwszej opcji.

Rysunek 8.14.

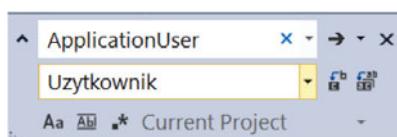
Wyszukiwanie ciągu znaków



Po kliknięciu strzałki w dół rozwinię się kolejne okno (rysunek 8.15). Wystarczy zamienić treść z ApplicationUser na Uzytkownik (wpisując w odpowiednie pola podane nazwy) i wybrać z listy rozwijanej opcję *Current Project*, aby podmienić tekst w całym projekcie OGL. Zamiana następuje po kliknięciu ikonki *Replace all* lub zastosowaniu kombinacji klawiszy *Alt+A*. Potwierdź zamianę, klikając przycisk *Yes*. Visual Studio zamieni słowa we wszystkich plikach projektu.

Rysunek 8.15.

Zamiana ciągu znaków



Następnie należy wyciąć kod klasy Uzytkownik. W folderze *Models* dodaj plik o nazwie *Uzytkownik*, w którym wklej wycięty kod. Konieczne jest zainportowanie bibliotek dla pól podkreślonych na czerwono. Po wykonaniu operacji klasa powinna wyglądać w następujący sposób:

```
public class Uzytkownik : IdentityUser
{
    public async Task<ClaimsIdentity>
        ↪GenerateUserIdentityAsync(UserManager<Uzytkownik> manager)
    {
        var userIdentity = await manager.CreateIdentityAsync(this,
            ↪DefaultAuthenticationTypes.ApplicationCookie);
        // Add custom user claims here
        return userIdentity;
    }
}
```

W tej klasie można dodawać własne pola dotyczące użytkownika. Teraz dodamy trzy pola (Imie, Nazwisko, PełneNazwisko), z czego tylko dwa (Imie, Nazwisko) zostaną dodane do tabeli Uzytkownik w bazie danych. Pole PełneNazwisko zostało oznaczone atrybutem [NotMapped], który nie pozwala tworzyć takiego pola w bazie danych. Jeśli odwołamy się do pola PełneNazwisko, Imie i Nazwisko zostaną zwrócone jako jedna wartość typu string. Wystarczy dopisać podany kod i zainportować biblioteki:

```
public class Uzytkownik : IdentityUser
{
    public Uzytkownik()
    {
        this.Ogloszenia = new HashSet<Ogloszenie>();
    }

    // Klucz podstawowy dziedziczący po klasie IdentityUser

    // Dodajemy pola Imie i Nazwisko
    public string Imie { get; set; }
    public string Nazwisko { get; set; }

    #region dodatkowe pole notmapped

    [NotMapped]      // using System.ComponentModel.DataAnnotations.Schema;
    [Display(Name = "Pan/Pani:")]
    public string PełneNazwisko
    {
        get { return Imie + " " + Nazwisko; }
    }

    #endregion

    public virtual ICollection<Ogloszenie> Ogloszenia
        ↪{ get; private set; }

    public async Task<ClaimsIdentity>
        ↪GenerateUserIdentityAsync(UserManager<Uzytkownik> manager)
    {
        // Note the authenticationType must match the one defined in
        // CookieAuthenticationOptions.AuthenticationType
    }
}
```

```

var userIdentity = await manager.CreateIdentityAsync(this,
    DefaultAuthenticationTypes.ApplicationCookie);
// Add custom user claims here
return userIdentity;
}
}

```

W kodzie zostały użyte regiony (#region NazwaRegionu oraz #endregion). Regiony pozwalają na oddzielenie części kodu i zwijanie go do jednej linii. Przykład zwiniętego regionu o nazwie dodatkowe pole notmapped prezentuje rysunek 8.16.

Rysunek 8.16.
Przykład zwianego regionu

```

public class Uzytkownik : IdentityUser
{
    2 references
    public Uzytkownik()
    {
        this.Ogloszenia = new HashSet<Ogloszenie>();
    }

    //klucz podstawowy odziedziczony z klasy IdentityUser

    //Dodajemy pola Imie i Nazwisko:
    1 reference
    public string Imie { get; set; }
    1 reference
    public string Nazwisko { get; set; }

    dodatkowe pole notmapped

    2 references
    public virtual ICollection<Ogloszenie> Ogloszenia { get; private set; }
}

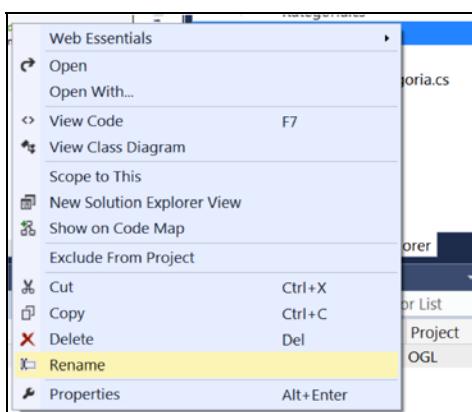
```

Tym sposobem utworzyliśmy kompletny model dla aplikacji. Kolejnym krokiem będzie utworzenie bądź zaktualizowanie klasy kontekstu.

Etap 1. Krok 3. Tworzenie klasy kontekstu

Ponieważ w domyślnym projekcie została już utworzona klasa kontekstu, dostosowano ją do konkretnych potrzeb bez tworzenia nowego kontekstu. Na początek zmieniamy nazwę pliku z *IdentityModels* na *OglContext* (rysunek 8.17).

Rysunek 8.17.
Zmiana nazwy kontekstu



Po zmianie nazwy pliku otwórz plik kontekstu (teraz już o nazwie *OglContext*) i zmień nazwę klasy z *ApplicationDbContext* na *OglContext* (razem z konstruktorem i sta-tyczną metodą).

Dodatkowo zamień:

IdentityDbContext<Uzytkownik>

na:

IdentityDbContext

i usuń:

, *throwIfV1Schema: false*

Kod klasy przed zmianami wygląda następująco:

```
public class ApplicationDbContext : IdentityDbContext<Uzytkownik>
{
    public ApplicationDbContext()
        : base("DefaultConnection", throwIfV1Schema: false)
    {
    }
    public static ApplicationDbContext Create()
    {
        return new ApplicationDbContext();
    }
}
```

A oto kod klasy po dokonaniu zmian:

```
public class OglContext : IdentityDbContext
{
    public OglContext()
        : base("DefaultConnection")
    {
    }
    public static OglContext Create()
    {
        return new OglContext();
    }
}
```

Teraz w całej aplikacji trzeba zamielić *ApplicationDbContext* na *OglContext*. Naciśnij klawisze *Ctrl+H* i zamień. Sposób zamiany opisano w przypadku klasy *Uzytkownik*.

Standardowo klasa kontekstu dziedziczy po klasie *DbContext*, ale w przygotowywanej aplikacji wykorzystamy to, że MVC posiada zaimplementowaną całą logikę odpowiedzialną za ciasteczka, tworzenie kont użytkowników, sprawdzanie haseł, bezpieczeństwo, wsparcie dla claimów, logowanie przez Facebook, Google itp. Dlatego też można wykorzystać gotowy *IdentityDbContext*. Klasa *Uzytkownik* z tego samego powodu dziedziczyła po klasie *IdentityUser*.

Jeśli nie chcesz korzystać z gotowej funkcjonalności lub nie musisz uwierzytelnić użytkowników, możesz utworzyć klasę dziedziczącą po *DbContext*:

```
public class OglContext : DbContext
{}
```

Teraz, gdy mamy już klasę z kontekstem, można dodać do niej właściwości DbSet reprezentujące w pamięci kontekstu tabele z bazy danych. Dodaj kod:

```
public DbSet<Kategoria> Kategorie { get; set; }
public DbSet<Ogłoszenie> Ogłoszenia { get; set; }
public DbSet<Uzytkownik> Uzytkownik { get; set; }
public DbSet<Ogłoszenie_Kategoria> Ogłoszenie_Kategoria { get; set; }
```

Po dodaniu kodu klasa kontekstu wygląda następująco:

```
public class OglContext : DbContext
{
    public OglContext()
        : base("DefaultConnection")
    {
    }
    public static OglContext Create()
    {
        return new OglContext();
    }

    public DbSet<Kategoria> Kategorie { get; set; }
    public DbSet<Ogłoszenie> Ogłoszenia { get; set; }
    public DbSet<Uzytkownik> Uzytkownik { get; set; }
    public DbSet<Ogłoszenie_Kategoria> Ogłoszenie_Kategoria
        { get; set; }
}
```

Następnie zainportuj biblioteki. Teraz kontekst jest już gotowy. Jednak podczas generowania bazy nazwy tabel w bazie danych zostaną zmienione z użyciem domyślnej konwencji tworzącej z nazwy liczbę mnogą. Aby aplikacja nie zmieniała nazw, trzeba nadpisać konwencję. Jeśli chcesz to zrobić dla kontekstu, skorzystaj z metody:

```
modelBuilder.Conventions.Remove<PluralizingTableNameConvention>();
```

Dopisz do klasy kontekstu następujący kod (można pominąć komentarze):

```
protected override void OnModelCreating(DbModelBuilder modelBuilder)
{
    // Potrzebne dla klas Identity
    base.OnModelCreating(modelBuilder);

    // using System.Data.Entity.ModelConfiguration.Conventions;
    // Wyłącza konwencję, która automatycznie tworzy liczbę mnogą dla nazw tabel
    // w bazie danych
    // Zamiast Kategorie została utworzona tabela o nazwie Kategories
    modelBuilder.Conventions.Remove
        <><PluralizingTableNameConvention>();

    // Wyłącza konwencję CascadeDelete
    // CascadeDelete zostanie włączone za pomocą Fluent API
    modelBuilder.Conventions.Remove
        <><OneToManyCascadeDeleteConvention>();

    // Używa się Fluent API, aby ustalić powiązanie pomiędzy tabelami
    // i włączyć CascadeDelete dla tego powiązania
```

```
modelBuilder.Entity<Ogloszenie>().IsRequired(x =>
    ↳ x.Uzytkownik).WithMany(x => x.Ogloszenia)
    ↳ .HasForeignKey(x => x.UzytkownikId)
    ↳ .WillCascadeOnDelete(true);
}
```

Ostatecznie klasa z kontekstem wygląda jak na listingu 8.1.

Listing 8.1. Klasa z kontekstem

```
public class OglContext : IdentityDbContext
{
    public OglContext()
        : base("DefaultConnection")
    {
    }
    public static OglContext Create()
    {
        return new OglContext();
    }

    public DbSet<Kategoria> Kategorie { get; set; }
    public DbSet<Ogloszenie> Ogloszenia { get; set; }
    public DbSet<Uzytkownik> Uzytkownik { get; set; }
    public DbSet<Ogloszenie_Kategoria> Ogloszenie_Kategoria
        ↳ { get; set; }

protected override void OnModelCreating(DbModelBuilder modelBuilder)
{
    // Potrzebne dla klas Identity
    base.OnModelCreating(modelBuilder);

    // using System.Data.Entity.ModelConfiguration.Conventions;
    // Wyłącza konwencję, która automatycznie tworzy liczbę mnogą dla nazw tabel
    // w bazie danych
    // Zamiast Kategorie zostałaby stworzona tabela o nazwie Kategories
    modelBuilder.Conventions.Remove
        ↳ <PluralizingTableNameConvention>();

    // Wyłącza konwencję CascadeDelete
    // CascadeDelete zostanie włączone za pomocą Fluent API
    modelBuilder.Conventions.Remove
        ↳ <OneToManyCascadeDeleteConvention>();

    // Używa się Fluent API, aby ustalić powiązanie pomiędzy tabelami
    // i włączyć CascadeDelete dla tego powiązania
    modelBuilder.Entity<Ogloszenie>().IsRequired
        ↳ (x => x.Uzytkownik).WithMany(x => x.Ogloszenia)
        ↳ .HasForeignKey(x => x.UzytkownikId)
        ↳ .WillCascadeOnDelete(true);
}
```

Poza wyłączeniem konwencji tworzącej liczbę mnogą wyłączono również globalnie dla całej aplikacji CascadeDelete za pomocą:

```
modelBuilder.Conventions.Remove<OneToManyCascadeDeleteConvention>();
```

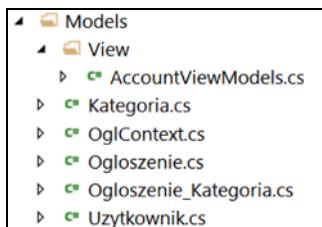
Następnie dla relacji Ogloszenie – Uzytkownik włączono CascadeTypeDelete za pomocą *Fluent API*:

```
modelBuilder.Entity<Ogloszenie>().HasRequired(x => x.Uzytkownik)
    .WithMany(x => x.Ogloszenia)
    .HasForeignKey(x => x.UzytkownikId)
    .WillCascadeOnDelete(true);
```

Mamy zatem przygotowane model i klasę kontekstu dla Entity Framework. W folderze *Models* znajdują się jeszcze pliki: *AccountViewModels* i *ManageViewModels*. Są to klasy z *ViewModel* na potrzeby logowania przez zewnętrzne serwisy i do zarządzania kontem użytkownika. Należy utworzyć folder *Views* w folderze *Models* i przenieść tam obydwa pliki. W ten sposób zachowamy porządek w strukturze aplikacji, dzięki czemu modele nie będą się mieszać z *ViewModel*. Strukturę folderu *Models* zaprezentowano na rysunku 8.18.

Rysunek 8.18.

Folder Models



Finalnie aplikację buduje się i przekompilowuje (*BUILD/Rebuild Solution*) oraz uruchamia klawiszem *F5*. Aplikacja powinna się uruchomić bez problemów (rysunek 8.19). W tym momencie, chociaż aplikacja została uruchomiona, nie ma jeszcze utworzonej bazy danych. Nie klikamy zakładek, aby aplikacja nie wygenerowała bazy danych. Baza danych zostaje utworzona dopiero w momencie pierwszego żądania (np. otwarcia podstrony korzystającej z bazy danych). Aplikacja nie posiada żadnych kontrolerów ani widoków korzystających z modelu. Zanim zostaną utworzone kontrolery i widoki, przykładowa aplikacja zostanie zmodyfikowana tak, aby posiadała bardziej modułową budowę i była lepiej zaprojektowana od strony architektonicznej.

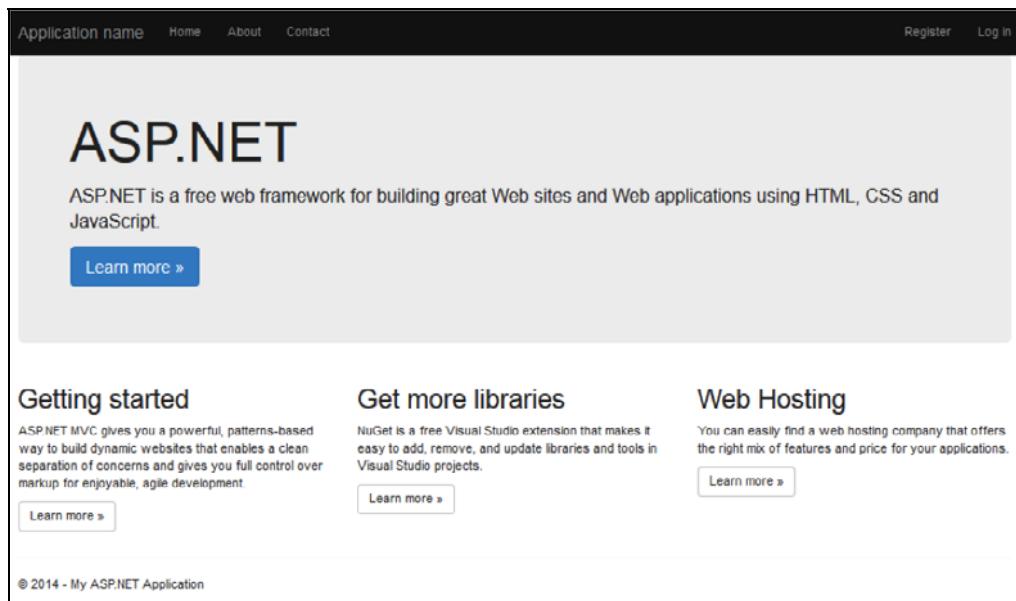
Przerwanie działania aplikacji zapewnia kombinacja klawiszy *Shift+F5*².

Etap 1. Krok 4. Przenoszenie warstwy modelu do osobnego projektu

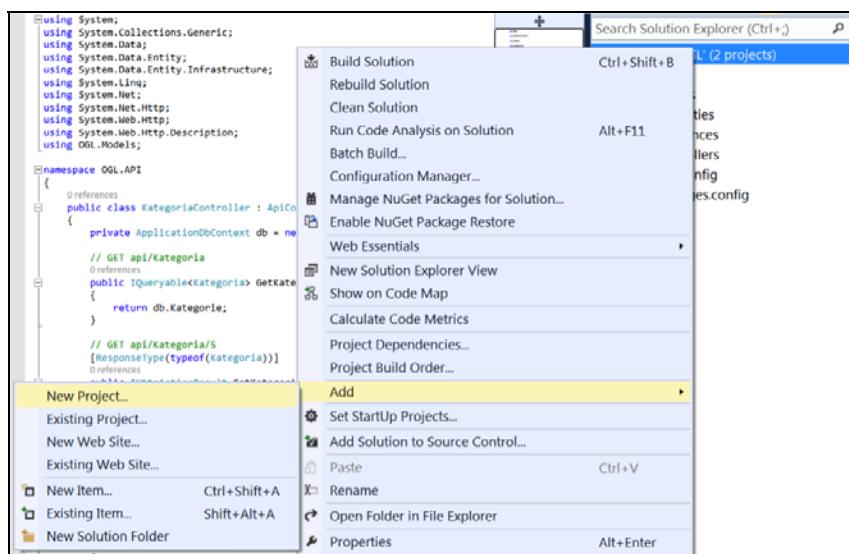
Ponieważ warstwa modelu może być wspólna dla kilku aplikacji i nie powinna być powiązana z widokami, kontrolerami i całą resztą aplikacji MVC, przeniesiemy ją do nowego projektu. Solucja³ aktualnie zawiera dwa projekty: *OGL* i *OGL.Tests*. Aby dodać nowy projekt, kliknij nazwę solucji prawym przyciskiem myszy i wybierz *Add/New Project...* (rysunek 8.20).

² Podczas działania aplikacji można wprowadzać zmiany jedynie w plikach z widokiem.

³ Solucja zawiera wiele projektów, każdy projekt to osobny plik *.dll*.



Rysunek 8.19. Uruchomiona aplikacja



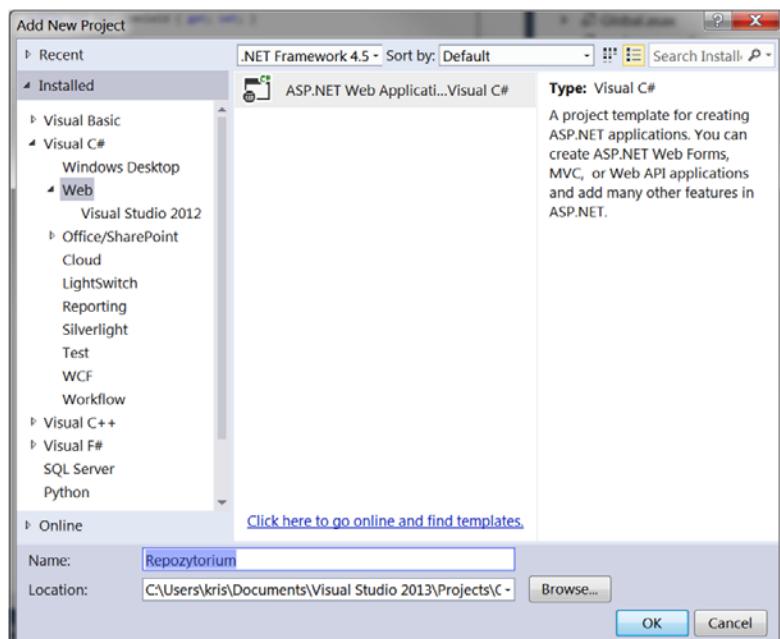
Rysunek 8.20. Dodawanie nowego projektu

Wybierz *ASP.NET Web Application* i nazwij⁴ projekt np. Repozytorium (rysunek 8.21).

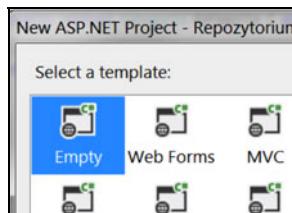
Wybierz szablon *Empty* (rysunek 8.22).

⁴ Nazwy *Model* dla projektu nie powinno się używać, ponieważ może powodować konflikty nazw w aplikacji.

Rysunek 8.21.
Tworzenie nowego projektu
Repozytorium

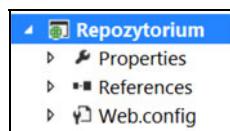


Rysunek 8.22.
Wybieranie szablonu



Zostanie utworzony pusty projekt o podanej nazwie (rysunek 8.23).

Rysunek 8.23.
Projekt Repozytorium



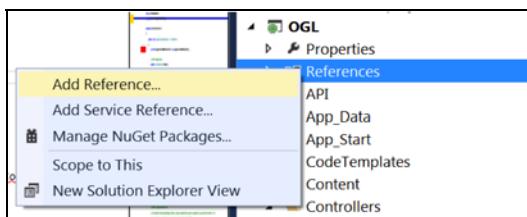
Dodawanie referencji pomiędzy projektami

Kolejnym etapem jest dodanie referencji. Referencje określają powiązania pomiędzy projektami, a więc wybiera się projekt, który korzysta z innego projektu⁵. Główny projekt, czyli OGL, będzie korzystał z Repozytorium w celu pobierania danych. Aby utworzyć referencję do Repozytorium z projektu OGL, kliknij prawym przyciskiem myszy *References* w projekcie OGL (rysunek 8.24) i wybierz *Add Reference*.

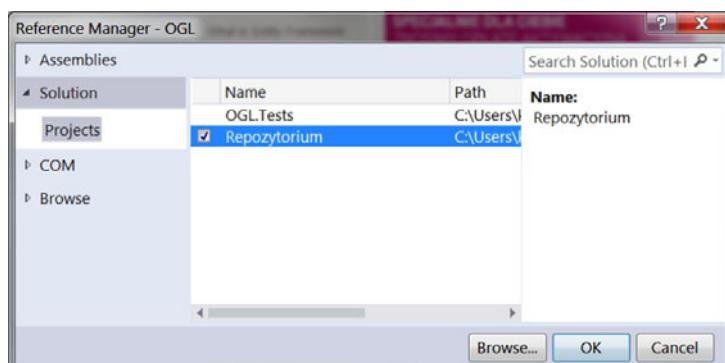
W kolejnym oknie zaznacz Repozytorium i kliknij OK (rysunek 8.25).

⁵ Nie można tworzyć referencji cyklicznych (z pierwszego projektu do drugiego i z drugiego do pierwszego).

Rysunek 8.24.
Dodawanie referencji do projektu



Rysunek 8.25.
Utworzenie referencji do projektu

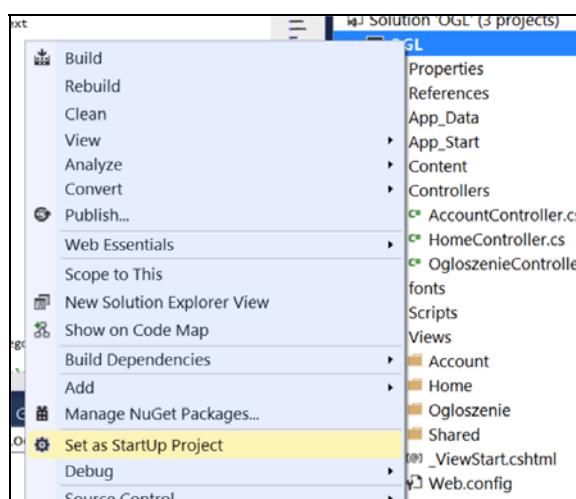


Od teraz w projekcie OGL można korzystać z klas projektu Repozytorium. Jednocześnie projekt Repozytorium nic nie wie o projekcie OGL i nie widzi jego klas.

Ustawienie projektu startowego

Ponieważ dodaliśmy nowy projekt, konieczne jest ustawienie, który projekt jest projektem głównym (startowym). W tym wypadku głównym projektem jest projekt OGL utworzony na samym początku. Aby ustawić projekt OGL jako startowy, kliknij prawym przyciskiem myszy projekt OGL i wybierz z menu kontekstowego opcję *Set as StartUp Project* (rysunek 8.26). Jeśli nie ustawi się projektu startowego, podczas uruchamiania aplikacji i w trakcie migracji mogą występować przeróżne błędy.

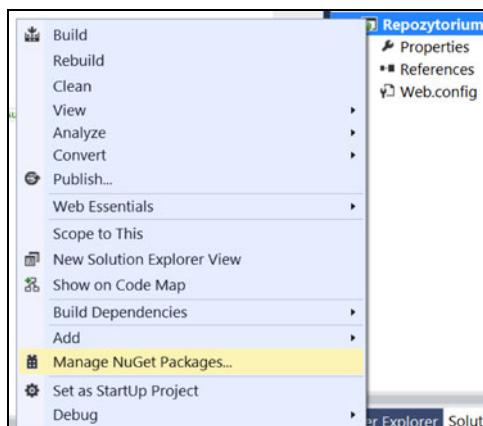
Rysunek 8.26.
Ustawianie projektu startowego



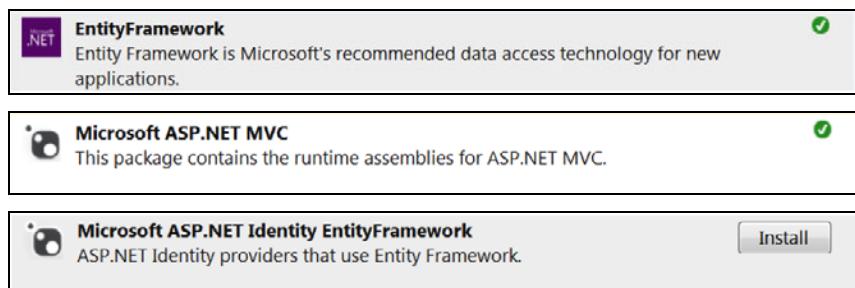
Instalacja bibliotek dla nowego projektu

Kolejnym krokiem jest instalacja niezbędnych bibliotek dla nowego projektu (Repozytorium) za pomocą NuGet (rysunek 8.27).

Rysunek 8.27.
Instalowanie bibliotek
dla projektu
Repozytorium



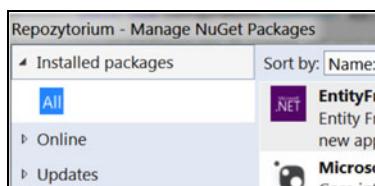
Przejdź na zakładkę *Online* (po lewej stronie) i zainstaluj pakiety widoczne na rysunku 8.28. Należy zadbać aby były to te same wersje bibliotek co w projekcie OGL.



Rysunek 8.28. Instalowanie pakietów

Po zainstalowaniu trzech pakietów przejdź na zakładkę *Installed packages/All*, aby sprawdzić, jakie pakiety zostały zainstalowane (rysunek 8.29).

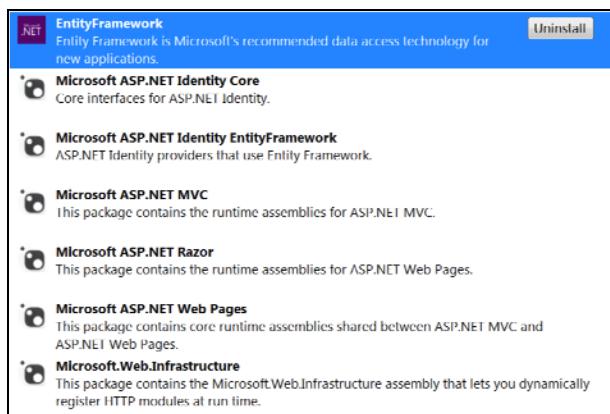
Rysunek 8.29.
Sprawdzanie
zainstalowanych
pakietów



Powinny zostać zainstalowane pakiety pokazane na rysunku 8.30⁶.

⁶ Niektóre pakiety wymagają innych do prawidłowego działania, dlatego zainstalowanych zostało więcej pakietów, niż podano.

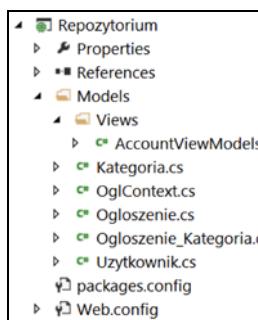
Rysunek 8.30.
Zainstalowane pakiety



Przenoszenie plików z modelem do osobnej warstwy (projektu)

Teraz można przenieść folder *Models* z projektu OGL do Repozytorium. Przenosimy wszystkie pliki poza plikiem o nazwie *ManageViewModels.cs*, ponieważ posiada on referencje do *Microsoft.Owin.Security*, a nie chcemy instalować pakietu OWIN w projekcie Repozytorium. Po przeniesieniu struktura projektu Repozytorium wygląda jak na rysunku 8.31.

Rysunek 8.31.
*Struktura projektu
Repozytorium
po dodaniu folderu
Models*



Konieczna jest jeszcze zmiana nazw w aplikacji z:

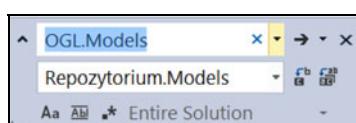
OGL.Models

na:

Repozytorium.Models

Aby to zrobić, użyj kombinacji klawiszy *Ctrl+H* i zamień nazwy dla całej solucji, a nie tylko dla jednego projektu (rysunek 8.32).

Rysunek 8.32.
*Zmiana nazw
w solucji*



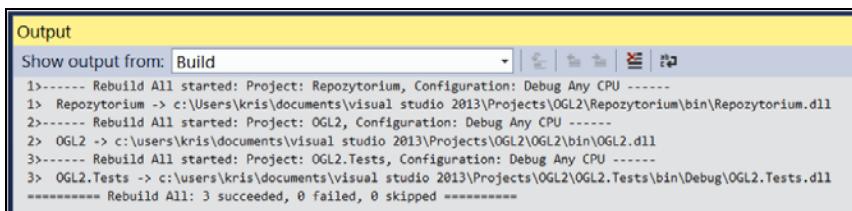
Podeczas zmiany nazw dla klas z modelem zostały również zamienione dyrektywy `using` odnoszące się do klas z modelem i kontekstem w kontrolerach oraz widokach z projektu OGL. Aktualnie istnieje tylko kontroler `AccountController`, w którym dyrektywa:

```
using Ogl.Models;
```

została zamieniona na:

```
using Repozytorium.Models;
```

Przebudowanie solucji odbywa się za pomocą opcji *Rebuild Solution* wybranej z zakładki — na tym etapie nie powinno być żadnych błędów podczas budowania projektu (rysunek 8.33). Jeśli jednak wystąpią problemy, należy zaktualizować pakiety w projekcie z testami `OGL.Tests`.



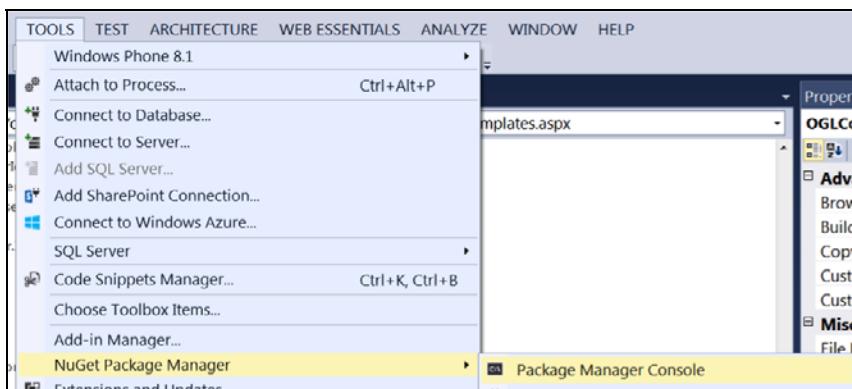
Rysunek 8.33. Przebudowanie solucji

Etap 1. Krok 5. Migracje

Podczas tworzenia aplikacji bardzo często aktualizuje się strukturę bazy danych. Do automatycznych aktualizacji struktury bazy danych wykorzystywany jest mechanizm migracji.

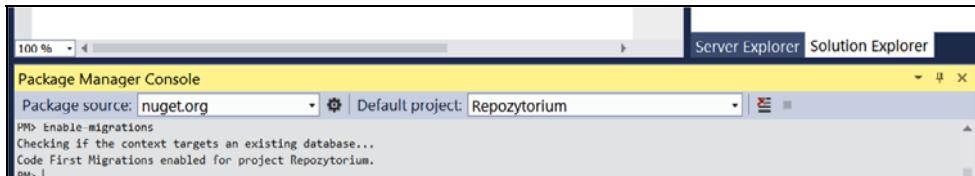
Instalacja migracji

Aby uruchomić migracje, otwórz okno *Package Manager Console* i wybierz *TOOLS/NuGet Package Manager/Package Manager Console* (rysunek 8.34).



Rysunek 8.34. Uruchamianie okna *Package Manager Console*

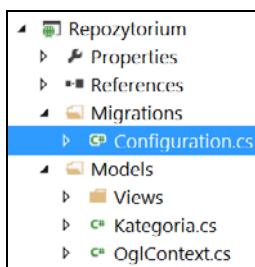
W nowo otwartym oknie (rysunek 8.35) wybierz projekt, dla którego zostaną włączone migracje (lista rozwijana *Default project*). W tym przypadku jest to projekt Repozytorium, ponieważ w Repozytorium znajduje się cała warstwa modelu. Wpisz komendę `Enable-Migrations` i kliknij klawisz *Enter*. Migracje zostaną zainstalowane. W przypadku więcej niż jednej klasy kontekstu w aplikacji należy podać jeszcze nazwę kontekstu, dla którego włączane są migracje, oraz folder, w jakim mają się one znajdować.



Rysunek 8.35. Włączenie migracji dla projektu Repozytorium

Po zainstalowaniu migracji w projekcie Repozytorium zostanie utworzony folder *Migrations*, a w nim plik konfiguracyjny *Configuration* (rysunek 8.36).

Rysunek 8.36.
Plik konfiguracyjny
migracji



Konfiguracja migracji

Poniżej przedstawiono zawartość pliku konfiguracyjnego migracji:

```
internal sealed class Configuration :  
    DbMigrationsConfiguration<Repozytorium.Models.OglContext>  
{  
    public Configuration()  
    {  
        AutomaticMigrationsEnabled = false;  
    }  
  
    protected override void Seed  
    (Repozytorium.Models.OglContext context)  
    {  
        // This method will be called after migrating to the latest version.  
  
        // You can use the DbSet<T>.AddOrUpdate() helper extension method  
        // to avoid creating duplicate seed data. E.g.  
        // context.People.AddOrUpdate(  
        //     p => p.FullName,  
        //     new Person { FullName = "Andrew Peters" },  
        //     new Person { FullName = "Brice Lambson" },  
        //     new Person { FullName = "Rowan Miller" }  
        // );
```

```
//  
}  
}
```

Migracje dzielą się na dwa typy: stratne i bezstratne. Migracje stratne występują, gdy nie ma możliwości zmiany struktury bazy danych bez wykasowania zapisanych danych. Bezstratne to takie, które nie wymagają usunięcia danych (np. dodanie dodatkowego pola do tabeli). W pliku konfiguracyjnym znajduje się metoda Seed, która jest uruchamiana podczas aktualizacji bazy danych po wykonaniu migracji. Metoda Seed wprowadza dane startowe do bazy danych, aby nie było potrzeby dodawania ich ręcznie po każdej aktualizacji struktury bazy.

Aby włączyć automatyczne migracje oraz włączyć migracje stratne, dodaj następujący kod w konstruktorze klasy konfiguracyjnej:

```
public Configuration()  
{  
    AutomaticMigrationsEnabled = true;  
    AutomaticMigrationDataLossAllowed = true;  
}
```

Od teraz bez pytania po zmianach w klasach z modelem będą wykonywane również migracje stratne, po czym zostanie wykonana metoda Seed.

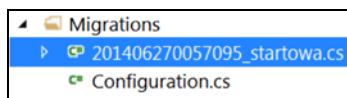
Tworzenie migracji początkowej

Aby utworzyć migrację startową, która wygeneruje kod odpowiedzialny za utworzenie bazy danych, w oknie *Package Manager Console* wpisz komendę:

```
Add-migration startowa
```

Zostanie utworzony plik zawierający w nazwie datę i nazwę podaną w komendzie (rysunek 8.37).

Rysunek 8.37.
Migracja startowa



Utworzona została klasa o nazwie startowa dziedzicząca po klasie DbMigration i zawierająca dwie metody: Up() oraz Down().

W metodzie Up() znajduje się kod zawierający zmiany dokonane od wcześniejszego wywołania migracji. Ponieważ baza danych nie została jeszcze utworzona, metoda Up() zawiera kod tworzący bazę danych (listing 8.2).

Listing 8.2. Kod metody Up()

```
public override void Up()  
{  
    CreateTable(  
        "dbo.Kategoria",  
        c => new  
    {
```

```
Id = c.Int(nullable: false, identity: true),
Nazwa = c.String(nullable: false),
ParentId = c.Int(nullable: false),
MetaTytul = c.String(maxLength: 72),
MetaOpis = c.String(maxLength: 160),
MetaSlowa = c.String(maxLength: 160),
Tresc = c.String(maxLength: 500),
})
.PrimaryKey(t => t.Id);

CreateTable(
    "dbo.Ogloszenie_Kategoria",
    c => new
    {
        Id = c.Int(nullable: false, identity: true),
        KategoriaId = c.Int(nullable: false),
        OgloszenieId = c.Int(nullable: false),
    })
.PrimaryKey(t => t.Id)
.ForeignKey("dbo.Kategoria", t => t.KategoriaId)
.ForeignKey("dbo.Ogloszenie", t => t.OgloszenieId)
.Index(t => t.KategoriaId)
.Index(t => t.OgloszenieId);

CreateTable(
    "dbo.Ogloszenie",
    c => new
    {
        Id = c.Int(nullable: false, identity: true),
        Tresc = c.String(maxLength: 500),
        Tytul = c.String(maxLength: 72),
        DataDodania = c.DateTime(nullable: false),
        UzytkownikId = c.String(nullable: false,
            maxLength: 128),
    })
.PrimaryKey(t => t.Id)
.ForeignKey("dbo.AspNetUsers", t => t.UzytkownikId,
    cascadeDelete: true)
.Index(t => t.UzytkownikId);

CreateTable(
    "dbo.AspNetUsers",
    c => new
    {
        Id = c.String(nullable: false, maxLength: 128),
        Email = c.String(maxLength: 256),
        EmailConfirmed = c.Boolean(nullable: false),
        PasswordHash = c.String(),
        SecurityStamp = c.String(),
        PhoneNumber = c.String(),
        PhoneNumberConfirmed = c.Boolean(nullable:
            false),
        TwoFactorEnabled = c.Boolean(nullable: false),
        LockoutEndDateUtc = c.DateTime(),
        LockoutEnabled = c.Boolean(nullable: false),
        AccessFailedCount = c.Int(nullable: false),
    })
.PrimaryKey(t => t.Id)
.ForeignKey("dbo.Ogloszenie", t => t.OgloszenieId)
.ForeignKey("dbo.Kategoria", t => t.KategoriaId)
.Index(t => t.OgloszenieId)
.Index(t => t.KategoriaId);
```

```

        UserName = c.String(nullable: false, maxLength:
    ↳256),
        Imie = c.String(),
        Nazwisko = c.String(),
        Wiek = c.Int(),
        Discriminator = c.String(nullable: false,
    ↳maxLength: 128),
    })
.PrimaryKey(t => t.Id)
.Index(t => t.UserName, unique: true, name:
    ↳"UserNameIndex");

// Dalsza część kodu została usunięta
}

```

Metoda `Down()` zawiera kod, który pozwala na odwrócenie zmian dokonanych przez metodę `Up()`. W tym przypadku usuwa wszystkie tabele (listing 8.3).

Listing 8.3. Kod metody `Down()`

```

public override void Down()
{
    DropForeignKey("dbo.AspNetUserRoles", "UserId",
    ↳"dbo.AspNetUsers");
    DropForeignKey("dbo.AspNetUserLogins", "UserId",
    ↳"dbo.AspNetUsers");
    DropForeignKey("dbo.AspNetUserClaims", "UserId",
    ↳"dbo.AspNetUsers");
    DropForeignKey("dbo.AspNetUserRoles", "RoleId",
    ↳"dbo.AspNetRoles");
    DropForeignKey("dbo.Ogloszenie", "UzytkownikId",
    ↳"dbo.AspNetUsers");
    DropForeignKey("dbo.Ogloszenie_Kategoria", "OgloszenieId",
    ↳"dbo.Ogloszenie");
    DropForeignKey("dbo.Ogloszenie_Kategoria", "KategoriaId",
    ↳"dbo.Kategoria");
    DropIndex("dbo.AspNetRoles", "RoleNameIndex");
    DropIndex("dbo.AspNetUserRoles", new[] { "RoleId" });
    DropIndex("dbo.AspNetUserRoles", new[] { "UserId" });
    DropIndex("dbo.AspNetUserLogins", new[] { "UserId" });
    DropIndex("dbo.AspNetUserClaims", new[] { "UserId" });
    DropIndex("dbo.AspNetUsers", "UserNameIndex");
    DropIndex("dbo.Ogloszenie", new[] { "UzytkownikId" });
    DropIndex("dbo.Ogloszenie_Kategoria", new[] { "OgloszenieId" });
    DropIndex("dbo.Ogloszenie_Kategoria", new[] { "KategoriaId" });
    DropTable("dbo.AspNetRoles");
    DropTable("dbo.AspNetUserRoles");
    DropTable("dbo.AspNetUserLogins");
    DropTable("dbo.AspNetUserClaims");
    DropTable("dbo.AspNetUsers");
    DropTable("dbo.Ogloszenie");
    DropTable("dbo.Ogloszenie_Kategoria");
    DropTable("dbo.Kategoria");
}

```

Uruchomienie pierwszej migracji

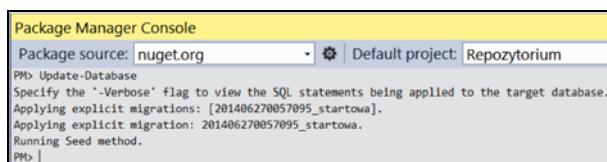
Aby uruchomić utworzoną migrację, należy wpisać komendę Update-Database. Zostanie uruchomiona migracja startowa (rysunek 8.38). Po wykonanej migracji zostanie wywołana metoda Seed(), która na razie nie ma zawartości.

Jeśli migracja się nie powiedzie i otrzymamy następujący błąd:

```
Cannot attach the file 'C:\Users\nazwa-uzytownika\Documents\Visual Studio 2013\Projects\OGL\OGL\App_Data\aspnet-OGL-20140807102453.mdf' as database
→ 'aspnet-OGL-20140807102453'.
```

Rysunek 8.38.

Uruchomienie migracji startowej



to należy zmienić nazwę pliku *.mdf* z bazą danych. Błąd występuje, gdy kilkakrotnie tworzona jest baza o tej samej nazwie. Aby zmienić nazwę, przechodzimy do pliku *Web.config* w projekcie OGL i zmieniamy w dwóch miejscach nazwy atrybutów w sekcji *connectionStrings*: Data Source i Initial Catalog. W naszym przypadku zamieniliśmy końcówkę z 453 na 454.

Po zmianie kod wygląda następująco:

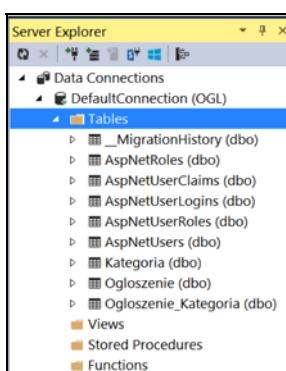
```
connectionString="Data Source=(LocalDb)\v11.0;AttachDbFilename=|DataDirectory|\aspnet-OGL-20140807102454.mdf;Initial Catalog=aspnet-OGL-20140807102454;
→Integrated Security=True"
```

Po zmianie nazwy tworzenie bazy danych odbywa się prawidłowo.

W oknie *Server Explorer* (aby włączyć okno, wybierz *VIEW/Server Explorer*) w Visual Studio można teraz zobaczyć, że została utworzona baza danych (rysunek 8.39).

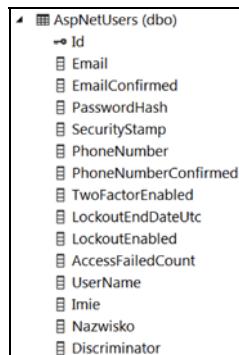
Rysunek 8.39.

Struktura bazy danych



Klasa *Uzytkownik* w bazie danych nazywa się *AspNetUsers*, ponieważ dziedziczy po klasie *IdentityUser*, i zawiera wiele dodatkowych odziedziczonych pól (rysunek 8.40). Nie zawiera pola *PelneNazwisko*, które zostało oznaczone atrybutem *[NotMapped]*.

Rysunek 8.40.
Struktura tabel
AspNetUsers
(*Uzytkownik*)



Metoda Seed()

Kolejnym krokiem jest implementacja metody `Seed()` znajdującej się w pliku konfiguracyjnym migracji, aby po każdej stratnej migracji nie było potrzeby dodawania nowych danych (listing 8.4).

Listing 8.4. Metoda `Seed()`

```

protected override void Seed(Repozytorium.Models.OglContext context)
{
    //Do debugowania metody seed
    //if(System.Diagnostics.Debugger.IsAttached == false)
    //System.Diagnostics.Debugger.Launch();
    SeedRoles(context);
    SeedUsers(context);
    SeedOgłoszenia(context);
    SeedKategorie(context);
    SeedOgłoszenie_Kategoria(context);
}
private void SeedRoles(OglContext context)
{
    var roleManager = new
    ↳RoleManager<Microsoft.AspNet.Identity.EntityFramework.IdentityRole>
    ↲(new RoleStore<IdentityRole>());
    if (!roleManager.RoleExists("Admin"))
    {
        var role = new
        ↳ Microsoft.AspNet.Identity.EntityFramework.IdentityRole();
        role.Name = "Admin";
        roleManager.Create(role);
    }
}
private void SeedUsers(OglContext context)
{
    var store = new UserStore<Uzytkownik>(context);
    var manager = new UserManager<Uzytkownik>(store);
    if (!context.Users.Any(u => u.UserName == "Admin"))
    {
        var user = new Uzytkownik { UserName = "Admin" };
    }
}
  
```

```
var adminresult = manager.Create(user, "12345678");

if (adminresult.Succeeded)
    manager.AddToRole(user.Id, "Admin");
}

}

private void SeedOgloszenia(OglContext context)
{
    var idUzytkownika = context.Set<Uzytkownik>()
        .Where(u=>u.UserName == "Admin")
        .FirstOrDefault().Id;
    for (int i = 1; i <= 10; i++)
    {
        var ogl = new Ogloszenie()
        {
            Id = i,
            UzytkownikId = idUzytkownika,
            Tresc = "Treść ogłoszenia" + i.ToString(),
            Tytul = "Tytuł ogłoszenia" + i.ToString(),
            DataDodania = DateTime.Now.AddDays(-i)
        };
        context.Set<Ogloszenie>().AddOrUpdate(ogl);
    }
    context.SaveChanges();
}

private void SeedKategorie(OglContext context)
{
    for (int i = 1; i <= 10; i++)
    {
        var kat = new Kategoria()
        {
            Id = i,
            Nazwa = "Nazwa kategorii" + i.ToString(),
            Tresc = "Treść ogłoszenia" + i.ToString(),
            MetaTytul = "Tytuł kategorii" + i.ToString(),
            MetaOpis = "Opis kategorii" + i.ToString(),
            MetaSlowa = "Słowa kluczowe do kategorii" + i.ToString(),
            ParentId = i
        };
        context.Set<Kategoria>().AddOrUpdate(kat);
    }
    context.SaveChanges();
}

private void SeedOgloszenie_Kategoria(OglContext context)
{
    for (int i = 1; i < 10; i++)
    {
        var okat = new Ogloszenie_Kategoria()
        {
            Id = i,
            OgloszenieId = i / 2 + 1,
            KategoriaId = i / 2 + 2
        };
        context.Set<Ogloszenie_Kategoria>().AddOrUpdate(okat);
    }
}
```

```

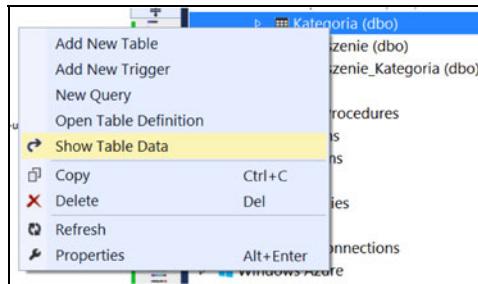
        }
        context.SaveChanges();
    }
}

```

Wykorzystaliśmy metodę AddOrUpdate, która nie będzie duplikować danych przy każdym wywołaniu metody Seed(). Po zaimplementowaniu metody Seed() zimportuj biblioteki, zapisz plik i uruchom migrację komendą Update-Database.

Aby sprawdzić, czy dane zostały faktycznie dodane, w oknie *Server Explorer* kliknij prawym przyciskiem myszy tabelę Kategoria i wybierz z menu kontekstowego opcję *Show Table Data* (rysunek 8.41).

Rysunek 8.41.
Sprawdzanie danych w tabeli



Zostanie otwarte okno (rysunek 8.42), w którym widać dane dodane przez metodę Seed(), a w tym przypadku metodę SeedKategorie() wywołaną w metodzie Seed(). Podobnie można sprawdzić pozostałe tabele.

dbo.Kategoria [Data]		201406270057095_startowa.cs					
KluczPo...	Nazwa	Par...	MetaTytul	MetaOpis	SlowaKluczowe	Tresc	
1	Nazwa Kategorii1	1	Tytuł kategorii1	Opis kategorii1	Slowa kluczow...	Treść ogłoszenia1	
2	Nazwa Kategorii2	2	Tytuł kategorii2	Opis kategorii2	Slowa kluczow...	Treść ogłoszenia2	
3	Nazwa Kategorii3	3	Tytuł kategorii3	Opis kategorii3	Slowa kluczow...	Treść ogłoszenia3	
4	Nazwa Kategorii4	4	Tytuł kategorii4	Opis kategorii4	Slowa kluczow...	Treść ogłoszenia4	
5	Nazwa Kategorii5	5	Tytuł kategorii5	Opis kategorii5	Slowa kluczow...	Treść ogłoszenia5	
6	Nazwa Kategorii6	6	Tytuł kategorii6	Opis kategorii6	Slowa kluczow...	Treść ogłoszenia6	
7	Nazwa Kategorii7	7	Tytuł kategorii7	Opis kategorii7	Slowa kluczow...	Treść ogłoszenia7	
8	Nazwa Kategorii8	8	Tytuł kategorii8	Opis kategorii8	Slowa kluczow...	Treść ogłoszenia8	
9	Nazwa Kategorii9	9	Tytuł kategorii9	Opis kategorii9	Slowa kluczow...	Treść ogłoszenia9	
10	Nazwa Kategorii10	10	Tytuł kategorii10	Opis kategorii10	Slowa kluczow...	Treść ogłoszenia10	
*	NULL	NULL	NULL	NULL	NULL	NULL	

Rysunek 8.42. *Dane w tabeli*

Debugowanie metody Seed

Ponieważ nie ma możliwości debugowania metody Seed() podczas uruchamiania aplikacji lub migracji, konieczne jest utworzenie nowej instancji Visual Studio na potrzeby debugowania. Aby to zrobić, na początku metody Seed() należy dodać (odkomentować) następujący kod:

```

if (System.Diagnostics.Debugger.IsAttached == false)
    System.Diagnostics.Debugger.Launch();

```

Zastosowanie tego kodu podczas startu metody Seed() uruchomi drugą instancję Visual Studio, w której będzie możliwość debugowania i sprawdzenia, jakie błędy zawiera metoda.

Zmiany w modelu i kolejna migracja

Dodaj pole Wiek do klasy Uzytkownik:

```
public int Wiek { get; set; }
```

Zapisz plik i uruchom komendę:

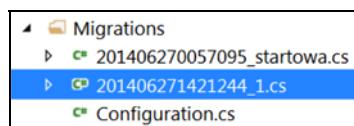
```
Add-migration 1
```

gdzie 1 to nazwa migracji.

Zostanie utworzona nowa migracja (rysunek 8.43).

Rysunek 8.43.

Kolejna migracja



Oto kod nowej migracji:

```
public partial class _1 : DbMigration
{
    public override void Up()
    {
        AddColumn("dbo.AspNetUsers", "Wiek", c => c.Int());
    }

    public override void Down()
    {
        DropColumn("dbo.AspNetUsers", "Wiek");
    }
}
```

Jak widać, w metodzie Up() zostaje dodana kolumna Wiek do tabeli AspNetUsers typu int. Metoda Down() usuwa kolumnę Wiek z tabeli.

Aby wprowadzić zmiany w bazie danych, uruchom komendę Update-Database.

Otrzymano błąd podczas wykonywania metody Seed():

```
The 'Wiek' property on 'Uzytkownik' could not be set to a 'null' value. You must
set this property to a non-null value of type 'System.Int32'.
```

Pole Wiek nie może być puste, ponieważ jest to wartość typu int. Aby było możliwe przypisanie wartości null, pole Wiek należy oznaczyć jako Nullable. Dodaj pytajnik w klasie z modelem i zapisz plik:

```
public int? Wiek { get; set; }
```

Od teraz pole `Wiek` może być nieustawione, a więc mieć wartość `null`. Ponowne uruchomienie komendy `Update-Database` nie powinno skutkować błędami podczas wykonywania metody `Seed()`.

Po odświeżeniu i sprawdzeniu struktury tabeli w bazie danych w oknie *Server Explorer* widać nowo dodaną kolumnę (rysunek 8.44).

Rysunek 8.44.

Nowa kolumna
w tabeli

Imię
Nazwisko
Discriminator
Wiek

W tym momencie można przerobić metodę `SeedUsers()` znajdująca się w pliku konfiguracyjnym migracji, aby przypisywała wiek użytkownikowi. Zmiany dokonane w tej metodzie będą widoczne dopiero po usunięciu i ponownym utworzeniu bazy danych. Metoda `Seed` podczas aktualizacji bazy danych pominie ten fragment kodu, ponieważ użytkownik `Admin` już istnieje. Wiersz:

```
var user = new Uzytkownik { UserName = "Admin" };
```

należy zastąpić następującym:

```
var user = new Uzytkownik { UserName = "Admin", Wiek = 12 };
```

Praca z błędami i niespójnością w migracjach

Zapewne każdemu zdarzy się pozmieniać coś w strukturze bazy danych lub namieszać tak, że trudno będzie się samemu połapać. Wykonane migracje są zapisywane w bazie danych w tabeli `_MigrationHistory` (rysunek 8.45).

Rysunek 8.45.

Tabela
`_MigrationHistory`

Tables
MigrationHistory (dbo)
MigrationId
ContextKey
Model
ProductVersion
AspNetRoles (dbo)

Jeśli dodasz np. pole `Wiek`, wykonasz migrację i zaktualizujesz bazę danych, a następnie usuniesz to pole z bazy danych, to mechanizm migracji „zgłupieje” i zwróci błąd, że brakuje kolumny `Wiek` w tabeli. Można dodać tę kolumnę do bazy danych lub usunąć migrację, która ją dodawała. Aby usunąć migrację, trzeba wykasować ostatni plik z folderu `Migrations`⁷ oraz wykasować odpowiedni wiersz z tabeli `_MigrationHistory` o nazwie takiej jak migracja. Można wykasować więcej migracji, ważne, aby zachować spójność struktury bazy danych z ostatnią migracją. Jeśli nie można lub nie opłaca się naprawiać struktury, można usunąć wszystkie tabele z bazy danych i wykasować wszystkie migracje, zostawiając tylko plik konfiguracyjny z metodą `Seed`. Następnie dodaj i uruchom migrację początkową.

⁷ Kasuje się w kolejności od ostatnich migracji.

Etap 1. Podsumowanie (warstwa modelu i migracje)

Na tym etapie istnieje już baza danych. Można ją aktualizować, ale nadal brakuje aplikacji, która korzysta z tej bazy danych. W pewnym stopniu warstwa modelu/bazy danych jest odseparowana od warstwy widoku i kontrolerów. W większości przypadków baza danych jest gotowa, trzeba tylko dodać aplikację, która potrafi na niej działać. Klasy odwzorowują strukturę w bazie danych i w najlepszym podejściu powinny zostać przeniesione do osobnego projektu, do którego referencje posiadałby praktycznie każdy projekt z solucji (prawie każdy projekt korzysta z klas z modelem). Projekt z modelem nie powinien mieć referencji do żadnych bibliotek, takich jak np. EF, ponieważ ma być niezależny od konkretnego sposobu dostępu do danych⁸. Jeśli model jest niezależny, to można korzystać z kilku bibliotek lub zamieniać te odpowiedzialne za warstwę dostępu do danych. W tym przypadku nie ma możliwości ani sensu przenoszenia klas z modelem do osobnego projektu, ponieważ korzystamy z klas Identity i gotowej funkcjonalności do logowania, która wymaga EF (obsługa zaimplementowana jest przy użyciu EF). Klasa Uzytkownik dziedziczy po IdentityUser, która wymaga EF. Jeśli przeniesie się wszystkie klasy poza klasą Uzytkownik do osobnego projektu z modelem, to nie będzie możliwości utworzenia powiązania pomiędzy klasami Uzytkownik a Ogloszenie (nie można dodawać cyklicznych referencji). Istnieją dwa wyjścia: można albo pozostawić klasy z modelem w repozytorium, albo utworzyć projekt z modelem i programowo (bez relacji w bazie danych, dwie osobne bazy danych) utworzyć powiązanie „jeden do jednego” pomiędzy tabelą AspNetUsers w bazie danych a dodatkową tabelą Uzytkownik w drugiej bazie danych. W drugim podejściu występują dwie bazy danych i dwa osobne konteksty. Podczas zakładania konta niezbędne byłoby utworzenie użytkownika o tym samym Id w dwóch tabelach (Uzytkownik i AspNetUsers). Kontekst dla klas Identity działałby tylko i wyłącznie na tabelach od ASP.NET Identity odpowiedzialnych np. za logowanie, ciasteczka, a drugi kontekst aplikacji działałby na tabeli Uzytkownik i powiązanych z nią tabelach, np. Ogloszenia, Kategorie.

W aplikacji dla uproszczenia skorzystano z pierwszego podejścia, w którym klasy z modelem są w projekcie repozytorium, ponieważ będą korzystać tylko z EF.

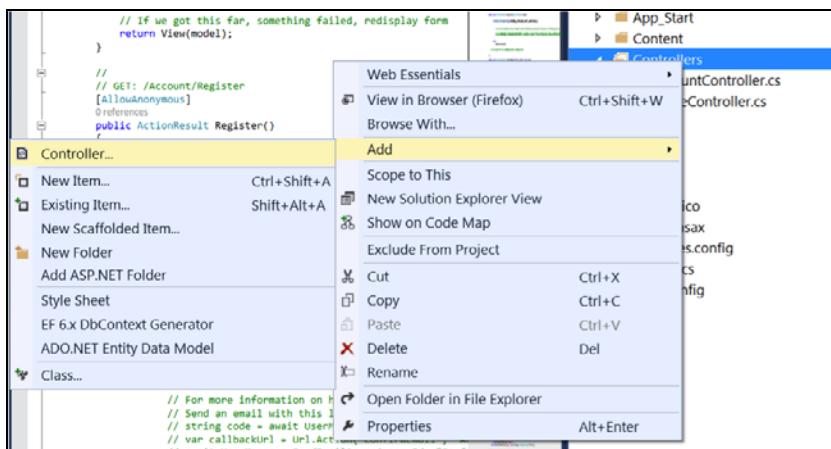
Etap 2. Krok 1. Dodawanie kontrolerów i widoków — akcja Index

Ponieważ baza danych jest gotowa, możemy przejść do tworzenia części aplikacji odpowiedzialnej za kontakt z użytkownikiem, czyli kontrolerów i widoków. Widok jest odpowiedzialny za wyświetlanie danych z modelu, natomiast kontroler za obsługę zdarzeń/akcji użytkownika.

Dodawanie kontrolera z widokami

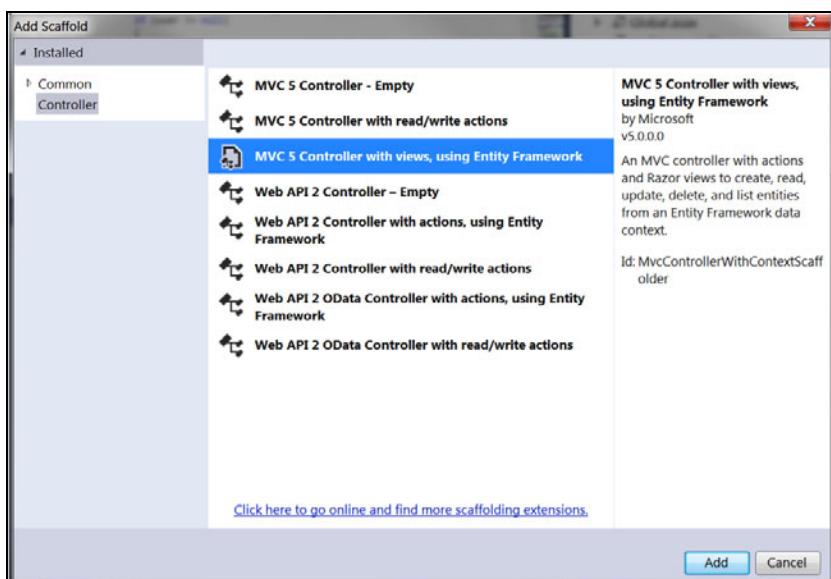
Aby dodać kontroler, kliknij prawym przyciskiem myszy folder *Controllers* i wybierz *Add/Controller*, jak na rysunku 8.46.

⁸ EF służy do dostępu do danych, podobnie jak NHibernate.



Rysunek 8.46. Dodawanie kontrolera

Następnie wybierz opcję *MVC 5 Controller with views, using Entity Framework* i kliknij *Add*, jak na rysunku 8.47.

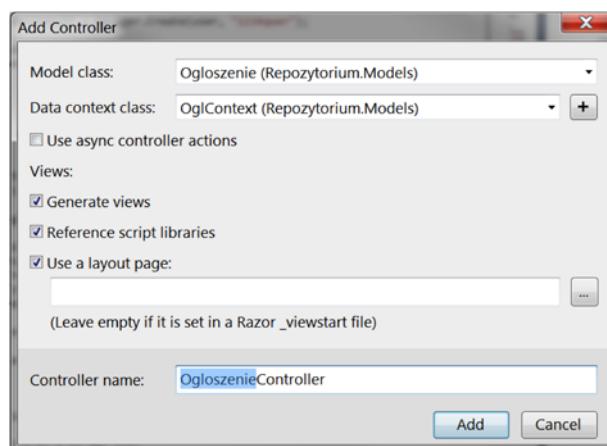


Rysunek 8.47. Wybieranie opcji dla kontrolera

W kolejnym kroku wybierz klasę z modelem, na podstawie której tworzony jest kontroler (rysunek 8.48). Rozpocznij od klasy *Ogłoszenie*. Wybierz klasę kontekstu, która ma zostać użyta do operacji na bazie danych. W tym programie będzie to *OglContext*. Zaznacz opcję *Generate views*, która tworzy folder o tej samej nazwie co kontroler w folderze *Views*, a w nim pliki z widokami dla operacji CRUD. Jako nazwę wprowadź *OgłoszenieController*. Nie zaznaczaj opcji *async*, ponieważ w tym przypadku nie potrzeba metod asynchronicznych. Pole *Use a layout page* pozostaw puste, tak by został wybrany domyślny layout ustwiony w pliku *_ViewStart.cshtml*.

Rysunek 8.48.

Wybór klas z modelem



W utworzonym kontrolerze znajduje się:

- ◆ nowy obiekt kontekstu:

```
private OglContext db = new OglContext();
```

- ◆ metoda Dispose():

```
protected override void Dispose(bool disposing)
{
    if (disposing)
    {
        db.Dispose();
    }
    base.Dispose(disposing);
}
```

- ◆ metody kontrolera dla operacji CRUD.

Całą klasę prezentuje listing 8.5.

Listing 8.5. Kod klasy

```
public class OgloszenieController : Controller
{
    private OglContext db = new OglContext();

    // GET: /Ogłoszenie/
    public ActionResult Index()
    {
        var ogłoszenia = db.Ogłoszenia.Include(o => o.Uzytkownik);
        return View(ogłoszenia.ToList());
    }

    // GET: /Ogłoszenie/Details/5
    public ActionResult Details(int? id)
    {
        if (id == null)
        {
```

```
        return new
            ↪HttpStatusCodeResult(HttpStatusCode.BadRequest);
    }
    Ogloszenie ogloszenie = db.Ogloszenia.Find(id);
    if (ogloszenie == null)
    {
        return HttpNotFound();
    }
    return View(ogloszenie);
}

// GET: /Ogloszenie/Create
public ActionResult Create()
{
    ViewBag.UzytkownikId = new SelectList(db.Users, "Id",
        ↪"Email");
    return View();
}

// POST: /Ogloszenie/Create
[HttpPost]
[ValidateAntiForgeryToken]
public ActionResult Create([Bind(Include="Id,Tresc,Tytul,
    ↪DataDodania, UzytkownikId")] Ogloszenie ogloszenie)
{
    if (ModelState.IsValid)
    {
        db.Ogloszenia.Add(ogloszenie);
        db.SaveChanges();
        return RedirectToAction("Index");
    }
    ViewBag.UzytkownikId = new SelectList(db.Users, "Id",
        ↪"Email", ogloszenie.UzytkownikId);
    return View(ogloszenie);
}

// GET: /Ogloszenie/Edit/5
public ActionResult Edit(int? id)
{
    if (id == null)
    {
        return new
            ↪HttpStatusCodeResult(HttpStatusCode.BadRequest);
    }
    Ogloszenie ogloszenie = db.Ogloszenia.Find(id);
    if (ogloszenie == null)
    {
        return HttpNotFound();
    }
    ViewBag.UzytkownikId = new SelectList(db.Users, "Id",
        ↪"Email", ogloszenie.UzytkownikId);
    return View(ogloszenie);
}

// POST: /Ogloszenie/Edit/5
[HttpPost]
[ValidateAntiForgeryToken]
```

```
public ActionResult Edit([Bind(Include= "Id,Tresc,Tytul,DataDodania, UzytkownikId")]
    ↳Ogłoszenie ogłoszenie)
{
    if (ModelState.IsValid)
    {
        db.Entry(ogłoszenie).State = EntityState.Modified;
        db.SaveChanges();
        return RedirectToAction("Index");
    }
    ViewBag.UzytkownikId = new SelectList(db.Users, "Id",
    ↳"Email", ogłoszenie.UzytkownikId);
    return View(ogłoszenie);
}

// GET: /Ogłoszenie/Delete/5
public ActionResult Delete(int? id)
{
    if (id == null)
    {
        return new HttpStatusCodeResult(HttpStatusCode.BadRequest);
    }
    Ogłoszenie ogłoszenie = db.Ogłoszenia.Find(id);
    if (ogłoszenie == null)
    {
        return HttpNotFound();
    }
    return View(ogłoszenie);
}

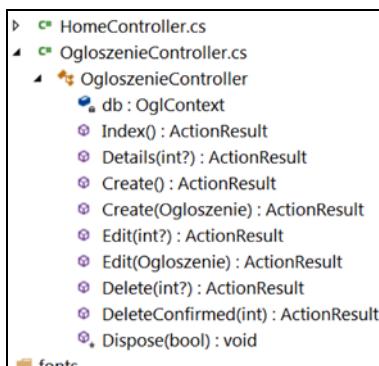
// POST: /Ogłoszenie/Delete/5
[HttpPost, ActionName("Delete")]
[ValidateAntiForgeryToken]
public ActionResult DeleteConfirmed(int id)
{
    Ogłoszenie ogłoszenie = db.Ogłoszenia.Find(id);
    db.Ogłoszenia.Remove(ogłoszenie);
    db.SaveChanges();
    return RedirectToAction("Index");
}

protected override void Dispose(bool disposing)
{
    if (disposing)
    {
        db.Dispose();
    }
    base.Dispose(disposing);
}
```

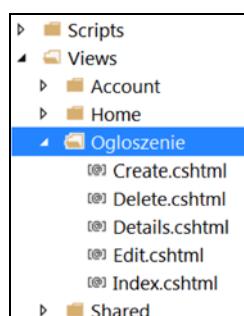
Strukturę klasy OgłoszenieController prezentuje rysunek 8.49.

Oprócz klasy kontrolera zostały utworzone również widoki dla poszczególnych akcji (rysunek 8.50).

Rysunek 8.49.
Struktura klasy



Rysunek 8.50.
Struktura folderu
z widokami



Pierwsze uruchomienie aplikacji i routing

Teraz można uruchomić aplikację. Kliknij *Rebuild Solution*, a następnie klawisz *F5*, aby to zrobić. Żeby przejść do akcji z kontrolera, wpisz adres (rysunek 8.51). Numer portu po słowie localhost będzie inny dla każdego uruchomienia, dlatego należy dopisać tylko /Ogloszenie/.

Rysunek 8.51.
Adres uruchomionej
aplikacji



Routing domyślnie przekieruje do akcji Index z kontrolera OgloszenieController według domyślnej metody routingu. Domyślna metoda wygląda następująco:

```

routes.MapRoute(
    name: "Default",
    url: "{controller}/{action}/{id}",
    defaults: new { controller = "Home", action = "Index", id =
        UrlParameter.Optional }
);

```

Jeśli nie podasz w adresie ani kontrolera, ani nazwy akcji, to domyślnie zostanie wyświetlona akcja Index z kontrolera Home. A zatem dla strony głównej <http://AspNetMvc.pl/> zostanie wyświetlona strona z kontrolera Home i akcja/widok Index. Przykładowo dla strony głównej podane adresy pokażą tę samą stronę:

- ◆ <http://AspNetMvc.pl/>,
- ◆ <http://AspNetMvc.pl/Home/>,
- ◆ <http://AspNetMvc.pl/Home/Index/>.

Jeśli podasz nazwę kontrolera, ale nie wskażesz nazwy akcji, zostanie wywołana akcja Index z podanego kontrolera. Na przykład <http://AspNetMvc.pl/Ogloszenie/> wyświetli akcję Index z kontrolera Ogloszenie. Przykładowo dla ogłoszeń podane adresy pokażą tę samą stronę:

- ◆ <http://AspNetMvc.pl/Ogloszenie/>,
- ◆ <http://AspNetMvc.pl/Ogloszenie/Index/>.

Po zamianie domyślnej akcji na Create w regule routing adres http://AspNetMvc.pl/Ogloszenie/_ pokaże na to samo miejsce co adres⁹ <http://AspNetMvc.pl/Ogloszenie/Create/>.

Lista ogłoszeń (akcja Index) — aktualizacja widoku/wyglądu strony

Kolejnym elementem są ogłoszenia. Ponieważ metoda Seed dodała dane początkowe, powinna pokazać się lista ogłoszeń (akcja Index to inaczej lista) (rysunek 8.52).

Index				
Create				
Email	Treść ogłoszenia:	Tytuł ogłoszenia:	Data dodania:	
Treść ogłoszenia1	Tytuł ogłoszenia1	2014-06-26	Edit Details Delete	
Treść ogłoszenia2	Tytuł ogłoszenia2	2014-06-25	Edit Details Delete	
Treść ogłoszenia3	Tytuł ogłoszenia3	2014-06-24	Edit Details Delete	
Treść ogłoszenia4	Tytuł ogłoszenia4	2014-06-23	Edit Details Delete	
Treść ogłoszenia5	Tytuł ogłoszenia5	2014-06-22	Edit Details Delete	
Treść ogłoszenia6	Tytuł ogłoszenia6	2014-06-21	Edit Details Delete	
Treść ogłoszenia7	Tytuł ogłoszenia7	2014-06-20	Edit Details Delete	
Treść ogłoszenia8	Tytuł ogłoszenia8	2014-06-19	Edit Details Delete	
Treść ogłoszenia9	Tytuł ogłoszenia9	2014-06-18	Edit Details Delete	
Treść ogłoszenia10	Tytuł ogłoszenia10	2014-06-17	Edit Details Delete	

Rysunek 8.52. Akcja Index

Ponieważ ogłoszenia posiadają relację z tabelą Uzytkownik, w pierwszej kolumnie została automatycznie wybrany Email użytkownika. Uzytkownik utworzony przez metodę Seed nie posiada e-maila, dlatego kolumna ta jest pusta. W zamian można wyświetlić nazwę użytkownika, czyli UserName.

Poniżej zaprezentowano kod widoku wygenerowanego automatycznie (folder Views/Ogloszenie/, plik Index.cshtml):

⁹ Wystąpiłyby wtedy jednak problem ze stroną główną, ponieważ kontroler Home nie posiada akcji Create.

```

@model IEnumerable<Repozytorium.Models.Ogłoszenie>

{@{
    ViewBag.Title = "Index";
}

<h2>Index</h2>

<p>
    @Html.ActionLink("Create", "Create")
</p>
<table class="table">
    <tr>
        <th>
            @Html.DisplayNameFor(model => model.Uzytkownik.Email)
        </th>
        <th>
            @Html.DisplayNameFor(model => model.Tresc)
        </th>
        <th>
            @Html.DisplayNameFor(model => model.Tytul)
        </th>
        <th>
            @Html.DisplayNameFor(model => model.DataDodania)
        </th>
        <th></th>
    </tr>
    @foreach (var item in Model)
    {
        <tr>
            <td>
                @Html.DisplayFor(modelItem => item.Uzytkownik.Email)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.Tresc)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.Tytul)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.DataDodania)
            </td>
            <td>
                @Html.ActionLink("Edit", "Edit", new { id=item.Id }) |
                @Html.ActionLink("Details", "Details",
                    new { id=item.Id }) |
                @Html.ActionLink("Delete", "Delete", new { id=item.Id })
            </td>
        </tr>
    }
</table>

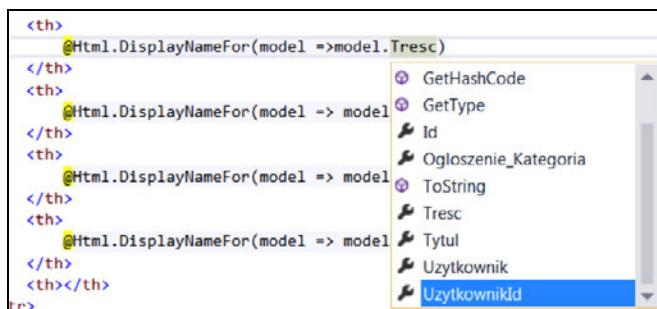
```

Wygenerowany widok jest typowany:

```
@model IEnumerable<Repozytorium.Models.Ogłoszenie>
```

dzięki czemu Visual Studio może podpowiadać składnię (rysunek 8.53).

Rysunek 8.53.
Podpowiedź składni



Oto zmiany, jakie wprowadzono do widoku (listing 8.6, rysunek 8.54):

♦ Tytuł Index

<h2>Index</h2>

został zamieniony na Lista ogłoszeń:

<h2>Lista ogłoszeń</h2>

♦ W pierwszej kolumnie zamiast Email wyświetlana jest nazwa użytkownika.

♦ Zamienione zostały tytuł kolumny w tabeli oraz znacznik <th>, użyto także HTML helpera o nazwie DisplayNameFor, który wyświetla nazwę kolumny. Jeśli w klasie z modelem dodany jest atrybut, np. [DisplayName = "Email użytkownika:"], to wyświetlana jest jego wartość, czyli kolumna nazywałaby się Email użytkownika:, a jeśli nie ma atrybutu (tak jak w tym przypadku), wyświetlana jest nazwa właściwości z klasy z modelem. A zatem słowo Email:

```
<th>
    @Html.DisplayNameFor(model => model.Uzytkownik.Email)
</th>
```

zamieniono na:

```
<th>
    @Html.DisplayNameFor(model => model.Uzytkownik.UserName)
</th>
```

♦ Zamienione zostały wartości w tabeli wyświetlane w pętli oraz znacznik <td>, użyto także HTML helpera o nazwie DisplayFor do wyświetlania wartości. A zatem słowo Email:

```
<td>
    @Html.DisplayFor(modelItem => item.Uzytkownik.Email)
</td>
```

zamieniono na:

```
<td>
    @Html.DisplayFor(modelItem => item.Uzytkownik.UserName)
</td>
```

♦ Linki zostały przetłumaczone na język polski, np.:

```
@Html.ActionLink("Edit", "Edit", new { id=item.Id }) |
```

zamieniono na:

```
@Html.ActionLink("Edytuj", "Edit", new { id=item.Id }) |
```

Zmiany w widokach można przeprowadzać w trakcie działania aplikacji. Aby zobaczyć zmiany, należy odświeżyć stronę.

Listing 8.6. Kod widoku po aktualizacji

```
@model IEnumerable<Repozytorium.Models.Ogloszenie>

 @{
    ViewBag.Title = "Index";
}

<h2>Lista ogłoszeń</h2>

<p>
    @Html.ActionLink("Dodaj nowe ogłoszenie", "Create")
</p>


| @Html.DisplayNameFor(model => model.Uzytkownik.UserName)                                                          | @Html.DisplayNameFor(model => model.Tresc) | @Html.DisplayNameFor(model => model.Tytul) | @Html.DisplayNameFor(model => model.DataDodania) |
|-------------------------------------------------------------------------------------------------------------------|--------------------------------------------|--------------------------------------------|--------------------------------------------------|
| @Html.DisplayFor(modelItem => item.Uzytkownik.UserName)                                                           | @Html.DisplayFor(modelItem => item.Tresc)  | @Html.DisplayFor(modelItem => item.Tytul)  | @Html.DisplayFor(modelItem => item.DataDodania)  |
| <a href="#">Edytuj</a>           @Html.ActionLink("Szczegóły", "Details", new         {             id=item.Id }) | <a href="#">Usuń</a>                       |                                            |                                                  |


```

Lista Ogłoszeń				
Dodaj nowe ogłoszenie				
UserName	Treść ogłoszenia:	Tytuł ogłoszenia:	Data dodania:	
Admin	Treść ogłoszenia1	Tytuł ogłoszenia1	2014-06-26	Edytuj Szczegóły Usuń
Admin	Treść ogłoszenia2	Tytuł ogłoszenia2	2014-06-25	Edytuj Szczegóły Usuń
Admin	Treść ogłoszenia3	Tytuł ogłoszenia3	2014-06-24	Edytuj Szczegóły Usuń
Admin	Treść ogłoszenia4	Tytuł ogłoszenia4	2014-06-23	Edytuj Szczegóły Usuń
Admin	Treść ogłoszenia5	Tytuł ogłoszenia5	2014-06-22	Edytuj Szczegóły Usuń
Admin	Treść ogłoszenia6	Tytuł ogłoszenia6	2014-06-21	Edytuj Szczegóły Usuń
Admin	Treść ogłoszenia7	Tytuł ogłoszenia7	2014-06-20	Edytuj Szczegóły Usuń
Admin	Treść ogłoszenia8	Tytuł ogłoszenia8	2014-06-19	Edytuj Szczegóły Usuń
Admin	Treść ogłoszenia9	Tytuł ogłoszenia9	2014-06-18	Edytuj Szczegóły Usuń
Admin	Treść ogłoszenia10	Tytuł ogłoszenia10	2014-06-17	Edytuj Szczegóły Usuń

Rysunek 8.54. Wygląd strony po aktualizacji

```
</tr>
}
</table>
```

Lista ogłoszeń a pobieranie danych

W wygenerowanym kodzie dla operacji Index (listy) oprócz danych ogłoszeń ładowane są również dane użytkownika. Kod akcji Index z kontrolera OgloszenieController wygląda następująco:

```
public ActionResult Index()
{
    var ogloszenia = db.Ogloszenia.Include(o => o.Uzytkownik);
    return View(ogloszenia.ToList());
}
```

Z pomocą metody `Include()` dla każdego ogłoszenia ładowany jest użytkownik. Aby sprawdzić, jakie zapytanie zostało wysłane do bazy danych, skorzystano z nowości wprowadzonej w EF6, a więc logowania SQL. Przerwij działanie programu i na początku akcji Index dodaj linijkę:

```
db.Database.Log = message => Trace.WriteLine(message);
```

Zimportuj bibliotekę (`using System.Diagnostics;`) dla podkreślonego kodu.

Kod akcji po zmianach wygląda następująco:

```
public ActionResult Index()
{
    db.Database.Log = message => Trace.WriteLine(message);
    var ogloszenia = db.Ogloszenia.Include(o => o.Uzytkownik);
    return View(ogloszenia.ToList());
}
```

Następnie zbuduj (*BUILD/Rebuild Solution*) i uruchom aplikację (*F5*). W oknie *Output* po odwiedzeniu podstrony *Ogłoszenie* powinien się pojawić następujący kod, który został wysłany do bazy danych:

```

SELECT
    [Extent1].[Id] AS [Id],
    [Extent1].[Tresc] AS [Tresc],
    [Extent1].[Tytul] AS [Tytul],
    [Extent1].[DataDodania] AS [DataDodania],
    [Extent1].[UzytkownikId] AS [UzytkownikId],
    CASE WHEN ([Extent2].[AccessFailedCount] IS NULL) THEN
        ↪CAST(NULL AS varchar(1)) ELSE '2X0X' END AS [C1],
    [Extent2].[Id] AS [Id1],
    [Extent2].[Email] AS [Email],
    [Extent2].[EmailConfirmed] AS [EmailConfirmed],
    [Extent2].[PasswordHash] AS [PasswordHash],
    [Extent2].[SecurityStamp] AS [SecurityStamp],
    [Extent2].[PhoneNumber] AS [PhoneNumber],
    [Extent2].[PhoneNumberConfirmed] AS [PhoneNumberConfirmed],
    [Extent2].[TwoFactorEnabled] AS [TwoFactorEnabled],
    [Extent2].[LockoutEndDateUtc] AS [LockoutEndDateUtc],
    [Extent2].[LockoutEnabled] AS [LockoutEnabled],
    [Extent2].[AccessFailedCount] AS [AccessFailedCount],
    [Extent2].[UserName] AS [UserName],
    [Extent2].[Imie] AS [Imie],
    [Extent2].[Nazwisko] AS [Nazwisko],
    [Extent2].[Wiek] AS [Wiek]
    FROM [dbo].[Ogłoszenie] AS [Extent1]
    LEFT OUTER JOIN [dbo].[AspNetUsers] AS [Extent2] ON ([Extent2].[Discriminator] = N'Uzytkownik') AND ([Extent1].[UzytkownikId] = [Extent2].[Id])
-- Executing at 2014-06-27 20:23:54 +02:00
-- Completed in 71 ms with result: SqlDataReader

```

Jak widać, wygenerowany kod korzysta z operacji *join*, aby pobrać użytkowników dla ogłoszeń. Czas wykonania zapytania to 71 ms (podczas korzystania z bazy danych na hostingu; jeśli korzystasz z domyślnego połączenia, czyli bazy danych znajdującej się lokalnie na komputerze, czas wykonania zapytania będzie dużo krótszy).

Optymalizacja listy ogłoszeń

Teraz przyszła kolej na optymalizację zapytania. Zamiast wyświetlać nazwę użytkownika (*UserName*), będzie wyświetlane jego ID, które jest zapisane w tabeli *Ogłoszenie* jako klucz obcy. Dzięki temu nie będzie konieczności wykonywania operacji *join* i pobierania danych z tabeli *Uzytkownik*.

Na początek zmień plik widoku *Index* dla ogłoszeń. Poniższy kod:

```

<th>
    @Html.DisplayNameFor(model => model.Uzytkownik.UserName)
</th>

```

zamień na:

```
<th>
    @Html.DisplayNameFor(model => model.UzytkownikId)
</th>
```

Podmień także wiersz:

```
<td>
    @Html.DisplayFor(modelItem => item.Uzytkownik.UserName)
</td>
```

na:

```
<td>
    @Html.DisplayFor(modelItem => item.UzytkownikId)
</td>
```

W pliku OgloszenieController usuń z zapytania w akcji Index metodę `Include()`, która wyłącza *Lazy Loading* (ładuje dane użytkowników dla ogłoszeń), oraz dodaj metodę `AsNoTracking()`, która wyłączy śledzenie danych przez kontekst. Dane są przeznaczone tylko do odczytu, dlatego nie trzeba śledzić ich stanu ani przechowywać ich w pamięci operacyjnej. Wykorzystano *Lazy Loading*, aby nie pobierać danych użytkowników.

Oto ciało metody przed aktualizacją:

```
public ActionResult Index()
{
    db.Database.Log = message => Trace.WriteLine(message);
    var ogloszenia = db.Ogloszenia.Include(o => o.Uzytkownik);
    return View(ogloszenia.ToList());
}
```

A to ciało metody po aktualizacji:

```
public ActionResult Index()
{
    db.Database.Log = message => Trace.WriteLine(message);
    var ogloszenia = db.Ogloszenia.AsNoTracking();
    return View(ogloszenia.ToList());
}
```

Po ponownym uruchomieniu aplikacji można przejść na podstronę ogłoszeń, w której znajduje się kolumna `UzytkownikId` (rysunek 8.55). Jak widać, `UzytkownikId` nie jest zwykłą liczbą, a jest to wartość typu GUID (ang. *Globally Unique Identifier*)¹⁰.

W oknie *Output* pojawi się zapytanie wysłane do bazy danych. Tym razem pobrane zostają tylko ogłoszenia bez operacji `join`, co jest dużo szybsze:

¹⁰ GUID jest tworzony na podstawie czasu wygenerowania oraz liczb pseudolosowych, co daje bardzo duże prawdopodobieństwo, że nie powtórzy się dwa razy. GUID jest stosowany jako ID dla użytkownika przez ASP.NET Identity.

Lista Ogłoszeń				
UzytkownikaId	Treść ogłoszenia:	Tytuł ogłoszenia:	Data dodania:	
73184b1f-5f56-4b66-ab48-f5d337a36c17	Treść ogłoszenia1	Tytuł ogłoszenia1	2014-06-26	Edytuj Szczegóły Usuń
73184b1f-5f56-4b66-ab48-f5d337a36c17	Treść ogłoszenia2	Tytuł ogłoszenia2	2014-06-25	Edytuj Szczegóły Usuń
73184b1f-5f56-4b66-ab48-f5d337a36c17	Treść ogłoszenia3	Tytuł ogłoszenia3	2014-06-24	Edytuj Szczegóły Usuń
73184b1f-5f56-4b66-ab48-f5d337a36c17	Treść ogłoszenia4	Tytuł ogłoszenia4	2014-06-23	Edytuj Szczegóły Usuń
73184b1f-5f56-4b66-ab48-f5d337a36c17	Treść ogłoszenia5	Tytuł ogłoszenia5	2014-06-22	Edytuj Szczegóły Usuń
73184b1f-5f56-4b66-ab48-f5d337a36c17	Treść ogłoszenia6	Tytuł ogłoszenia6	2014-06-21	Edytuj Szczegóły Usuń
73184b1f-5f56-4b66-ab48-f5d337a36c17	Treść ogłoszenia7	Tytuł ogłoszenia7	2014-06-20	Edytuj Szczegóły Usuń
73184b1f-5f56-4b66-ab48-f5d337a36c17	Treść ogłoszenia8	Tytuł ogłoszenia8	2014-06-19	Edytuj Szczegóły Usuń
73184b1f-5f56-4b66-ab48-f5d337a36c17	Treść ogłoszenia9	Tytuł ogłoszenia9	2014-06-18	Edytuj Szczegóły Usuń
73184b1f-5f56-4b66-ab48-f5d337a36c17	Treść ogłoszenia10	Tytuł ogłoszenia10	2014-06-17	Edytuj Szczegóły Usuń

Rysunek 8.55. Podstrona ogłoszeń

```

SELECT
    [Extent1].[Id] AS [Id],
    [Extent1].[Tresc] AS [Tresc],
    [Extent1].[Tytuł] AS [Tytuł],
    [Extent1].[DataDodania] AS [DataDodania],
    [Extent1].[UzytkownikaId] AS [UzytkownikaId]
FROM [dbo].[Ogłoszenie] AS [Extent1]

-- Executing at 2014-06-27 21:24:06 +02:00

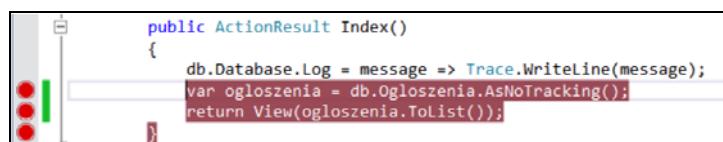
-- Completed in 45 ms with result: SqlDataReader

```

Operacja bez join trwała 45 ms. Operacja z join trwała 71 ms. Dla bardziej skomplikowanych danych różnica byłaby dużo większa.

Etap 2. Krok 2. Debugowanie oraz metody AsNoTracking() i ToList()

Dzięki debuggerowi można sprawdzić, co i w jakiej kolejności dzieje się w aplikacji: jakie wartości mają zmienne w określonych miejscach kodu oraz jak działają poszczególne metody, np. `ToList()` i `AsNoTracking()`. Aby ustawić punkt przerwania (ang. *Breakpoint*), w którym ma się zatrzymać aplikacja, kliknij po lewej stronie obok linii. Linijka zostanie zaznaczona na czerwono i pokaże się czerwona kropka po lewej stronie. Zaznacz trzy punkty przerwania w miejscach oznaczonych na rysunku 8.56.

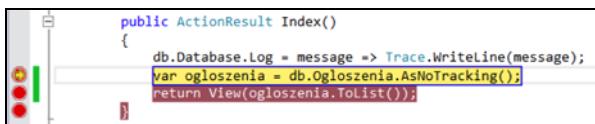
Rysunek 8.56.
Ustawianie punktów przerwania

Uruchom aplikację w trybie debugowania klawiszem *F5*. Aby wystartować bez debugowania (aplikacja nie będzie się zatrzymywać w zaznaczonych miejscach), należy użyć skrótu *Ctrl+F5*.

Po przejściu do podstrony *Ogłoszenia* strona się nie ładuje. W Visual Studio pierwsza linia oznaczona jako punkt przerwania zostaje zaznaczona na żółto (rysunek 8.57).

Rysunek 8.57.

Debugowanie aplikacji

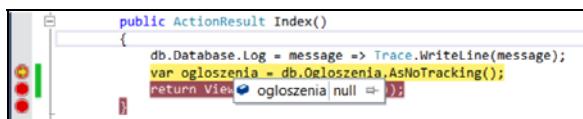


Sprawdzanie wartości zmiennych

W tym momencie aplikacja została zatrzymana i można przeglądać wartości zmiennych. Aby zobaczyć, jakie wartości mają zmienne, wystarczy zatrzymać się myszką nad jedną z nich. Pokażę się wtedy okno, a w nim wartość danej zmiennej w tym momencie. Po najechaniu na zmienną *ogłoszenia*, na której zatrzymał się program, widać, że w tym momencie ma ona wartość *null* (rysunek 8.58).

Rysunek 8.58.

Wartość zmiennej



Drugim sposobem sprawdzania wartości zmiennych jest otworzenie okna *Locals* w Visual Studio (rysunek 8.59).

Rysunek 8.59.

Okno Locals

Name	Value	Type
this	(OGL.Controllers.OgłoszenieController)	OGL.Controllers.OgłoszenieController
base	(OGL.Controllers.OgłoszenieController)	System.Web.Mvc.Controller (OGL.Controllers.OgłoszenieController)
db	(Repozytorium.Models.OglContext)	Repozytorium.Models.OglContext
ogłoszenia	null	System.Data.Entity.Infrastructure.DbQuery<Repozytorium.Models.Ogłoszenie>

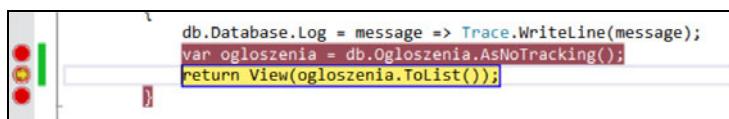
Poza tymi dwoma sposobami jest jeszcze okno *Watch*, które pozwala śledzić tylko te zmienne, które zostaną wybrane poprzez wpisanie ich nazwy.

Metoda *ToList()* i odroczone wykonanie (Deferred Execution)

Aby przejść do kolejnego punktu przerwania, kliknij klawisz *F5*. Aplikacja zatrzyma się w kolejnej linijce (rysunek 8.60).

Rysunek 8.60.

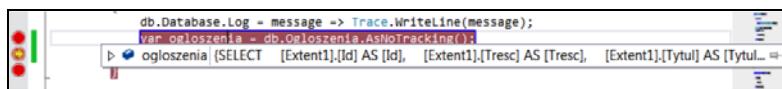
Poruszanie się w debuggerze



Sprawdziliśmy wartość zmiennej *ogłoszenia*. Jak widać, dane nie zostały jeszcze pobrane (nie ma wypisanego zapytania w oknie *Output*), chociaż linijka została wykonana. Zmienna pokazuje, że ma wartość zapytania. To zachowanie to odroczone (opóźnione)

wykonanie (ang. *Deferred Execution*). Zapytania zostają wykonane dopiero wtedy, gdy są potrzebne dane. W tym momencie nie ma jeszcze potrzeby, aby je pobierać (rysunek 8.61).

Rysunek 8.61.
Pobieranie danych



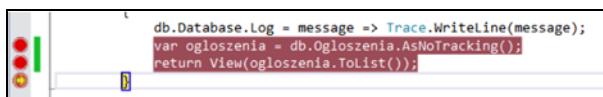
Aby śledzić zmienną `ogłoszenia`, należy ją wpisać w oknie *Watch*, jak zaprezentowano na rysunku 8.62.



Rysunek 8.62. Dopisywanie zmiennej w oknie *Watch*

Przechodzenie do kolejnego punktu przerwania umożliwia klawisz *F5*. Kolejnym elementem jest teraz nawias zamykający blok kodu metody `Index` (rysunek 8.63).

Rysunek 8.63.
Kolejny punkt
przerwania



W oknie *Output* można zaobserwować, że zapytanie zostało wykonane:

```
SELECT
[Extent1].[Id] AS [Id],
[Extent1].[Tresc] AS [Tresc],
[Extent1].[Tytuł] AS [Tytuł],
[Extent1].[DataDodania] AS [DataDodania],
[Extent1].[UzytkownikId] AS [UzytkownikId]
FROM [dbo].[Ogłoszenie] AS [Extent1]
```

```
-- Executing at 2014-06-27 23:00:11 +02:00
```

```
-- Completed in 45 ms with result: SqlDataReader
```

W oknie *Watch* widać, że dane zostały pobrane i są zapisane w zmiennej `ogłoszenie` (rysunek 8.64).

Zapytanie zostało wykonane, ponieważ użyliśmy metody `ToList()`, która wyłącza opóźnione wykonanie. Jeśli usunie się metodę `ToList()` i ponownie przejdzie do tego punktu przerwania, można zobaczyć, że dane nie zostały jeszcze pobrane z bazy danych. Dopiero po kolejnym kliknięciu klawisza *F5* i wyjściu ze wszystkich punktów przerwań zostanie wykonane zapytanie, a dane zostaną pobrane.

Metoda AsNoTracking()

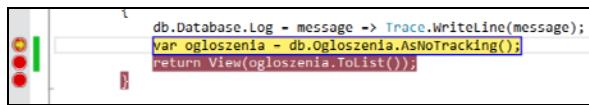
Następnie poddamy analizie działanie metody `AsNoTracking()` i mechanizm śledzenia obiektów przez obiekt kontekstu. Uruchom aplikację i przejdź do podstrony *Ogłoszenia*. Aplikacja zatrzyma się na pierwszym punkcie przerwania (rysunek 8.65).

Watch 1	
Name	Value
ogloszenia	(SELECT [Extent1].[Id] AS [Id], [Extent1].[Tresc] AS [Tresc], [Extent1].[Titul] AS [Titul], [Extent1].[DataDodania] AS [DataDodania] FROM [Ogloszenia] AS [Extent1])
Non-Public m	
Results View	Expanding the Results View will enumerate the IEnumerable
[0]	[System.Data.Entity.DynamicProxies.Ogloszenie_1BF42DDE731C37E485D302A26982A218A9D453DDA8970A40D767E7E4D5FCB22C]
[1]	[System.Data.Entity.DynamicProxies.Ogloszenie_1BF42DDE731C37E485D302A26982A218A9D453DDA8970A40D767E7E4D5FCB22C]
[2]	[System.Data.Entity.DynamicProxies.Ogloszenie_1BF42DDE731C37E485D302A26982A218A9D453DDA8970A40D767E7E4D5FCB22C]
[3]	[System.Data.Entity.DynamicProxies.Ogloszenie_1BF42DDE731C37E485D302A26982A218A9D453DDA8970A40D767E7E4D5FCB22C]
[4]	[System.Data.Entity.DynamicProxies.Ogloszenie_1BF42DDE731C37E485D302A26982A218A9D453DDA8970A40D767E7E4D5FCB22C]
[5]	[System.Data.Entity.DynamicProxies.Ogloszenie_1BF42DDE731C37E485D302A26982A218A9D453DDA8970A40D767E7E4D5FCB22C]
[6]	[System.Data.Entity.DynamicProxies.Ogloszenie_1BF42DDE731C37E485D302A26982A218A9D453DDA8970A40D767E7E4D5FCB22C]
[7]	[System.Data.Entity.DynamicProxies.Ogloszenie_1BF42DDE731C37E485D302A26982A218A9D453DDA8970A40D767E7E4D5FCB22C]
[8]	[System.Data.Entity.DynamicProxies.Ogloszenie_1BF42DDE731C37E485D302A26982A218A9D453DDA8970A40D767E7E4D5FCB22C]
[9]	[System.Data.Entity.DynamicProxies.Ogloszenie_1BF42DDE731C37E485D302A26982A218A9D453DDA8970A40D767E7E4D5FCB22C]

Rysunek 8.64. Informacje na temat zmiennej w oknie Watch

Rysunek 8.65.

Ponowne debugowanie aplikacji



W oknie *Watch* usunęliśmy wcześniej wpisany obiekt *Ogłoszenie* i wpisaliśmy *db*, a więc obiekt kontekstu. Po rozwinięciu zawartości i przejęciu do ogłoszeń (rysunek 8.66) widać, że obiekt jest aktualnie pusty (*Count* = 0).

Rysunek 8.66.

Debugowanie obiektu *db*

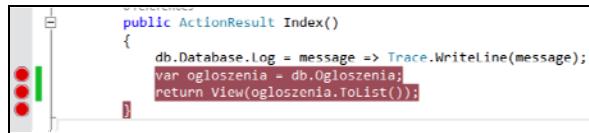
Watch 1	
Name	Value
db	{Repozytorium.Models.OgContext}
base	{Repozytorium.Models.OgContext}
Kategorie	{System.Data.Entity.DbSet`1[Repozytorium.Models.Kategoria]}
Ogłoszenia	[SELECT [Extent1].[Id] AS [Id], [Extent1].[Tresc] AS [Tresc], [Extent1].[Titul] AS [Titul], [Extent1].[DataDodania] AS [DataDodania] FROM [Ogloszenia] AS [Extent1]]
base	[SELECT [Extent1].[Id] AS [Id], [Extent1].[Tresc] AS [Tresc], [Extent1].[Titul] AS [Titul], [Extent1].[DataDodania] AS [DataDodania] FROM [Ogloszenia] AS [Extent1]]
Local	Count = 0
Non-Public	
Results View	Expanding the Results View will enumerate the IEnumerable

Po kliknięciu *F5* aplikacja zatrzyma się na kolejnym punkcie przerwania, jednak w oknie *Output* widać, że zapytanie dalej nie zostało wykonane. Jeśli klikniesz kolejny raz *F5*, aplikacja zatrzyma się na ostatnim punkcie przerwania. Wcześniej została wykonana metoda *ToList()*, czyli dane zostały pobrane, co można sprawdzić w oknie *Output*. Po przejęciu do okna *Watch* i sprawdzeniu, czy ogłoszenia zostały zapisane do kontekstu, możesz ponownie zobaczyć, że *Count* = 0, czyli brakuje danych. Jeśli klikniesz *F5* ostatni raz, przeglądarka wyświetli stronę z danymi. W obiekcie kontekstu nie zostały zapisane żadne dane przez cały czas działania programu. W ten sposób zademonstrowano zasadę działania metody *AsNoTracking()*, czyli wyłączającej zapis danych do obiektu kontekstu i ich śledzenie.

Następnie przerwij działanie programu, wciskając *Shift+F5*, i usuń wywołanie metody *AsNoTracking()* w akcji *Index*. Kod akcji *Index* wygląda teraz jak na rysunku 8.67.

Rysunek 8.67.

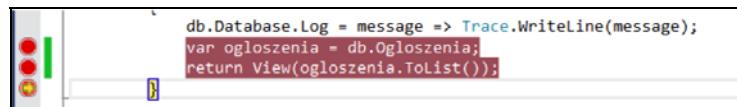
Akcja *Index* po zmianach



Uruchom ponownie aplikację i przejdź do ogłoszeń. Po dwukrotnym kliknięciu F5 przejdziesz do ostatniego punktu przerwania (rysunek 8.68).

Rysunek 8.68.

Przejście do ostatniego punktu przerwania



Metoda `ToList()` została wykonana, dane zostały pobrane, a zapytanie wyświetliło się w oknie *Output*. Po przejściu do okna *Watch*, w którym śledzi się wartości obiektu kontekstu, zobaczyś, że 10 ogłoszeń zostało zapisanych do kontekstu (pamięci operacyjnej) (rysunek 8.69).

Watch 1	
Name	Value
db	{Repositorium.Models.OglContext}
base	{Repositorium.Models.OglContext}
Kategorie	[SELECT [Extent1].[KluczPodstawowyKategorii] AS [KluczPodstawowyKategorii], [Extent1].[Nazwa] AS [Nazwa], [Extent1].[ParentId] AS [ParentId]
Ogłoszenia	[SELECT [Extent1].[Id] AS [Id], [Extent1].[Tresc] AS [Tresc], [Extent1].[Titul] AS [Titul], [Extent1].[DataDodania] AS [DataDodania], [Extent1].[
base	[SELECT [Extent1].[Id] AS [Id], [Extent1].[Tresc] AS [Tresc], [Extent1].[Titul] AS [Titul], [Extent1].[DataDodania] AS [DataDodania], [Extent1].[
Local	Count = 10
[0]	{System.Data.Entity.DynamicProxies.Ogloszenie_1BF42DDE731C37E485D302A26982A21BA9D453DA8970A40D767E7E4D5FCB22C}
[1]	{System.Data.Entity.DynamicProxies.Ogloszenie_1BF42DDE731C37E485D302A26982A21BA9D453DA8970A40D767E7E4D5FCB22C}
[2]	{System.Data.Entity.DynamicProxies.Ogloszenie_1BF42DDE731C37E485D302A26982A21BA9D453DA8970A40D767E7E4D5FCB22C}
[3]	{System.Data.Entity.DynamicProxies.Ogloszenie_1BF42DDE731C37E485D302A26982A21BA9D453DA8970A40D767E7F4D5FCB22C}
[4]	{System.Data.Entity.DynamicProxies.Ogloszenie_1BF42DDE731C37E485D302A26982A21BA9D453DA8970A40D767E7E4D5FCB22C}
[5]	{System.Data.Entity.DynamicProxies.Ogloszenie_1BF42DDE731C37E485D302A26982A21BA9D453DA8970A40D767E7E4D5FCB22C}
[6]	{System.Data.Entity.DynamicProxies.Ogloszenie_1BF42DDE731C37E485D302A26982A21BA9D453DA8970A40D767E7E4D5FCB22C}
[7]	{System.Data.Entity.DynamicProxies.Ogloszenie_1BF42DDE731C37E485D302A26982A21BA9D453DA8970A40D767E7E4D5FCB22C}
[8]	{System.Data.Entity.DynamicProxies.Ogloszenie_1BF42DDE731C37E485D302A26982A21BA9D453DA8970A40D767E7F4D5FCB22C}
[9]	{System.Data.Entity.DynamicProxies.Ogloszenie_1BF42DDE731C37E485D302A26982A21BA9D453DA8970A40D767E7E4D5FCB22C}
Raw Vir	
Non-Publi	
Results View Expanding the Results View will enumerate the IEnumerable	

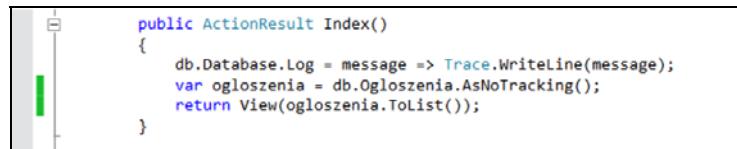
Rysunek 8.69. Ogłoszenia zapisane do kontekstu

Teraz bez wywołania metody `AsNoTracking()` dane są śledzone przez kontekst. W operacjach tylko do odczytu jest to marnowaniem pamięci i zasobów serwera, dlatego konieczne jest stosowanie metody `AsNoTracking()`. W oknie *Output* (logowanie zapytań z EF6) nie widać różnicy w czasie wykonania zapytania pomiędzy `AsNoTracking()` i bez `AsNoTracking()`, ponieważ zapytanie jest wykonywane tak samo. Dopiero po pobraniu z bazy dane są zapisywane do kontekstu lub nie.

Przerwij teraz działanie programu i ponownie dodaj metodę `AsNoTracking()`, po czym wyłącz punkt przerwań poprzez kliknięcie czerwonych kropek. Metoda wygląda jak na rysunku 8.70:

Rysunek 8.70.

Zapytanie z wykorzystaniem metody `AsNoTracking()`



Etap 2. Krok 3. Poprawa wyglądu i optymalizacja pod kątem SEO

Kolejnym krokiem będzie poprawa wyglądu za pomocą domyślnie dołączonej do projektu MVC biblioteki Bootstrap oraz optymalizacja strony pod kątem wyszukiwarki Google.

Poprawa wyglądu strony za pomocą Twitter Bootstrap

Aplikacja wygląda nieciekawie — wszystko jest białe (rysunek 8.71). Dodaj zatem kolory za pomocą biblioteki Bootstrap¹¹.

Lista Ogłoszeń				
Dodaj nowe ogłoszenie				
UżytkownikaId	Treść ogłoszenia:	Tytuł ogłoszenia:	Data dodania:	
73184b1f-5f56-4b66-ab48-f5d337a36c17	Treść ogłoszenia1	Tytuł ogłoszenia1	2014-06-26	Edytuj Szczegóły Usuń
73184b1f-5f56-4b66-ab48-f5d337a36c17	Treść ogłoszenia2	Tytuł ogłoszenia2	2014-06-25	Edytuj Szczegóły Usuń
73184b1f-5f56-4b66-ab48-f5d337a36c17	Treść ogłoszenia3	Tytuł ogłoszenia3	2014-06-24	Edytuj Szczegóły Usuń
73184b1f-5f56-4b66-ab48-f5d337a36c17	Treść ogłoszenia4	Tytuł ogłoszenia4	2014-06-23	Edytuj Szczegóły Usuń
73184b1f-5f56-4b66-ab48-f5d337a36c17	Treść ogłoszenia5	Tytuł ogłoszenia5	2014-06-22	Edytuj Szczegóły Usuń
73184b1f-5f56-4b66-ab48-f5d337a36c17	Treść ogłoszenia6	Tytuł ogłoszenia6	2014-06-21	Edytuj Szczegóły Usuń
73184b1f-5f56-4b66-ab48-f5d337a36c17	Treść ogłoszenia7	Tytuł ogłoszenia7	2014-06-20	Edytuj Szczegóły Usuń
73184b1f-5f56-4b66-ab48-f5d337a36c17	Treść ogłoszenia8	Tytuł ogłoszenia8	2014-06-19	Edytuj Szczegóły Usuń
73184b1f-5f56-4b66-ab48-f5d337a36c17	Treść ogłoszenia9	Tytuł ogłoszenia9	2014-06-18	Edytuj Szczegóły Usuń
73184b1f-5f56-4b66-ab48-f5d337a36c17	Treść ogłoszenia10	Tytuł ogłoszenia10	2014-06-17	Edytuj Szczegóły Usuń

Rysunek 8.71. Podstawowy wygląd aplikacji

Aby ożywić aplikację, za pomocą biblioteki Bootstrap dodaj kolorowe przyciski. Na stronie z listą ogłoszeń są cztery typy przycisków. Każdemu przypisano inny kolor. Bootstrap domyślnie posiada kilka predefiniowanych stylów dla przycisków (rysunek 8.72).

Rysunek 8.72.

Przyciski Bootstrap



Dla linku *Dodaj nowe ogłoszenie* użyj klasy *Primary*, a więc będzie to niebieski przycisk.

W pliku z widokiem *Index*, znajdującym się w folderze *Views/Ogłoszenie*, zamień linię:

```
<p>
    @Html.ActionLink("Dodaj nowe ogłoszenie", "Create")
</p>
```

¹¹ Listę komponentów można znaleźć na stronie: <http://getbootstrap.com/components/>.

na:

```
<p>
    @Html.ActionLink("Dodaj nowe ogłoszenie", "Create", null,
        new { @class = "btn btn-primary" })
</p>
```

Skorzystaj tutaj z HTML helpera o nazwie ActionLink i dodaj klasę CSS dla elementu HTML:

```
, null, new { @class = "btn btn-primary" }
```

Kod HTML wygenerowany przez ten HTML helper wygląda następująco:

```
<p>
    <a class="btn btn-primary" href="/Ogłoszenie/Create">
        →Dodaj nowe ogłoszenie</a>
</p>
```

Wygląd aplikacji po zmianach prezentuje rysunek 8.73.

Lista Ogłoszeń			
Dodaj nowe ogłoszenie			
UżytkownikaId	Treść ogłoszenia:	Tytuł ogłoszenia:	Data dodania:
73184b1f-5f56-4b66-ab48-f5d337a36c17	Treść ogłoszenia1	Tytuł ogłoszenia1	2014-06-26
73184b1f-5f56-4b66-ab48-f5d337a36c17	Treść ogłoszenia2	Tytuł ogłoszenia2	2014-06-25
73184b1f-5f56-4b66-ab48-f5d337a36c17	Treść ogłoszenia3	Tytuł ogłoszenia3	2014-06-24
73184b1f-5f56-4b66-ab48-f5d337a36c17	Treść ogłoszenia4	Tytuł ogłoszenia4	2014-06-23

Rysunek 8.73. Wygląd aplikacji po zmianie koloru przycisku

Dla pozostałych linków dodaj klasy:

```
<td>
    @Html.ActionLink("Szczegóły", "Details", new { id = item.Id },
        new { @class = "btn btn-warning" })
    <br />
    @Html.ActionLink("Edytuj ", "Edit", new { id = item.Id },
        new { @class = "btn btn-primary" })
    <br />
    @Html.ActionLink("Usuń", "Delete", new { id = item.Id },
        new { @class = "btn btn-danger" })
</td>
```

Wygląd aplikacji po kolejnych zmianach prezentuje rysunek 8.74.

Podświetlanie wierszy za pomocą CSS

Kolejnym krokiem będzie dodanie podświetlenia do pierwszego wiersza z nagłówkami (<th>):

```
tr:first-child{
    background-color:#efefef;
}
```

Lista Ogłoszeń			
UzytkownikId	Treść ogłoszenia:	Tytuł ogłoszenia:	Data dodania:
73184b1f-5f56-4b66-ab48-f5d337a36c17	Treść ogłoszenia1	Tytuł ogłoszenia1	2014-06-26
			Szczegóły Edytuj Usuń
73184b1f-5f56-4b66-ab48-f5d337a36c17	Treść ogłoszenia2	Tytuł ogłoszenia2	2014-06-25
			Szczegóły Edytuj Usuń
73184b1f-5f56-4b66-ab48-f5d337a36c17	Treść ogłoszenia3	Tytuł ogłoszenia3	2014-06-24
			Szczegóły Edytuj Usuń
73184b1f-5f56-4b66-ab48-f5d337a36c17	Treść ogłoszenia4	Tytuł ogłoszenia4	2014-06-23
			Szczegóły Edytuj Usuń

Rysunek 8.74. Wygląd aplikacji po zmianie stylu dla kolejnych przycisków

Następnie podświetl wiersz po najechaniu na niego (<tr>):

```
tr:hover td{
    background-color:#efefef;
}
```

oraz dodaj odstępy pomiędzy przyciskami:

```
.btn {
    margin:2px;
}
```

Aby dodać style do tworzonej strony w folderze *Content*, należy wyedytować plik *Site.css* i na końcu pliku dopisać wcześniej przedstawiony kod.

Teraz wystarczy zapisać zmiany w pliku i odświeżyć stronę (zmiany w plikach z widokiem nie wymagają komplikacji i można je przeprowadzać w trakcie działania programu). Teraz aplikacja wygląda nieco inaczej (rysunek 8.75).

Optymalizacja pod kątem pozycjonowania — SEO

Każda podstrona powinna zawierać tytuł, metaopis i słowa kluczowe. Do pliku z szablonem *_Layout.cshtml* należy dodać pola *description* i *keywords*, ponieważ znacznik *title* został dodany domyślnie. Edytuj w folderze *Views/Shared* plik *_Layout.cshtml*.

Nagłówek dokumentu HTML wygląda następująco:

```
<head>
<meta charset="utf-8" />
<meta name="viewport" content="width=device-width, initial-scale=1.0">
<title>@ViewBag.Title - My ASP.NET Application</title>
```

Lista Ogłoszeń			
UżytkownikId	Treść ogłoszenia:	Tytuł ogłoszenia:	Data dodania:
73184b1f-5f56-4b66-ab48-f5d337a36c17	Treść ogłoszenia1	Tytuł ogłoszenia1	2014-06-26
			Szczegóły Edytuj Usuń
73184b1f-5f56-4b66-ab48-f5d337a36c17	Treść ogłoszenia2	Tytuł ogłoszenia2	2014-06-25
			Szczegóły Edytuj Usuń
73184b1f-5f56-4b66-ab48-f5d337a36c17	Treść ogłoszenia3	Tytuł ogłoszenia3	2014-06-24
			Szczegóły Edytuj Usuń
73184b1f-5f56-4b66-ab48-f5d337a36c17	Treść ogłoszenia4	Tytuł ogłoszenia4	2014-06-23
			Szczegóły Edytuj Usuń

Rysunek 8.75. Wygląd aplikacji po kolejnych zmianach

```
@Styles.Render("~/Content/css")
@Scripts.Render("~/bundles/modernizr")
</head>
```

Dodaj następujące dwie linijki:

```
<meta name="description" content="@ViewBag.Opis" />
<meta name="keywords" content="@ViewBag.SlоваКлючевые" />
```

i zmień nazwę właściwości ViewBag z Title na Tytuł:

```
<title>@ViewBag.Tytuł</title>
```

Ostatecznie nagłówek wygląda następująco:

```
<head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width,
        initial-scale=1.0">
    <title>@ViewBag.Tytuł</title>
    <meta name="description" content="@ViewBag.Opis" />
    <meta name="keywords" content="@ViewBag.SlоваКлючевые" />
    @Styles.Render("~/Content/css")
    @Scripts.Render("~/bundles/modernizr")
</head>
```

Przejdź teraz do widoku Index z folderu *Ogłoszenia*. Na samej górze znajduje się kod:

```
@model IEnumerable<Repozytorium.Models.Ogłoszenie>

 @{
    ViewBag.Title = "Index";
}
```

Do właściwości ViewBag dodaj Opis i SłowaKluczowe oraz zamień nazwę z Title na Tytuł:

```
@model IEnumerable<Repozytorium.Models.Ogłoszenie>

{@
    ViewBag.Tytuł = "Lista ogłoszeń - metatytuł do 60 znaków";
    ViewBag.Opis = "Lista ogłoszeń z naszej aplikacji
                    ↪ metaopis do 160 znaków";
    ViewBag.SłowaKluczowe = "Lista, ogłoszeń, słowa,
                            ↪kluczowe, aplikacja";
}
```

Po najechaniu na zakładkę w przeglądarce wyświetli się tytuł (rysunek 8.76).

Rysunek 8.76.

Tytuł strony



Oto kod HTML wygenerowany za pomocą polecenia wyświetlającego źródło strony w dowolnej przeglądarce (rysunek 8.77):

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width,
        ↪initial-scale=1.0">
    <title>Lista ogłoszeń - metatytuł do 60 znaków</title>
    <meta name="description" content="Lista ogłoszeń z
        ↪naszej aplikacji - metaopis do 160 znaków" />
    <meta name="keywords" content="Lista, ogłoszeń, słowa,
        ↪kluczowe, aplikacja" />
    <link href="/Content/css?v=kgpd4rq0lsz-ifY0XZdVfBat-
        ↪ce3K9MqghW1bMzncl" rel="stylesheet" />
    <script src="/bundles/modernizr?v=K-
        FFPnTIXPUlQamnX3qHX_A5r7TM2xbAguEmpm3041"> </script>
</head>
```

Rysunek 8.77.

Wygląd
w wyszukiwarce
Google

Podgląd źródła strony - Pomoc Opery

help.opera.com/Windows/12.10/pl/sourceview.html ▾

Źródło strony wyświetlane jest na nowej karcie. Przeglądanie kodu jest łatwiejsze
 dzięki podświetlaniu składni. Podgląd źródła jest nie tylko przeglądarką, ale ...

->Tytuł
-> Opis

W zależności od zapytania wyszukiwarka może wyświetlić treść ze strony zamiast metaopisu, jeśli uzna, że treść będzie bardziej odpowiednia niż opis strony.

Etap 2. Podsumowanie

Na tym etapie istnieje aplikacja, która potrafi wyświetlić listę ogłoszeń i jest zoptymalizowana pod wyszukiwarki. Pozostaje jeszcze kwestia ustawiania odpowiednich tytułów i opisów dla konkretnych podstron. Pozycjonowane będą kategorie, dlatego w klasie z modelem dla kategorii zostały dodane pola, w których będzie można zapisywać tytuł, opis i słowa kluczowe, a następnie wyświetlać je dla wybranej kategorii. Do poprawy pozostały jeszcze pozostałe akcje z kontrolera, a więc Create, Edit,

Details i Delete, jednak wcześniej dokonamy zmiany aplikacji od strony architektonicznej. Im mniej akcji, tym mniej przerabiania, dlatego pozostałe akcje zostaną poprawione i opisane w dalszej części.

Etap 3. Krok 1. Poprawa architektury aplikacji

W wygenerowanym przez Visual Studio kodzie kontrolera znajdują się zapytania LINQ i obiekt kontekstu. Warstwa kontrolerów jest odpowiedzialna za interakcję z użytkownikami i obsługę zdarzeń, dlatego złą praktyką jest trzymanie w niej logiki dostępu do danych. Aby poprawić architekturę aplikacji, przeniesiemy zapytania LINQ i obiekt kontekstu do osobnych metod, a kontroler będzie tylko wywoływał te metody.

Przeniesienie zapytania LINQ do osobnej metody

W dalszym ciągu zmianom podlega akcja Index z kontrolera Ogloszenie, która wyświetla ogłoszenia. Usuniesz jedynie metodę `ToList()`, ponieważ nie ma potrzeby natychmiastowego pobierania danych. Aktualnie metoda wygląda następująco:

```
public ActionResult Index()
{
    var ogloszenia = db.Ogloszenia.AsNoTracking();
    return View(ogloszenia);
}
```

Następnie przenieś zapytanie oraz funkcję odpowiedzialną za logowanie zapytań do osobnej metody o nazwie `PobierzOgloszenia()`, która będzie zwracać typ `IQueryable<Ogloszenie>`. Kod po przenosinach wygląda następująco:

```
public ActionResult Index()
{
    var ogloszenia = PobierzOgloszenia();
    return View(ogloszenia);
}

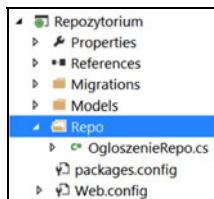
public IQueryable<Ogloszenie> PobierzOgloszenia()
{
    db.Database.Log = message => Trace.WriteLine(message);
    return db.Ogloszenia.AsNoTracking();
}
```

Akcja `Index` nie korzysta już z obiektu kontekstu o nazwie `db` — otrzymuje tylko gotowe dane do wyświetlenia. Jednak w dalszym ciągu metoda `PobierzOgloszenia()` znajduje się w klasie kontrolera i obiekt kontekstu jest tworzony w kontrolerze.

Przeniesienie metody do repozytorium

Kolejnym krokiem jest przeniesienie metody `PobierzOgloszenia()` do projektu Repozytorium odpowiedzialnego za kontakt z bazą danych. W projekcie Repozytorium dodaj folder o nazwie `Repo`, a w nim plik (*Add/Class*) o nazwie `OgloszenieRepo`. Na rysunku 8.78 zaprezentowano strukturę projektu.

Rysunek 8.78.
*Struktura projektu
 Repozytorium
 po zmianach*



Wytnij metodę PobierzOgłoszenia() z kontrolera i wklej ją w klasie OgłoszenieRepo. Przenieś również kod odpowiedzialny za tworzenie kontekstu znajdujący się na samym początku klasy oraz część kodu odpowiedzialną za logowanie zapytań do bazy danych. Zainportuj także biblioteki dla części kodu podkreślonych na czerwono.

Kod klasy OgłoszenieRepo wygląda teraz następująco:

```
public class OgłoszenieRepo
{
    private OglContext db = new OglContext();

    public IQueryable<Ogłoszenie> PobierzOgłoszenia()
    {
        db.Database.Log = message => Trace.WriteLine(message);
        return db.Ogłoszenia.AsNoTracking();
    }
}
```

Wróćmy do kontrolera — ponieważ usunęliśmy obiekt kontekstu z kontrolera, wszystkie akcje będą podświetlone na czerwono. Metody, z których nie korzystasz (wszystkie poza akcją Index), zostaną zakomentowane. Zaznacz metody od Details aż do Dispose i kliknij *Ctrl+K*, a następnie *Ctrl+C*. Część kodu prezentuje rysunek 8.79.

Rysunek 8.79.
*Kod kontrolera
 po zakomentowaniu*

```
public class OgłoszenieController : Controller
{
    // GET: Ogłoszenie
    public ActionResult Index()
    {
        db.Database.Log = message => Trace.WriteLine(message);
        var ogłoszenia = PobierzOgłoszenia();
        return View(ogłoszenia);
    }

    // GET: Ogłoszenie/Details/5
    public ActionResult Details(int? id)
    //{
    //    if (id == null)
    //    {
    //        return new HttpStatusCodeResult(HttpStatusCode.BadRequest);
    //    }
    //    Ogłoszenie ogłoszenie = db.Ogłoszenia.Find(id);
    //    if (ogłoszenie == null)
    //    {
    //        return HttpNotFound();
    //    }
    //    return View(ogłoszenie);
    //}

    // GET: Ogłoszenie/Create
    public ActionResult Create()
    //{
    //    ViewBag.UzytkownikId = new SelectList(db.Users, "Id", "Email");
    //    return View();
    //}
}
```

W miejscu gdzie wcześniej znajdował się obiekt kontekstu (na początku klasy), teraz został utworzony obiekt repozytorium:

```
OgłoszenieRepo repo = new OgłoszenieRepo();
```

i dodana została dyrektywa using na początku pliku:

```
using Repozytorium.Repo;
```

Kolejnym krokiem jest wywołanie metody PobierzOgłoszenia() na rzecz tego obiektu. Zamień:

```
var ogłoszenia = PobierzOgłoszenia();
```

na:

```
var ogłoszenia = repo.PobierzOgłoszenia();
```

Ostatecznie klasa OgłoszenieController wygląda następująco:

```
public class OgłoszenieController : Controller
{
    OgłoszenieRepo repo = new OgłoszenieRepo();
    //GET: /Ogłoszenie/
    public ActionResult Index()
    {
        var ogłoszenia = repo.PobierzOgłoszenia();
        return View(ogłówzenia);
    }
    // Tutaj zakomentowany kod/akcje
}
```

Teraz możesz uruchomić aplikację. Działa tak samo jak poprzednio, ale kod kontrolera jest dużo bardziej elegancki.

Etap 3. Krok 2. Zastosowanie kontenera Unity — IoC

Pomimo że architektura aplikacji wygląda sensownie, to pozostała jeszcze jedna rzecz do zrobienia. W dalszym ciągu w kontrolerze tworzona jest instancja repozytorium. Jeśli utworzonych zostanie kilkudziesiąt kontrolerów i w każdym będzie tworzona instancja tego samego repozytorium, to w wypadku gdy zajdzie potrzeba zamiany repozytorium na nowszą wersję, np. OgłoszenieRepoNowe, konieczne będzie wprowadzenie zmian we wszystkich plikach kontrolerów. Aby przenieść odpowiedzialność za to, jaka instancja repozytorium jest tworzona w danym kontrolerze, wykorzystamy kontener IoC — odwrócenie sterowania. Aby była możliwość wstrzykiwania różnych typów repozytoriów dla kontrolera, w miejscu gdzie używaliśmy klasy OgłoszenieRepo, trzeba użyć interfejsu, a więc szkieletu, który jest implementowany przez klasy oraz dodać konstruktor.

Wstrzykiwanie repozytorium poprzez konstruktor w kontrolerze

Aby dodać konstruktor, użyj snippetów w Visual Studio. Wpisz słowo ctor i kliknij dwukrotnie klawisz Tab. Zostanie wygenerowany następujący kod konstruktora:

```
public OgloszenieController()
{
}
```

Do konstruktora będzie wstrzykiwany (ustawiany) typ repozytorium. Konstruktor ma wyglądać następująco:

```
public OgloszenieController(IOgloszenieRepo repo)
{
    _repo = repo;
}
```

Przed konstruktorem dodaj pole prywatne, dla którego zostanie wstrzyknięta instancja repozytorium:

```
private readonly IOgloszenieRepo _repo;
```

Klasa OgloszenieController wygląda teraz następująco:

```
public class OgloszenieController : Controller
{
    private readonly IOgloszenieRepo _repo;

    public OgloszenieController(IOgloszenieRepo repo)
    {
        _repo = repo;
    }

    // GET: /Ogloszenie/
    public ActionResult Index()
    {
        var ogloszenia = _repo.PobierzOgloszenia();
        return View(ogloszenia);
    }

    // Tutaj zakomentowany kod/akcje
}
```

Konstruktor jest wywoływany w trakcie generowania obiektu klasy, dlatego podczas tworzenia kontrolera zostanie również utworzona instancja klasy repozytorium.

Tworzenie interfejsu dla repozytorium

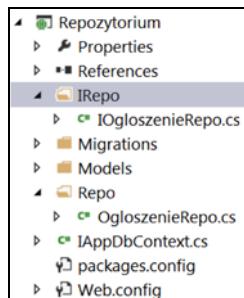
Kolejnym krokiem jest utworzenie interfejsu `IOgloszenieRepo`. W projekcie Repozytorium dodaj folder o nazwie `IRepo`, a w nim plik o nazwie `IOgloszenieRepo`. Struktura projektu wygląda teraz następująco (rysunek 8.80).

Interfejs zawiera jedynie deklaracje metod, czyli nazwy, parametry i zwracany typ metod, które mają się znaleźć w klasie implementującej ten interfejs. Ponieważ repozytorium posiada tylko jedną metodę `PobierzOgloszenia()`, interfejs będzie zawierał tylko jedną deklarację.

Poniżej zaprezentowano kod interfejsu `IOgloszenieRepo`:

Rysunek 8.80.

*Utworzenie interfejsu
IOgloszenieRepo*



```
public interface IOgloszenieRepo
{
    IQueryable<Ogloszenie> PobierzOgloszenia();
}
```

Repozytorium OgloszenieRepo musi dziedziczyć po interfejsie IOgloszenieRepo i implementować jego składowe. Aby dziedziczyć po interfejsie, dodaj następujący kod po prawej stronie od nazwy klasy:

```
: IOgloszenieRepo
```

Klasa repozytorium wygląda teraz następująco:

```
public class OgloszenieRepo : IOgloszenieRepo
{
    private OglContext db = new OglContext();

    public IQueryable<Ogloszenie> PobierzOgloszenia()
    {
        db.Database.Log = message => Trace.WriteLine(message);
        var ogloszenia = db.Ogloszenia.AsNoTracking();
        return ogloszenia;
    }
}
```

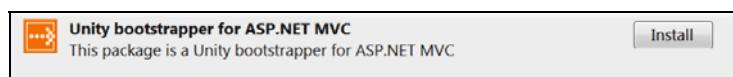
Po dodaniu interfejsu, jeśli nie zrobiłeś tego wcześniej, zaimplementuj biblioteki w plikach *OgloszenieController*, *OgloszenieRepo* i *IOgloszenieRepo*.

Instalacja kontenera IoC Unity

Po przygotowaniu klasy i interfejsu trzeba zainstalować bibliotekę IoC, która będzie się zajmowała wstrzykiwaniem implementacji klasy przez konstruktor. Za pośrednictwem NuGet w projekcie OGL zainstaluj bibliotekę *Unity.Mvc* (rysunek 8.81). Jest to jedna z najpopularniejszych bibliotek IoC.

Rysunek 8.81.

*Instalacja biblioteki
Unity*



Po zainstalowaniu biblioteki w klasie *UnityConfig* w folderze *App_Start* dopisz następujący kod do metody *RegisterTypes()*:

```
container.RegisterType<AccountController>(new InjectionConstructor());
container.RegisterType<ManageController>(new InjectionConstructor());
container.RegisterType<IOgloszenieRepo, OgloszenieRepo>(new
    ↳PerRequestLifetimeManager());
```

Kod klasy wygląda teraz następująco:

```
/// <summary>
/// Specifies the Unity configuration for the main container.
/// </summary>
public class UnityConfig
{
    #region Unity Container
    private static Lazy<IUnityContainer> container = new
        ↳Lazy<IUnityContainer>(() =>
    {
        var container = new UnityContainer();
        RegisterTypes(container);
        return container;
    });
    /// <summary>
    /// Gets the configured Unity container.
    /// </summary>
    public static IUnityContainer GetConfiguredContainer()
    {
        return container.Value;
    }
    #endregion

    /// <summary>Registers the type mappings with the Unity container.</summary>
    /// <param name="container">The unity container to configure.</param>
    /// <remarks>There is no need to register concrete types such as controllers or API
    /// controllers (unless you want to
    /// change the defaults), as Unity allows resolving a concrete type even if it was not
    /// previously registered.</remarks>
    public static void RegisterTypes(IUnityContainer container)
    {
        // NOTE: To load from web.config uncomment the line below. Make sure to add a
        // Microsoft.Practices.Unity.Configuration to the using statements.
        // container.LoadConfiguration();

        // TODO: Register your types here
        // container.RegisterType<IPrductRepository, ProductRepository>();

        container.RegisterType<AccountController>(new
            ↳InjectionConstructor());
        container.RegisterType<ManageController>(new
            ↳InjectionConstructor());
        container.RegisterType<IOgloszenieRepo, OgloszenieRepo>(new
            ↳PerRequestLifetimeManager());
    }
}
```

Znak `///` w powyższym kodzie oznacza komentarze do dokumentacji.

Zainimportuj biblioteki i uruchom aplikację.

Wstrzykiwanie kontekstu do repozytorium

Aktualnie istnieje kontroler, który nie posiada zależności od kontekstu, jednak teraz ta zależność jest w repozytorium. Repozytorium jest zależne od kontekstu. Kolejnym krokiem będzie użycie IoC do wstrzykiwania obiektu kontekstu dla repozytorium.

W analogiczny sposób tworzy się interfejs `IRepo`. Zamień słowo `class` na `interface`. Następnie wklej właściwości `DbSet` z kontekstu `OglContext`. Dodaj metodę `SaveChanges()` oraz właściwość `Database`, która służy do logowania zapytań do bazy danych. Zainimportuj biblioteki. Tak wygląda zawartość gotowego interfejsu:

```
public interface IOglContext
{
    DbSet<Kategoria> Kategorie { get; set; }
    DbSet<Ogloszenie> Ogloszenia { get; set; }
    DbSet<Uzytkownik> Uzytkownik { get; set; }
    DbSet<Ogloszenie_Kategoria> Ogloszenie_Kategoria { get; set; }

    int SaveChanges();
    Database Database { get; }
}
```

Kolejnym krokiem jest dodanie do kontekstu informacji, że dziedziczy po interfejsie `IOglContext`. Przejdz do pliku `OglContext` i zamień pierwszą linię z:

```
public class OglContext : IdentityDbContext
```

na:

```
public class OglContext : IdentityDbContext, IOglContext
```

po czym zainimportuj bibliotekę dla interfejsu.

Przejdz teraz do `OgloszenieRepo` i utwórz konstruktor, w którym będzie wstrzykiwana instancja repozytorium.

Zamień:

```
private OglContext db = new OglContext();
public IQueryable<Ogloszenie> PobierzOgloszenia()
{
    db.Database.Log = message => Trace.WriteLine(message);
    var ogloszenia = db.Ogloszenia.AsNoTracking();
    return ogloszenia;
}
```

na:

```
private readonly IOglContext _db;
public OgloszenieRepo(IOglContext db)
{
    _db = db;

    public IQueryable<Ogloszenie> PobierzOgloszenia()
    {
        _db.Database.Log = message => Trace.WriteLine(message);
    }
}
```

```
var ogłoszenia = _db.Ogłoszenia.AsNoTracking();
return ogłoszenia;
}
```

Ostatnią rzeczą jest skonfigurowanie kontenera Unity, aby wstrzykiwał w miejsce interfejsu `IoglContext` instancję klasy `OglContext`. Dodaj do metody `RegisterTypes()` w pliku `UnityConfig` poniższą linijkę:

```
container.RegisterType<IoglContext, OglContext>(new PerRequestLifetimeManager());
```

Po dodaniu metoda wygląda następująco:

```
public static void RegisterTypes(IUnityContainer container)
{
    container.RegisterType<AccountController>(new
        ↪InjectionConstructor());
    container.RegisterType<ManageController>(new
        ↪InjectionConstructor());

    container.RegisterType<IOgłoszenieRepo,
        ↪OgłoszenieRepo>(new PerRequestLifetimeManager());
    container.RegisterType<IoglContext, OglContext>(new
        ↪PerRequestLifetimeManager());
}
```

Zimportuj biblioteki. Można zbudować i uruchomić aplikację — wszystko powinno działać prawidłowo.

Cykl życia obiektu a kontener IoC

W aplikacjach internetowych dane są zazwyczaj pobierane tylko jeden raz i nie ma potrzeby przechowywania obiektu w pamięci po zakończonym żądaniu. Dlatego w metodzie `RegisterTypes()` ustawiono cykl życia obiektu jako `PerRequest` zarówno dla kontekstu, jak i repozytorium. Oznacza to, że przy każdym żądaniu zostanie utworzony nowy obiekt, a po zakończeniu żądania kontener IoC automatycznie wywoła metodę `Dispose()`:

```
protected override void Dispose(bool disposing)
{
    if (disposing)
    {
        db.Dispose();
    }
    base.Dispose(disposing);
}
```

Etap 3. Podsumowanie

W tym etapie przenieśliśmy całą logikę odpowiadającą za operacje na danych do osobnej warstwy (osobnego projektu). Od teraz kontrolery nie zajmują się pobieraniem danych, tylko obsługą zdarzeń i zwracaniem odpowiednich widoków.

Dodatkowo skorzystaliśmy z IoC. Teraz instancje repozytoriów wstrzykuje się w kontrolerach poprzez konstruktor. Pozwoliło to na usunięcie zależności kontrolera od konkretnego repozytorium (działa się na interfejsie), a typem wstrzykiwanych repozytoriów

steruje się w jednym miejscu — pliku konfiguracyjnym kontenera Unity. Następnie usunęliśmy zależność repozytorium od konkretnego kontekstu, korzystając z interfejsu `IOglContext` i wstrzykując instancję kontekstu poprzez konstruktor, tak jak w przypadku repozytorium. Wszystkie zależności pomiędzy warstwami lub modułami w aplikacji powinny być tworzone przy użyciu IoC i wstrzykiwaniu implementacji poprzez konstruktor.

Etap 4. Krok 1. Akcje Details, Create, Edit, Delete

Details

Akcja `Details` wyświetla szczegółowe informacje o pojedynczym ogłoszeniu. W kontrolerze `OgłoszenieController` zaznacz zakomentowany kod akcji `Details` i odkomentuj za pomocą skrótów `Ctrl+K` i `Ctrl+U`.

Kod akcji wygląda następująco:

```
public ActionResult Details(int? id)
{
    if (id == null)
    {
        return new HttpStatusCodeResult(HttpStatusCode.BadRequest);
    }

    Ogloszenie ogloszenie = db.Ogloszenia.Find(id);
    if (ogloszenie == null)
    {
        return HttpNotFound();
    }
    return View(ogloszenie);
}
```

Metoda Details() w repozytorium

Utwórz metodę w repozytorium, która pobierze ogłoszenie. Dodaj zatem w interfejsie `IOgloszenieRepo` następującą deklarację metody:

```
Ogloszenie GetOgloszenieById(int id);
```

Interfejs wygląda teraz następująco:

```
public interface IOgloszenieRepo : IDisposable
{
    IQueryble<Ogloszenie> PobierzOgloszenia();
    Ogloszenie GetOgloszenieById(int id);
}
```

Kolejnym krokiem jest implementacja metody `GetOgloszenieById` w repozytorium `OgłoszenieRepo`. Dodaj do repozytorium `OgłoszenieRepo` następującą metodę:

```
public Ogloszenie GetOgloszenieById(int id)
{
    Ogloszenie ogloszenie = _db.Ogloszenia.Find(id);
    return ogloszenie;
}
```

Ostatnim krokiem jest wykorzystanie w kontrolerze nowo dodanej metody. Zamień linijkę:

```
Ogłoszenie ogłoszenie = db.Ogłoszenia.Find(id);  
na:
```

```
Ogłoszenie ogłoszenie = _repo.GetOgłoszenieById((int)id);
```

Ponieważ metoda `GetOgłoszenieById()` z repozytorium jako parametr przyjmuje zmienną typu `int`, a nie `int?`, tak jak akcja `Details`, wykonywane jest rzutowanie typu `int?` na typ `int`. Przed wywołaniem metody sprawdzana jest wartość zmiennej, dlatego nie ma możliwości, aby rzutowanie się nie powiodło.

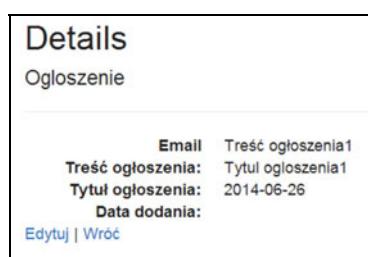
Ostatecznie akcja `Details` wygląda następująco:

```
// GET: /Ogłoszenie/Details/5  
public ActionResult Details(int? id)  
{  
    if (id == null)  
    {  
        return new HttpStatusCodeResult(HttpStatusCode.BadRequest);  
    }  
    Ogłoszenie ogłoszenie = _repo.GetOgłoszenieById((int)id);  
    if (ogłoszenie == null)  
    {  
        return HttpNotFound();  
    }  
    return View(ogłoszenie);  
}
```

Aktualizacja i optymalizacja SEO dla widoku Details

Po uruchomieniu aplikacji kliknij przycisk *Szczegóły* na podstronie z listą ogłoszeń (rysunek 8.82).

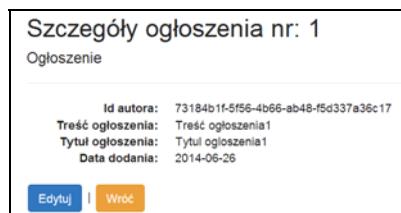
Rysunek 8.82.
Strona ze szczegółami ogłoszenia



Popraw wygląd aplikacji, tak jak zostało to zrobione dla akcji `Index` (rysunek 8.83, listing 8.7):

- ♦ ustaw tytuł, opis i słowa kluczowe,
- ♦ przetłumacz linki,
- ♦ dodaj kolorowe przyciski za pomocą Bootstrapa,
- ♦ wyświetl identyfikator użytkownika w miejsce e-maila.

Rysunek 8.83.
Strona ze szczegółami po wprowadzeniu zmian



Listing 8.7. Kod widoku Details po zmianach

```
@model Repository.Models.Ogłoszenie

{@
    ViewBag.Tytuł = "Szczegóły ogłoszenia nr" + Model.Id;
    ViewBag.Opis = "Szczegóły ogłoszenia nr" + Model.Id + "
        →Opis do Google";
    ViewBag.SłowaKluczowe = "Ogłoszenie, " + Model.Id + ",
        ↪szczegóły";
}

<h2>Szczegóły ogłoszenia nr @Model.Id</h2>

<div>
    <h4>Ogłoszenie</h4>
    <hr />
    <dl class="dl-horizontal">
        <dt>
            Id autora:
        </dt>
        <dd>
            @Html.DisplayFor(model => model.UzytkownikId)
        </dd>
        <dt>
            @Html.DisplayNameFor(model => model.Tresc)
        </dt>
        <dd>
            @Html.DisplayFor(model => model.Tresc)
        </dd>
        <dt>
            @Html.DisplayNameFor(model => model.Tytuł)
        </dt>
        <dd>
            @Html.DisplayFor(model => model.Tytuł)
        </dd>
        <dt>
            @Html.DisplayNameFor(model => model.DataDodania)
        </dt>
        <dd>
            @Html.DisplayFor(model => model.DataDodania)
        </dd>
    </dl>
</div>
<p>
    @Html.ActionLink("Edytuj", "Edit", new { id = Model.Id },
        new { @class = "btn btn-primary" })

```

```
@Html.ActionLink("Wróć", "Index", null, new { @class =
    ➔"btn btn-warning" })
</p>
```

Delete

Akcja Delete służy do usunięcia ogłoszenia. W kontrolerze OgloszenieController zaznacz zakomentowany kod metody Delete i użąd ponownie skrótów *Ctrl+K* i *Ctrl+U*, aby odkomentować kod akcji. Akcja Delete posiada dwie metody: GET (nazwa Delete) i POST (nazwa DeleteConfirmed). Należy odkomentować tylko pierwszą akcję o nazwie Delete. Pierwsza akcja jest identyczna jak metoda Details. Zamienia się w niej kod analogicznie jak dla akcji Details. Po zmianach metoda wygląda następująco:

```
public ActionResult Delete(int? id)
{
    if (id == null)
    {
        return new HttpStatusCodeResult(HttpStatusCode.BadRequest);
    }

    Ogloszenie ogloszenie = _repo.GetOgloszenieById((int)id);
    if (ogloszenie == null)
    {
        return HttpNotFound();
    }
    return View(ogloszenie);
}
```

Wykorzystujesz tutaj tę samą metodę z repozytorium, której użyłeś w akcji Details. Bez użycia repozytorium konieczne byłoby napisanie drugi raz tego samego zapytania, a w razie modyfikacji trzeba by nanieść zmiany w wielu miejscach (akcjach) w aplikacji. Jeśli używa się repozytorium, to modyfikacja jednej metody powoduje zmianę w wielu miejscach w aplikacji jednocześnie.

Aktualizacja widoku dla akcji Delete

Kolejnym krokiem jest poprawienie pliku z widokiem Delete (rysunek 8.84, listing 8.8). Ponownie wprowadź te same zmiany co w widoku Details, czyli:

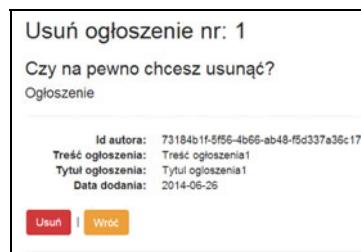
- ♦ ustaw tytuł, opis i słowa kluczowe,
- ♦ przetłumacz linki,
- ♦ dodaj kolorowe przyciski za pomocą Bootstrapa,
- ♦ wyświetl identyfikator użytkownika w miejscu e-maila.

Listing 8.8. Kod widoku Delete po zmianach

```
@model Repozytorium.Models.Ogloszenie

 @{
    ViewBag.Tytuł = "Usuń ogłoszenie nr" + Model.Id;
    ViewBag.Opis = "Usuń ogłoszenie nr" + Model.Id +
    ➔" Opis do Google";
```

Rysunek 8.84.
Wygląd strony
dla akcji Delete
po zmianach



```

ViewBag.SlowaKluczowe = "Ogłoszenie, " + Model.Id + ", usuń";
}

<h2>Usuń ogłoszenie nr @Model.Id</h2>

<h3>Czy na pewno chcesz usunąć?</h3>
<div>
    <h4>Ogłoszenie</h4>
    <hr />
    <dl class="dl-horizontal">
        <dt>
            Id autora:
        </dt>
        <dd>
            @Html.DisplayFor(model => model.UzytkownikId)
        </dd>
        <dt>
            @Html.DisplayNameFor(model => model.Tresc)
        </dt>
        <dd>
            @Html.DisplayFor(model => model.Tresc)
        </dd>
        <dt>
            @Html.DisplayNameFor(model => model.Tytul)
        </dt>
        <dd>
            @Html.DisplayFor(model => model.Tytul)
        </dd>
        <dt>
            @Html.DisplayNameFor(model => model.DataDodania)
        </dt>
        <dd>
            @Html.DisplayFor(model => model.DataDodania)
        </dd>
    </dl>
    @using (Html.BeginForm())
    {
        @Html.AntiForgeryToken()

        <div class="form-actions no-color">
            <input type="submit" value="Usuń"
                   class="btn btn-danger" />
            @Html.ActionLink("Wróć", "Index", null, new { @class =
                   "btn btn-warning" })
        </div>
    }
}

```

```
</div>
}
</div>
```

Metoda POST dla akcji Delete

Teraz możesz odkomentować (zaznaczyć, korzystając ze skrótów *Ctrl+K* i *Ctrl+U*) drugą metodę `DeleteConfirmed` (z `OgłoszenieController`), czyli metodę POST dla akcji `Delete`. Ponieważ nazwa metody jest inna niż nazwa akcji, należy użyć atrybutu `ActionName("Delete")`. Akcję oznacz atrybutem `[HttpPost]`. Domyślnie akcje są traktowane jako GET, dlatego dla poprzedniej akcji nie było atrybutu `[HttpGet]`.

Ciało metody `DeleteConfirmed()` przed przeróbką:

```
[HttpPost, ActionName("Delete")]
[ValidateAntiForgeryToken]
public ActionResult DeleteConfirmed(int id)
{
    Ogłoszenie ogłoszenie = db.Ogłoszenia.Find(id);
    db.Ogłoszenia.Remove(ogłoszenie);
    db.SaveChanges();
    return RedirectToAction("Index");
}
```

Przenieś teraz kod odpowiedzialny za usuwanie ogłoszenia do repozytorium.

Utwórz deklarację metody w interfejsie `IOgłoszenieRepo`:

```
bool UsunOgłoszenie(int id);
```

Metoda będzie zwracać wartość `true`, jeśli usuwanie zakończy się sukcesem, lub `false`, gdy wystąpi problem podczas usuwania.

Poniżej znajduje się implementacja metody `UsunOgłoszenie()`, którą należy dodać do repozytorium `OgłoszenieRepo`:

```
public bool UsunOgłoszenie(int id)
{
    Ogłoszenie ogłoszenie = _db.Ogłoszenia.Find(id);
    _db.Ogłoszenia.Remove(ogłoszenie);
    try
    {
        _db.SaveChanges();
        return true;
    }
    catch(Exception ex)
    {
        return false;
    }
}
```

Metoda `UsunOgłoszenie()` zwróci wartość `true`, jeśli zapis do bazy się powiedzie, lub wartość `false` w przeciwnym wypadku.

Metoda Delete() w repozytorium

Ostatnim krokiem jest aktualizacja metody DeleteConfirmed() z kontrolera OgloszenieController, aby korzystała z metody znajdującej się w repozytorium. W metodzie zaimplementuj rozwiązanie, które w razie niepowodzenia (metoda UsunOgloszenie zwróci false) spróbuje ponownie usunąć wybrane ogłoszenie. Po pomyślnym usunięciu metoda przenosi nas do akcji Index (po niepomyślnym również).

Ciało metody po aktualizacji wygląda jak poniżej:

```
// POST: /Ogloszenie/Delete/5
[HttpPost, ActionName("Delete")]
[ValidateAntiForgeryToken]
public ActionResult DeleteConfirmed(int id)
{
    for (int i = 0; i < 3; i++)
    {
        if (_repo.UsunOgloszenie(id))
            break;
    }
    return RedirectToAction("Index");
}
```

Kaskadowe usuwanie i błędy

Uruchomiliśmy aplikację, przeprowadziliśmy próbę usunięcia ogłoszenia, ale aplikacja wraca do akcji Index, a ogłoszenie nie zostaje usunięte. Podczas debugowania można zauważyc, że w bloku catch() w metodzie UsunOgloszenie z OgloszenieRepo zgłoszony jest wyjątek:

```
{"The DELETE statement conflicted with the REFERENCE constraint \"FK_dbo.
➥Ogloszenie_Kategoria_dbo.Ogloszenie_OgloszenieId\". The conflict occurred in
➥database \"aspnet-0GL22-20140630015355\", table \"dbo.Ogloszenie_Kategoria\",
➥column 'OgloszenieId'.\r\nThe statement has been terminated."}
```

Dzieje się tak, ponieważ ogłoszenie jest powiązane z kategoriami poprzez tabelę Ogloszenie_Kategoria. Aby usunięcie ogłoszenia było możliwe, konieczne jest wcześniejsze usunięcie wpisów z tabeli Ogloszenie_Kategoria, które w kluczu obcyim OgloszenieId są powiązane z usuwanym ogłoszeniem. Zatem w akcji Delete zostaną usunięte wszystkie wiersze z tabeli Ogloszenie_Kategoria, dla których OgloszenieId jest równe id ogłoszenia, a dopiero później będzie wykasowane również ogłoszenie. Jeśli w klasie kontekstu ustawilbyś kaskadowe usuwanie (ang. *Cascade Delete*) na true, to nie byłoby konieczności pisania dodatkowej metody i baza danych automatycznie usunęłaby dane z tabeli Ogloszenie_Kategoria.

Dodaj jeszcze następującą prywatną (dostępną tylko dla klasy repozytorium) metodę do OgloszenieRepo:

```
private void UsunPowiazanieOgloszenieKategoria(int idOgloszenia)
{
    var list = _db.Ogloszenie_Kategoria.Where(o=>o.OgloszenieId
➥== idOgloszenia);
    foreach (var el in list)
    {
```

```
        _db.Ogloszenie_Kategoria.Remove(e1);  
    }  
}
```

i wywołaj ją na początku w metodzie UsunOgloszenie():

```
public bool UsunOgloszenie(int id)  
{  
    UsunPowiazanieOgloszenieKategoria(id);  
  
    Ogloszenie ogloszenie = _db.Ogloszenia.Find(id);  
  
    _db.Ogloszenia.Remove(ogloszenie);  
    try  
    {  
        _db.SaveChanges();  
        return true;  
    }  
    catch(Exception ex)  
    {  
        return false;  
    }  
}
```

Uruchom ponownie aplikację i spróbuj usunąć ogłoszenie. Tym razem zadanie zostanie wykonane.

Przeniesienie metody SaveChanges() poza repozytorium

Sprawa komplikuje się, w przypadku gdy trzeba usunąć kilka ogłoszeń. Konieczne byłoby kilkakrotne wywołanie metody UsunOgloszenie() i za każdym razem wykonywana byłaby osobna transakcja do bazy danych. Dlatego w kolejnym kroku dodasz do repozytorium metodę SaveChanges(), która będzie wywoływana w kontrolerze. Takie rozwiązanie pozwoli na wykonywanie wielu operacji na obiekcie kontekstu bez konieczności zapisywania danych do bazy po każdej zmianie.

Dodaj deklarację metody do interfejsu IOgloszenieRepo:

```
void SaveChanges();
```

Do klasy OgloszenieRepo dodaj definicję metody:

```
public void SaveChanges()  
{  
    _db.SaveChanges();  
}
```

W metodzie UsunOgloszenie() usuń kod odpowiedzialny za zapis do bazy danych i zmień typ zwracany z metody na void. Ponieważ zapis odbywa się w kontrolerze, nie ma potrzeby zwracać informacji, czy zapis się powiodł. W interfejsie IOgloszenieRepo zmień również deklarację metody, aby zwracała typ void. Metoda UsunOgloszenie() po aktualizacji powinna wyglądać następująco:

```
public void UsunOgloszenie(int id)  
{  
    UsunPowiazanieOgloszenieKategoria(id);  
}
```

```

        Ogloszenie ogloszenie = _db.Ogloszenia.Find(id);
        _db.Ogloszenia.Remove(ogloszenie);
    }
}

```

Ostatnim krokiem jest wywołanie metody `SaveChanges()` na obiekcie repozytorium w kontrolerze. Kod metody `DeleteConfirmed()` z kontrolera wygląda teraz jak poniżej:

```

// POST: /Ogloszenie/Delete/5
[HttpPost, ActionName("Delete")]
[ValidateAntiForgeryToken]
public ActionResult DeleteConfirmed(int id)
{
    _repo.UnsunOgloszenie(id);
    try
    {
        _repo.SaveChanges();
    }
    catch
    {
    }

    return RedirectToAction("Index");
}

```

Teraz masz pełną kontrolę nad tym, kiedy dane są zapisywane do bazy danych. Jednak w dalszym ciągu nie jest to dobre rozwiązanie. W razie wystąpienia błędu użytkownik jest przekierowywany do akcji `Index`, czyli do listy ogłoszeń. Przygotujemy rozwiązanie, które w przypadku gdy wystąpi błąd podczas usuwania, wróci do tego samego widoku i wyświetli informacje o błędzie.

Obsługa błędów w akcji Delete

Na początek przebudujesz metodę `Delete()` z kontrolera. Dodaj jeden opcjonalny parametr typu `bool` o nazwie `bład`. Jeśli parametr zostanie przekazany do widoku, będzie to oznaczać, że wystąpił błąd i trzeba wyświetlić komunikat o błędzie. Aby przekazać z kontrolera do widoku informację o tym, że wystąpił błąd, użyj właściwości `ViewBag` o nazwie `Bład`. Kod poprawionej akcji `Delete` z kontrolera prezentuje się jak poniżej:

```

public ActionResult Delete(int? id, bool? blad)
{
    if (id == null)
    {
        return new HttpStatusCodeResult(HttpStatusCode.BadRequest);
    }

    Ogloszenie ogloszenie = _repo.GetOgloszenieById((int)id);
    if (ogloszenie == null)
    {
        return HttpNotFound();
    }
    if(blad !=null)
        ViewBag.Blad = true;

    return View(ogloszenie);
}

```

Teraz zajmij się widokiem. Jeśli ViewBag błędzie miał wartość true, to znaczy, że trzeba wyświetlić komunikat o błędzie. Dymek z czerwonym tłem utwórz za pomocą Bootstrapa i klasy alert alert-danger. Dodaj następujący kod pod znacznikiem <h2> do widoku Delete.cshtml:

```
@if (ViewBag.Blad == true)
{
    <div class="alert alert-danger" role="alert">
        Wystąpił błąd podczas usuwania.<br/>
        Spróbuj ponownie.
    </div>
}
```

Dzięki temu po wystąpieniu błędu zostanie wyświetlony komunikat widoczny na rysunku 8.85.

Rysunek 8.85.
Komunikat o błędzie



Ostatnim krokiem (metoda POST) jest wywołanie akcji Delete z parametrem blad = true, jeśli wystąpi wyjątek. W metodzie DeleteConfirmed() i w bloku catch dodaj następującą linię kodu:

```
return RedirectToAction("Delete", new { id = id, blad = true });
```

Ostatecznie metoda DeleteConfirmed() wygląda następująco:

```
// POST: /Ogloszenie/Delete/5
[HttpPost, ActionName("Delete")]
[ValidateAntiForgeryToken]
public ActionResult DeleteConfirmed(int id)
{
    _repo.UsunOgłoszenie(id);
    try
    {
        _repo.SaveChanges();
    }
    catch
    {
        return RedirectToAction("Delete", new
        {
            id = id, blad = true
        });
    }
    return RedirectToAction("Index");
}
```

Aby przetestować działanie dymka z informacją o błędzie, odwiedź adres /Ogloszenie/Delete/3?blad=True.

JavaScript, jQuery i okno potwierdzania wyboru

Kolejnym krokiem będzie dodanie okienka JavaScript pytającego o potwierdzenie usunięcia.

Sposób 1.

Do elementu <input> na stronie z widokiem Delete dodaj metodę onclick:

```
onclick="return confirm('Czy na pewno chcesz usunąć?')"
```

Linijkę:

```
<input type="submit" value="Usuń" class="btn btn-danger" />
```

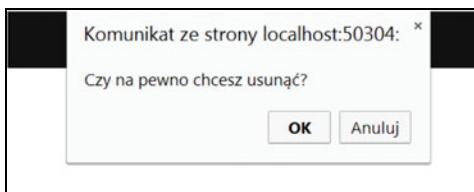
zamień na:

```
<input type="submit" value="Usuń" class="btn btn-danger"
    ↪ onclick="return confirm('Czy na pewno chcesz
    ↪ usunąć?')"/>
```

Po kliknięciu przycisku *Usuń* pokaże się okienko z komunikatem widocznym na rysunku 8.86.

Rysunek 8.86.

*Komunikat
przy usuwaniu*



Sposób 2.

W pliku z szablonem *Layout.cshtml* na samym końcu znajduje się następujący kod:

```
@RenderSection("scripts", required: false)
</body>
</html>
```

Jest to kod odpowiedzialny za wygenerowanie w tym miejscu dokumentu HTML sekcji scripts, jeśli taka sekcja jest zdefiniowana w pliku z widokiem. Required: false mówi o tym, że ta sekcja jest opcjonalna i nie musi być umieszczona na każdej podstronie (w każdym pliku z widokiem).

Wystarczy wkleić na samym dole pliku z widokiem Delete następujący kod korzystający z jQuery:

```
@section Scripts{
    <script type="text/javascript">
        $('.confirmation').on('click', function () {
            return confirm('Czy na pewno chcesz usunąć?');
        });
    </script>
}
```

Fragment kodu (`.on('click', function () {})`) jest odpowiedzialny za oczekiwanie na zdarzenie kliknięcia elementu posiadającego klasę `confirmation` (`$('.confirmation')`). Odpowiedzialny jest on również za wyświetlenie okienka z potwierdzeniem (`return confirm('Czy na pewno chcesz usunąć?');`).

Teraz w przeglądarce koniec wygenerowanego dokumentu HTML będzie wyglądał następująco:

```
<script type="text/javascript">
    $('.confirmation').on('click', function () {
        return confirm('Czy na pewno chcesz usunąć?');
    });
</script>
</body>
</html>
```

Aby wyświetliło się okienko z potwierdzeniem, trzeba jeszcze dodać klasę `confirmation` do elementu `<input>`:

```
<input type="submit" value="Usuń"
       ↗class="btn btn-danger confirmation" />
```

Efekt jest identyczny jak przy wykorzystaniu sposobu numer 1, ale przedstawione zostało wykorzystanie sekcji i kodu JavaScript lub jQuery.

Create

Dla akcji Create odkomentuj dwie akcje z kontrolera `OgłoszenieController`. Pierwsza z nich to akcja dla metody typu GET, która służy do wyświetlenia formularza dodawania ogłoszenia. W wygenerowanym kodzie do obiektu `ViewBag` przekazywana jest lista użytkowników. Należy tę linię kodu usunąć, ponieważ autorem może być tylko jedna osoba i na pewno nie można wybierać, kto ma być autorem ogłoszenia. Autor ogłoszenia oraz data będą automatycznie ustawiane w metodzie POST.

Kod metody `Create` przed usunięciem listy rozwijanej wygląda następująco:

```
// GET: Ogłoszenie/Create
public ActionResult Create()
{
    ViewBag.UzytkownikId = new SelectList(db.Users, "Id", "Email");
    return View();
}
```

A oto kod po usunięciu:

```
// GET: Ogłoszenie/Create
public ActionResult Create()
{
    return View();
}
```

Aktualizacja widoku dla akcji Create

Kolejnym krokiem jest przerobienie widoku Create. Usuń kod odpowiedzialny za pola UzytkownikID oraz DataDodania. Przetłumacz linki, zmień klasę CSS przycisku na btn-success, aby był zielony (listing 8.9, rysunek 8.87) oraz dodaj opisy do Google.

Listing 8.9. Kod widoku po zmianach

```
@model Repozytorium.Models.Ogloszenie

@{
    ViewBag.Tytul = "Dodaj ogłoszenie";
    ViewBag.Opis = "Dodaj ogłoszenie - opis do Google";
    ViewBag.SlowaKluczowe = "Ogłoszenie, dodaj";
}

<h2>Dodaj nowe ogłoszenie</h2>

@using (Html.BeginForm())
{
    @Html.AntiForgeryToken()

    <div class="form-horizontal">
        <h4>Ogłoszenie</h4>
        <hr />
        @Html.ValidationSummary(true, "", 
            new { @class = "text-danger" })

        <div class="form-group">
            @Html.LabelFor(model => model.Tresc, htmlAttributes:
                new { @class = "control-label col-md-2" })
            <div class="col-md-10">
                @Html.EditorFor(model => model.Tresc, new
                    { htmlAttributes = new { @class = "form-control" } })
                @Html.ValidationMessageFor(model => model.Tresc, "", 
                    new { @class = "text-danger" })
            </div>
        </div>

        <div class="form-group">
            @Html.LabelFor(model => model.Tytul, htmlAttributes: new
                { @class = "control-label col-md-2" })
            <div class="col-md-10">
                @Html.EditorFor(model => model.Tytul, new {
                    htmlAttributes = new { @class = "form-control" } })
                @Html.ValidationMessageFor(model => model.Tytul, "", 
                    new { @class = "text-danger" })
            </div>
        </div>

        <div class="form-group">
            <div class="col-md-offset-2 col-md-10">
                <input type="submit" value="Dodaj"
                    class="btn btn-success" />
            </div>
        </div>
    </div>
```

Rysunek 8.87.
Wygląd po zmianach



```
}
```

```
<div>
    @Html.ActionLink("Wróć", "Index", null, new
        { @class = "btn btn-warning" })
</div>
```

```
@section Scripts {
    @Scripts.Render("~/bundles/jqueryval")
}
```

Metoda POST dla akcji Create

Ostatnim krokiem jest implementacja metody POST dla akcji Create. Kod metody przed zmianami wygląda jak poniżej:

```
// POST: Ogloszenie/Create
[HttpPost]
[ValidateAntiForgeryToken]
public ActionResult Create([Bind(Include =
    "Id, Tresc, Tytul, DataDodania, UzytkownikId")] Ogloszenie ogloszenie)
{
    if (ModelState.IsValid)
    {
        db.Ogloszenia.Add(ogloszenie);
        db.SaveChanges();
        return RedirectToAction("Index");
    }

    ViewBag.UzytkownikId = new SelectList(db.Users, "Id", "Email",
        ogloszenie.UzytkownikId);
    return View(ogloszenie);
}
```

A oto lista zmian, które należy wprowadzić (listing 8.10):

- ◆ automatyczne przypisanie Id użytkownika, który dodaje ogłoszenie (aby metoda GetUserId() była dostępna, zainportuj bibliotekę using Microsoft.AspNet.Identity);

```
ogloszenie.UzytkownikId = User.Identity.GetUserId();
```

- ◆ automatyczne przypisanie aktualnej daty jako DataDodania:

```
ogloszenie.DataDodania = DateTime.Now;
```

- ◆ zabezpieczenie metody, by była dostępna tylko dla zalogowanych (jeśli użytkownik jest niezalogowany, nie ma możliwości przypisania id autora do ogłoszenia), poprzez użycie atrybutu:

```
[Authorize]
```

- ◆ usunięcie kodu tworzącego listę użytkowników — usuń linię:

```
ViewBag.UzytkownikId = new SelectList(db.Users, "Id", "Email",
    ↳ogloszenie.UzytkownikId);
```

- ◆ w razie wystąpienia błędu powrót do widoku dodawania:

```
try
{
    _repo.SaveChanges();
    return RedirectToAction("Index");
}
catch
{
    return View(ogloszenie);
}
```

- ◆ w bloku try przed zapisem wywołanie metody z repozytorium o nazwie Dodaj() służącej do dodania użytkownika, która zostanie utworzona w kolejnym kroku:

```
_repo.Dodaj(ogloszenie);
```

- ◆ ustalenie potrzebnych parametrów dzięki Bind[Include=""].

Listing 8.10. Kod po aktualizacji

```
[Authorize]
[HttpPost]
[ValidateAntiForgeryToken]
public ActionResult Create([Bind(Include = "Tresc,Tytul")] Ogloszenie
    ↳ogloszenie)
{
    if (ModelState.IsValid)
    {
        // using Microsoft.AspNet.Identity;
        ogloszenie.UzytkownikId = User.Identity.GetUserId();
        ogloszenie.DataDodania = DateTime.Now;
        try
        {
            _repo.Dodaj(ogloszenie);
            _repo.SaveChanges();
            return RedirectToAction("Index");
        }
        catch
        {
            return View(ogloszenie);
        }
    }
}
```

```

        }
        return View(ogloszenie);
    }
}

```

Zabezpieczenie metody przed atakami CSRF

Aby zabezpieczyć witrynę przed CSRF, wykorzystaj atrybut [ValidateAntiForgeryToken].

W pliku z widokiem znajduje się kod:

```
@Html.AntiForgeryToken()
```

który jest umieszczony w treści formularza:

```

@using (Html.BeginForm())
{
    @Html.AntiForgeryToken()
    ...
}

```

Podczas tworzenia widoku zostaje wygenerowany klucz, który jest przesyłany do przeglądarki w formie ukrytego pola <input type="hidden">:

```

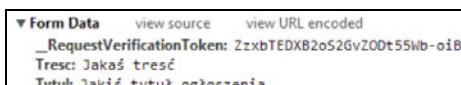
<input name="__RequestVerificationToken" type="hidden"
    value="ZzbTEDXB2oS2GvZ0Dt55Wb-
    ojBVj4WmSz61zhU0k2oLu2fjshQHukQJEs_lWGWeQvnJ8OV_fUFh
    FyzAkreXxbahJp1GTUUq8cf2xW12gTM7eIdpFRT0YdVNJ_prwYqP-
    sqPLw9m70EZ7gi_gwCozyta2" />

```

Następnie przeglądarka odsyła klucz w treści żądania POST (rysunek 8.88), po czym atrybut [ValidateAntiForgeryToken] sprawdza, czy przesłany klucz zgadza się z tym wysłanym podczas generowania widoku. Dla każdego żądania generowany jest inny klucz (token).

Rysunek 8.88.

Klucz w treści żądania



Binding i walidacja żądania POST

Każde żądanie POST (dane z żądania) jest walidowane, po czym poprzez binding (wiązania) sprawdzane są typy i wartości. Przykładowo jeśli zmienna Wiek w modelu danych jest typu int, zaś przesłana zostaje zmienna o nazwie Wiek, a w niej tekst (typu string), wystąpi błąd. Aby sprawdzić, czy przesłane dane pasują do modelu danych, wykorzystywany jest kod:

```
if (ModelState.IsValid)
```

Instrukcja if sprawdza, czy nie było błędów podczas tworzenia obiektu, na podstawie danych z żądania POST.

Binding pozwala również określić, jakie dane z żądania mają zostać zapisane do obiektu z modelem. Klasa ogloszenie zawiera pola takie jak DataDodania i UzytkownikId, które nie powinny być pobierane z przeglądarki, ponieważ mogą stanowić zagrożenie

(np. można oszukać datę dodania lub przypisać ogłoszenie innemu użytkownikowi). Aby określić, które dane będą zapisywane, wykorzystywany jest atrybut [Bind(Include = "")].

Domyślny kod wygląda następująco:

```
public ActionResult Create([Bind(Include =
    "Id,Tresc,Tytul,DataDodania,UzytkownikId")] Ogloszenie ogloszenie)
```

Ponieważ w tym przypadku ważne są tylko Tresc i Tytuł, pozostaw tylko te pola:

```
public ActionResult Create([Bind(Include = "Tresc,Tytul")] Ogloszenie ogloszenie)
```

Teraz nawet gdy zostaną przesłane dodatkowe dane w żądaniu, takie jak np. UzytkownikId, to zostaną zignorowane.

IdUzytkownika i DataDodania są uzupełniane w akcji POST po przesłaniu danych od klienta:

```
ogloszenie.UzytkownikId = User.Identity.GetUserId();
ogloszenie.DataDodania = DateTime.Now;
```

Pole Id dla ogłoszenia jest generowane automatycznie przez bazę danych (zostaje przypisany kolejny numer).

Metoda Create() w repozytorium

Pozostało jeszcze dodać do repozytorium metodę Dodaj służącą do dodawania nowego ogłoszenia.

W interfejsie IOgloszenieRepo dodać deklarację metody:

```
void Dodaj(Ogloszenie ogloszenie);
```

W repozytorium OgloszenieRepo dodać metodę:

```
public void Dodaj(Ogloszenie ogloszenie)
{
    _db.Ogloszenia.Add(ogloszenie);
}
```

Aplikacja jest gotowa do dodawania nowych ogłoszeń. Uruchom ponownie aplikację i przejdź do podstrony *Ogłoszenie*. Po kliknięciu *Dodaj nowe ogłoszenie* uzupełnij pola *Tytuł* i *Treść*, po czym kliknij przycisk *Zapisz/Dodaj/Create* (w zależności od tego, jak go nazwano). Aplikacja przekieruje Cię do strony logowania. Dzieje się tak, ponieważ dla metody POST został użyty atrybut [Authorize]. Aby przekierowanie nastąpiło przed otwarciem strony *Dodaj ogłoszenie*, należy dodać atrybut [Authorize] do metody GET w akcji Create:

```
// GET: Ogloszenie/Create
[Authorize]
public ActionResult Create()
{
    return View();
}
```

Aby dodawać ogłoszenia, należy się zalogować lub założyć nowe konto w aplikacji.

Validacja danych po stronie klienta odbywa się za pomocą biblioteki jQuery, która jest ładowana w sekcji Scripts na końcu pliku z widokiem:

```
@section Scripts {  
    @Scripts.Render("~/bundles/jqueryval")  
}
```

Skrypt jest generowany za pomocą *Binding and Minification*, a więc mechanizmu tworzenia paczek skryptów, który jest konfigurowany w pliku *BundleConfig* znajdującym się w folderze *App_Start*. Paczka *bundles/jqueryval* zawiera bibliotekę *jquery.validate* i jest skonfigurowana za pomocą:

```
bundles.Add(new ScriptBundle("~/bundles/jqueryval")  
    .Include("~/Scripts/jquery.validate*"));
```

Edit

Jako ostatnia zostanie opisana metoda `Edit` pozwalająca na edycję ogłoszenia. Należy odkomentować metody z kontrolera. Początkowy kod akcji wygląda następująco:

```
// GET: Ogloszenie/Edit/5  
public ActionResult Edit(int? id)  
{  
    if (id == null)  
    {  
        return new HttpStatusCodeResult(HttpStatusCode.BadRequest);  
    }  
    Ogloszenie ogloszenie = db.Ogloszenia.Find(id);  
    if (ogloszenie == null)  
    {  
        return HttpNotFound();  
    }  
    ViewBag.UzytkownikId = new SelectList(db.Users, "Id", "Email",  
        ogloszenie.UzytkownikId);  
    return View(ogloszenie);  
}
```

Zamień go na:

```
public ActionResult Edit(int? id)  
{  
    if (id == null)  
    {  
        return new HttpStatusCodeResult(HttpStatusCode.BadRequest);  
    }  
  
    Ogloszenie ogloszenie = _repo.GetOgloszenieById((int)id);  
    if (ogloszenie == null)  
    {  
        return HttpNotFound();  
    }  
    return View(ogloszenie);  
}
```

Kod jest identyczny jak dla akcji Details i Delete. Pobierane są informacje o ogłoszeniu, aby możliwa była ich edycja.

Aktualizacja widoku dla akcji Edit

Ponieważ nie będzie edytowane id użytkownika ani data dodania, zamień następujący kod z wygenerowanego widoku:

```
<div class="form-group">
    @Html.LabelFor(model => model.DataDodania, htmlAttributes: new {
        @class = "control-label col-md-2" })
    <div class="col-md-10">
        @Html.EditorFor(model => model.DataDodania, new {
            htmlAttributes = new { @class = "form-control" } })
        @Html.ValidationMessageFor(model => model.DataDodania, "", 
            new { @class = "text-danger" })
    </div>
</div>

<div class="form-group">
    @Html.LabelFor(model => model.UzytkownikId, "UzytkownikId",
        htmlAttributes: new { @class = "control-label
        col-md-2" })
    <div class="col-md-10">
        @Html.DropDownList("UzytkownikId", null, htmlAttributes:
            new { @class = "form-control" })
        @Html.ValidationMessageFor(model => model.UzytkownikId, "", 
            new { @class = "text-danger" })
    </div>
</div>
```

na pola ukryte:

```
@Html.HiddenFor(m=>m.DataDodania)
@Html.HiddenFor(m => m.UzytkownikId)
```

Walidacja odbywa się tak samo jak w akcji Create:

```
@section Scripts {
    @Scripts.Render("~/bundles/jqueryval")
}
```

Dodaj kolorowe przyciski i opisy do Google. Kod widoku zaprezentowano na listingu 8.11, a wygląd na rysunku 8.89.

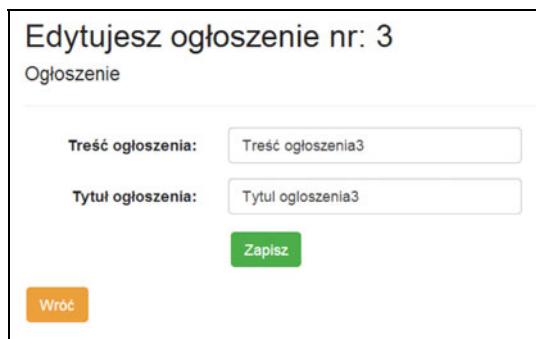
Listing 8.11. Kod widoku po zmianach

```
@model Repository.Models.Ogloszenie

 @{
    ViewBag.Tytul = "Edytujesz ogłoszenie nr: " + @Model.Id;
    ViewBag.Opis = "Edytujesz ogłoszenie nr: " + @Model.Id;
    ViewBag.SlowaKluczowe = "Edytujesz, ogłoszenie, " + @Model.Id;
}

<h2>Edytujesz ogłoszenie nr: @Model.Id</h2>
```

Rysunek 8.89.
Wygląd po zmianach



```
@using (Html.BeginForm())
{
    @Html.AntiForgeryToken()

    <div class="form-horizontal">
        <h4>Ogłoszenie</h4>
        <hr />
        @Html.ValidationSummary(true, "", new { @class = "text-
            danger" })
        @Html.HiddenFor(model => model.Id)

        <div class="form-group">
            @Html.LabelFor(model => model.Tresc, htmlAttributes: new
                { @class = "control-label col-md-2" })
            <div class="col-md-10">
                @Html.EditorFor(model => model.Tresc, new {
                    htmlAttributes = new { @class = "form-control" } })
                @Html.ValidationMessageFor(model => model.Tresc,
                    "", new { @class = "text-danger" })
            </div>
        </div>

        <div class="form-group">
            @Html.LabelFor(model => model.Tytul, htmlAttributes: new
                { @class = "control-label col-md-2" })
            <div class="col-md-10">
                @Html.EditorFor(model => model.Tytul, new {
                    htmlAttributes = new { @class = "form-control" } })
                @Html.ValidationMessageFor(model => model.Tytul,
                    "", new { @class = "text-danger" })
            </div>
        </div>

        @Html.HiddenFor(m=>m.DataDodania)
        @Html.HiddenFor(m => m.UzytkownikId)

        <div class="form-group">
            <div class="col-md-offset-2 col-md-10">
                <input type="submit" value="Zapisz" class="btn
                    btn-success" />
            </div>
        </div>
}
```

```

        </div>
    }

<div>
    @Html.ActionLink("Wróć", "Index", null, new { @class =
        ↳"btn btn-warning" })
</div>

@section Scripts {
    @Scripts.Render("~/bundles/jqueryval")
}

```

Implementacja metody POST

Teraz przerobisz metodę POST, aby korzystała z repozytorium, oraz zaimplementujesz obsługę błędów.

Kod metody przed zmianami wygląda następująco:

```

[HttpPost]
[ValidateAntiForgeryToken]
public ActionResult Edit([Bind(Include = "Id,Tresc,Tytul,DataDodania,
    ↳UzytkownikId")] Ogloszenie ogloszenie)
{
    if (ModelState.IsValid)
    {
        db.Entry(ogloszenie).State = EntityState.Modified;
        db.SaveChanges();
        return RedirectToAction("Index");
    }
    ViewBag.UzytkownikId = new SelectList(db.Users, "Id", "Email",
        ↳ogloszenie.UzytkownikId);
    return View(ogloszenie);
}

```

Wywołaj metodę Aktualizuj() na rzecz obiektu repozytorium i usuń ViewBag z użytkownikami. Następnie należałoby dodać atrybut [Authorize], aby zablokować dostęp do edycji niezalogowanym użytkownikom, jednak nie dodajemy go, ponieważ chcemy przetestować działanie akcji Edit.

Kod metody po zmianie prezentuje się jak poniżej:

```

[HttpPost]
[ValidateAntiForgeryToken]
public ActionResult Edit([Bind(Include = "Id,Tresc,Tytul,DataDodania,
    ↳UzytkownikId")] Ogloszenie ogloszenie)
{
    if (ModelState.IsValid)
    {
        try
        {
            _repo.Aktualizuj(ogloszenie);
            _repo.SaveChanges();
        }
        catch
    }
}

```

```
        {
            return View(ogloszenie);
        }
    }
    return View(ogloszenie);
}
```

Obsługa błędów dla akcji Edit

W metodzie Delete zaimplementowano obsługę błędów poprzez przekazanie opcjonalnego parametru bool w adresie URL, a następnie przekazywano go do widoku za pomocą ViewBag. Tym razem widok będzie odświeżany przy użyciu żądania POST, bez wywoływania akcji GET z parametrem w adresie URL.

Aby to zrobić, dodaj ViewBag o nazwie Blad przed zwróceniem widoku. W bloku catch będzie miał wartość true, ponieważ wystąpił błąd, natomiast na końcu metody będzie miał wartość false, ponieważ wszystko było dobrze. Dodaj również zakomentowany kod (jedną linię), który pozwoli na symulację wystąpienia błędu. Przypisz do pola UzytkownikaId wartość, której nie ma w bazie. Podczas gdy linia kodu jest zakomentowana, akcja zakończy się powodzeniem, a gdy odkomentuje się kod, akcja zakończy się błędem.

Kod metody po zmianach wygląda następująco:

```
[HttpPost]
[ValidateAntiForgeryToken]
public ActionResult Edit([Bind(Include = "Id,Tresc,Tytul,DataDodania,
    ↳UzytkownikaId")] Ogloszenie ogloszenie)
{
    if (ModelState.IsValid)
    {
        try
        {
            // ogloszenie.UzytkownikaId = "fdfgd";
            _repo.Aktualizuj(ogloszenie);
            _repo.SaveChanges();
        }
        catch
        {
            ViewBag.Blad = true;
            return View(ogloszenie);
        }
    }
    ViewBag.Blad = false;
    return View(ogloszenie);
}
```

Kolejnym krokiem jest aktualizacja pliku z widokiem, aby wyświetlał informacje o błędzie lub o pomyślnej aktualizacji.

Wpisz następujący kod pod nagłówkiem <h2> w widoku Edit:

```
<h2>Edytujesz ogłoszenie nr: @Model.Id</h2>
```

```
@if (ViewBag.Blad == true)
```

```

    {
        <div class="alert alert-danger" role="alert">
            Wystąpił błąd podczas edycji.<br />
            Spróbuj ponownie.
        </div>
    }
    else if (ViewBag.Blad == false)
    {
        <div class="alert alert-success" role="alert">
            Pomyślnie edytowano.
            Twoje ogłoszenie wygląda teraz następująco:
        </div>
    }
}

```

Wygląd po poprawnej aktualizacji zaprezentowano na rysunku 8.90, a po wystąpieniu błędów na rysunku 8.91.

Rysunek 8.90.

Wygląd dla pomyślnej aktualizacji

Edytujesz ogłoszenie nr: 2

Pomyślnie edytowano. Twoje ogłoszenie wygląda teraz następująco:

Ogłoszenie

Treść ogłoszenia: Treść ogłoszenia233

Tytuł ogłoszenia: Tytuł ogłoszenia2

Zapisz

Wróć

Rysunek 8.91.

Wygląd, gdy wystąpi błąd

Edytujesz ogłoszenie nr: 2

Wystąpił błąd podczas edycji.
Spróbuj ponownie.

Ogłoszenie

Treść ogłoszenia: Treść ogłoszenia233

Tytuł ogłoszenia: Tytuł ogłoszenia2

Zapisz

Wróć

Metoda Aktualizuj() w repozytorium

Pozostało jeszcze dodać do repozytorium metodę `Aktualizuj()`, która służy do wprowadzania zmian w ogłoszeniu.

W interfejsie `IOgloszenieRepo` dodaj deklarację metody:

```
void Aktualizuj(Ogloszenie ogloszenie);
```

W repozytorium `OgloszenieRepo` dodaj metodę:

```
public void Aktualizuj(Ogloszenie ogloszenie)
{
    _db.Entry(ogloszenie).State = EntityState.Modified;
}
```

W metodzie trzeba poinformować kontekst, że dane zostały zmienione, a gdy zostanie wywołana metoda `SaveChanges()`, należy zaktualizować dane w bazie danych. Aby możliwe było korzystanie z właściwości `Entry`, konieczne jest dodanie następującej linii w interfejsie `IOglContext`:

```
DbEntityEntry Entry(object entity);
```

Importujemy biblioteki i aplikacja jest gotowa do edycji ogłoszeń.

Etap 4. Krok 2. Aktualizacja szablonu `_Layout.cshtml`

Kolejnym krokiem jest edycja szablonu strony. Utworzymy rozwijane menu i dodamy linki do poszczególnych akcji z kontrolera `OgloszenieController`. W pliku `_Layout.cshtml` znajdującym się w folderze `Views/Shared` następujący kod:

```
<ul class="nav navbar-nav">
    <li>@Html.ActionLink("Home", "Index", "Home")</li>
    <li>@Html.ActionLink("About", "About", "Home")</li>
    <li>@Html.ActionLink("Contact", "Contact", "Home")</li>
</ul>
```

zamień na:

```
<ul class="nav navbar-nav">
    <li>@Html.ActionLink("Home", "Index", "Home")</li>

    <li class="dropdown">
        <a href="#" class="dropdown-toggle" data-toggle=
            ↪"dropdown">Ogłoszenia <span class="caret"></span></a>
        <ul class="dropdown-menu" role="menu">
            <li>@Html.ActionLink("Lista ogłoszeń", "Index",
                ↪"Ogloszenie")</li>
            @if (User.Identity.IsAuthenticated)
            {
                <li>@Html.ActionLink("Dodaj ogłoszenie", "Create",
                    ↪"Ogloszenie")</li>
            }
            <li class="divider"></li>
            <li>@Html.ActionLink("Lista jako PartialView", "Partial",
                ↪"Ogloszenie")</li>
        </ul>
    </li>
</ul>
```

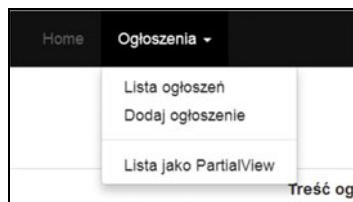
Usunąłeś linki do akcji About i Contact z kontrolera HomeController. Można teraz usunąć także pliki o nazwach *About* i *Contact* z folderu *Views/Home*.

Dodałeś linki *Lista ogłoszeń* i *Dodaj ogłoszenie*. Oprócz tych dwóch linków dodałeś również link do listy ogłoszeń zwróconej jako PartialView, czyli bez kodu z szablonem strony generowanym na podstawie pliku *_Layout.cshtml*. Link *Dodaj ogłoszenie* jest dostępny tylko dla zalogowanych użytkowników.

Do utworzenia menu rozwijanego posłużyły gotowe komponenty Bootstrapa (rysunek 8.92).

Rysunek 8.92.

Wygląd menu po zmianach



Etap 4. Krok 3. Widoki częściowe — PartialViews

Utworzysz teraz widok PartialView. Dodaj nową akcję do kontrolera OgloszenieController o nazwie Partial. Podobnie jak w akcji Index, pobierz dane z repozytorium za pomocą metody PobierzOgłoszenia(). Jedyną różnicą będzie zwracany typ — zamiast View będzie to PartialView.

Kod akcji Partial:

```
// GET: /Ogłoszenie/
public ActionResult Partial()
{
    var ogłoszenia = _repo.PobierzOgłoszenia();
    return PartialView("Index", ogłoszenia);
}
```

Wykorzystałeś ten sam plik z widokiem, a więc plik o nazwie Index. W poprzednich akcjach nazwa widoku była taka sama jak nazwa akcji, dlatego nie trzeba było podawać nazwy widoku jako pierwszego parametru w metodzie return View().

Gdyby został utworzony osobny plik z widokiem o nazwie *Partial* w folderze *Views/Ogłoszenie*, wystarczyłoby zwrócić:

```
return PartialView(ogłoszenia);
```

Poniżej porównano wygląd zwykłej strony, czyli razem z layoutem (rysunek 8.93), z wyglądem strony *Partial* (rysunek 8.94).

Application name Home Ogłoszenia -				Hello hmfghmf@fdg.pl! Log off
Lista Ogłoszeń				
Dodaj nowe ogłoszenie				
UzytkownikaId	Treść ogłoszenia:	Tytuł ogłoszenia:	Data dodania:	
2fc7102a-d3ef-4244-a46c-05ee0cc4aaa4	Treść ogłoszenia233	Tytuł ogłoszenia2	2014-06-29	Szczegóły Edytuj Usuń
2fc7102a-d3ef-4244-a46c-05ee0cc4aaa4	Treść ogłoszenia3	Tytuł ogłoszenia3	2014-06-28	Szczegóły Edytuj Usuń
2fc7102a-d3ef-4244-a46c-05ee0cc4aaa4	Treść ogłoszenia4	Tytuł ogłoszenia4	2014-06-27	Szczegóły Edytuj Usuń

Rysunek 8.93. Zwykła strona z szablonem (layoutem)

Lista Ogłoszeń				
Dodaj nowe ogłoszenie				
UzytkownikaId	Treść ogłoszenia:	Tytuł ogłoszenia:	Data dodania:	
2fc7102a-d3ef-4244-a46c-05ee0cc4aaa4	Treść ogłoszenia233	Tytuł ogłoszenia2	2014-06-29	Szczegóły Edytuj Usuń
2fc7102a-d3ef-4244-a46c-05ee0cc4aaa4	Treść ogłoszenia3	Tytuł ogłoszenia3	2014-06-28	Szczegóły Edytuj Usuń
2fc7102a-d3ef-4244-a46c-05ee0cc4aaa4	Treść ogłoszenia4	Tytuł ogłoszenia4	2014-06-27	Szczegóły Edytuj Usuń
2fc7102a-d3ef-4244-a46c-05ee0cc4aaa4	Treść ogłoszenia5	Tytuł ogłoszenia5	2014-06-26	Szczegóły Edytuj Usuń
2fc7102a-d3ef-4244-a46c-05ee0cc4aaa4	Treść ogłoszenia6	Tytuł ogłoszenia6	2014-06-25	Szczegóły Edytuj Usuń
2fc7102a-d3ef-4244-a46c-05ee0cc4aaa4	Treść ogłoszenia7	Tytuł ogłoszenia7	2014-06-24	Szczegóły Edytuj Usuń

Rysunek 8.94. Strona z widokiem Partial

Zwykła strona jest „wstawiana” w miejscu, w którym znajduje się kod @RenderBody() w pliku z layoutem. Razem z layoutem wczytywane są style CSS, dlatego strona *Partial* nie ma tego samego wyglądu. Nagłówek razem z menu znajduje się w szablonie, więc w widoku *Partial* go nie ma.

Kod HTML widoku *Partial* wygląda następująco:

```
<h2>Lista ogłoszeń</h2>

<p>
    <a class="btn btn-primary" href="/Ogłoszenie/Dodaj">
        Dodaj nowe ogłoszenie</a>
    </p>
    <table class="table">
        <tr>
            <th>
                UzytkownikId
            </th>
            <th>
                Treść ogłoszenia:
            </th>
            <th>
                Tytuł ogłoszenia:
            </th>
            <th>
                Data dodania:
            </th>
            <th></th>
        </tr>
        <tr>
            <td>
                2fc7102a-d3ef-4244-a46c-05ee0cc4aaa4
            </td>
            <td>
                Treść ogłoszenia2 - edytowane
            </td>
            <td>
                Tytuł ogłoszenia2
            </td>
            <td>
                2014-06-29
            </td>
            <td>
                <a class="btn btn-warning" href="/
                    ↳Ogłoszenie/Details/2">Szczegóły</a>
                <br />
                <a class="btn btn-primary" href="/
                    ↳Ogłoszenie/Edit/2">Edytuj</a>
                <br />
                <a class="btn btn-danger" href="/
                    ↳Ogłoszenie/Delete/2">Usuń</a>
            </td>
        </tr>
    </table>
```

Etap 4. Podsumowanie

Na tym etapie istnieje aplikacja, która ma zaimplementowane wszystkie typy akcji CRUD dla klasy ogłoszeń. Potrafi zwracać widoki *Partial* i wyświetla użytkownikom informacje o tym, czy operacja się powiodła, czy nie. Potrafi zabezpieczyć aplikację przed atakami CSRF dzięki atrybutowi [ValidateAntiForgeryToken]. Użytkownik wie, jak działa binding oraz jak sprawdzić, czy dane przekazywane w akcji POST są prawidłowego typu. Potrafi również określić, jakie pola chce wykorzystać z modelu przesłanego w żądaniu POST. Wykorzystaliśmy także framework Bootstrap do poprawy wyglądu aplikacji oraz skorzystaliśmy z JavaScriptu i jQuery do wyświetlenia okienka potwierdzania podczas usuwania.

Etap 5. Bezpieczeństwo, uwierzytelnianie i autoryzacja dostępu

Ważnym elementem jest bezpieczeństwo aplikacji. Obecnie każdy zalogowany użytkownik może dodać, usunąć lub edytować ogłoszenie. Aplikacja zostanie tak zabezpieczona, aby tylko właściciel mógł edytować i usuwać swoje ogłoszenia. Dodamy widok, w którym będą wyświetlane tylko ogłoszenia aktualnie zalogowanego użytkownika. Ukryjemy i zablokujemy opcje *Usuń* i *Edytuj* dla niezalogowanych użytkowników na liście ogłoszeń. Rozpoczniemy od uwierzytelniania i autoryzacji dostępu za pomocą ról.

Uwierzytelnianie i logowanie przez portale

Na początek będzie to uwierzytelnianie, czyli założenie konta na portalu.

Rejestracja nowego użytkownika

Aby założyć konto na portalu, kliknij w prawym rogu *Register*, co powoduje wyświetlenie strony rejestracji (rysunek 8.95).

Rysunek 8.95.

Rejestracja użytkownika

The screenshot shows a registration form titled "Register. Create a new account." It features three input fields: "Email", "Password", and "Confirm password", each with a corresponding text input box. Below these fields is a "Register" button.

Wpisz e-mail i hasło (dwukrotnie), po czym kliknij przycisk *Register* — konto zostało założone.

Uwierzytelnianie za pomocą Google

Możliwe jest również założenie konta na portalu na podstawie danych przesłanych z innego portalu. Wykorzystamy portal Google, ponieważ na Facebooku trzeba tworzyć aplikację. Google pozwala na uwierzytelnianie bez zakładania aplikacji. Aby włączyć logowanie przez Google, przejdź do pliku *Startup.Auth* w folderze *App_Start*. Znajduje się w nim kod:

```
// Uncomment the following lines to enable logging in with third party login providers
// app.UseMicrosoftAccountAuthentication(
//   clientId: "",
//   clientSecret: "");

// app.UseTwitterAuthentication(
//   consumerKey: "",
//   consumerSecret: "");

// app.UseFacebookAuthentication(
//   appId: "",
//   appSecret: "");

// app.UseGoogleAuthentication(new GoogleOAuth2AuthenticationOptions()
// {
//   ClientId = "",
//   ClientSecret = ""
//});
```

Aby włączyć uwierzytelnianie za pomocą wybranego portalu, należy odkomentować odpowiednie linie. Ponieważ nie zakładaliśmy aplikacji na Facebooku ani Google, nie posiadamy Id ani hasła potrzebnego do włączenia uwierzytelniania za pomocą zewnętrznych serwisów. Aby wykonać kolejne kroki, należy utworzyć aplikację Google i pobrać odpowiednie dane lub skorzystać z logowania poprzez Google bez podawania hasła. Miedzy zakomentowany kod wklej linijkę:

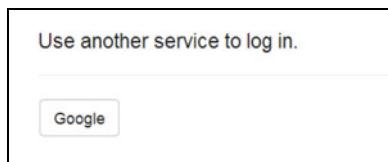
```
app.UseGoogleAuthentication();
```

Teraz można się już zalogować poprzez e-maila Google na hoście lokalnym.

Uruchom ponownie aplikację i kliknij *Log in* w prawym górnym rogu. Wyświetli się strona do logowania, a po prawej stronie link do logowania poprzez Google (rysunek 8.96).

Rysunek 8.96.

Logowanie z portalu
Google

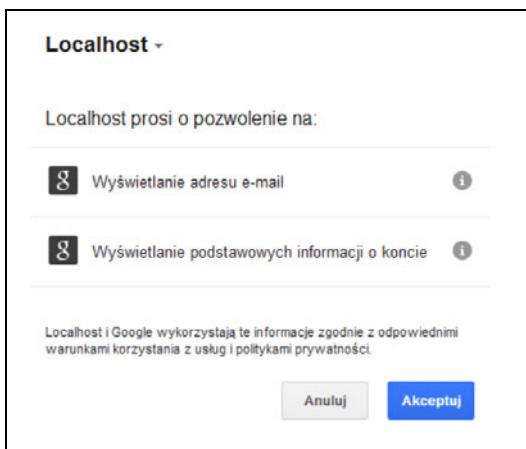


Jeśli klikniesz przycisk *Google*, kiedy nie jesteś zalogowany w Google, zostaniesz przekierowany na stronę logowania. Po zalogowaniu będzie trzeba wyrazić zgodę na wykorzystanie informacji (rysunek 8.97).

Jeśli w tym punkcie wystąpi błąd zawierający słowa *unregistered domain*, to niezbędne będzie odkomentowanie kodu:

Rysunek 8.97.

Zgoda na wykorzystanie informacji z konta Google



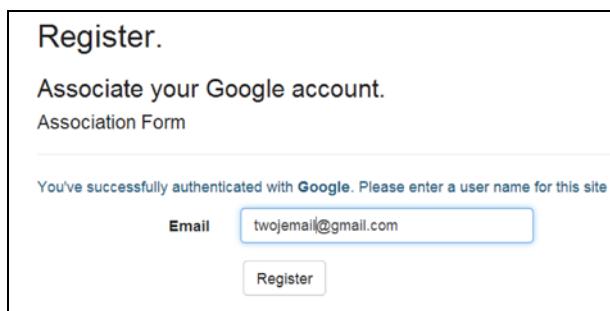
```
// app.UseGoogleAuthentication(new GoogleOAuth2AuthenticationOptions()
//{
//    ClientId = "",
//    ClientSecret = ""
//});
```

i wpisanie poprawnych danych, które możemy uzyskać, tworząc konto Google Apps.

Po zaakceptowaniu zostaniesz przekierowany do portalu i automatycznie zostanie pobrany adres e-mail (rysunek 8.98).

Rysunek 8.98.

Rejestracja z użyciem Google

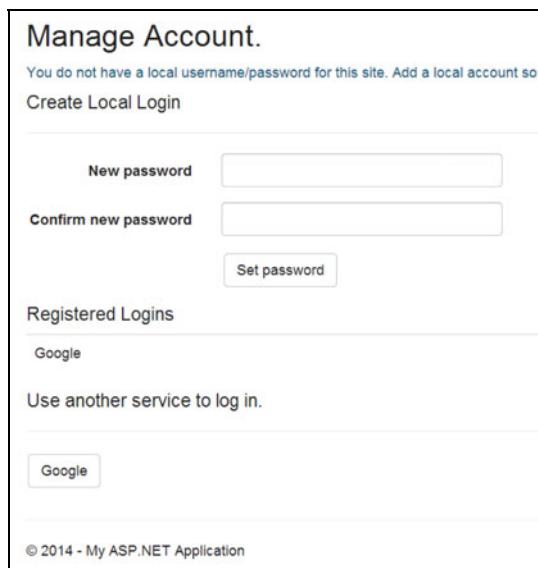


Po kliknięciu *Register* konto na portalu zostanie powiązane z e-mailem Google. Przy kolejnym logowaniu za pomocą Google od razu po pierwszym kliknięciu zostaniesz zalogowany. Logowanie za pomocą Facebooka itp. przebiega niemal identycznie, dlatego nie będziemy tego tutaj omawiać. Jeśli przejdziesz do swojego konta (kliknij nazwę użytkownika po prawej u góry), będziesz mógł dodać hasło do konta (rysunek 8.99), dzięki czemu będziesz się logował bez użycia Google.

Cała logika logowania znajduje się w *AccountController*. Pliki z klasami *ViewModel* wykorzystywane do logowania przeniesiono do *Repozytorium/Models/View*.

W ASP.NET MVC 5.2 i ASP.NET Identity 2.1, które zostały wydane razem z Visual Studio 2013 Update 3, dodany został *Manage Controller*, który pozwala na dwuetapowe uwierzytelnianie za pomocą kodu jednorazowego przy użyciu wiadomości SMS.

Rysunek 8.99.
Dodawanie hasła
do konta



Zaimplementowany został również mechanizm blokowania konta na pewien czas po kilkakrotnym nieprawidłowym wprowadzeniu hasła. Poniżej zamieszczono kod znajdujący się w pliku *IdentityConfig*, odpowiedzialny za ustawienia dwuetapowego uwierzytelniania i blokowania konta:

```
manager.UserLockoutEnabledByDefault = true;
manager.DefaultAccountLockoutTimeSpan = TimeSpan.FromMinutes(5);
manager.MaxFailedAccessAttemptsBeforeLockout = 5;
```

Autoryzacja — role

Autoryzacja polega na określeniu, czy dany użytkownik ma dostęp do wybranego zasobu bądź podstrony. W aplikacji w metodzie *Seed* uruchamianej podczas migracji utworzono użytkownika Admin i rolę Administrator.

Teraz dodamy rolę Pracownik. Dodaj następujący kod w metodzie *SeedRoles()*:

```
if (!roleManager.RoleExists("Pracownik"))
{
    var role = new Microsoft.AspNet.Identity.EntityFramework.IdentityRole();
    role.Name = "Pracownik";
    roleManager.Create(role);
}
```

Możliwe jest dodanie nowego użytkownika, takiego jak np. Marek, który będzie przypisany do roli Pracownik i będzie posiadał e-mail *marek@AspNetMvc.pl*, oraz np. użytkownika Prezes, który będzie przypisany do roli Admin i będzie posiadał e-mail *prezes@AspNetMvc.pl*.

Dodaj kod do metody *SeedUsers()*:

```
if (!context.Users.Any(u => u.UserName == "Marek"))
{
    var user = new Uzytkownik { UserName = "marek@AspNetMvc.pl" };
    var adminresult = manager.Create(user, "1234Abc.");
    if (adminresult.Succeeded)
        manager.AddToRole(user.Id, "Pracownik");
}

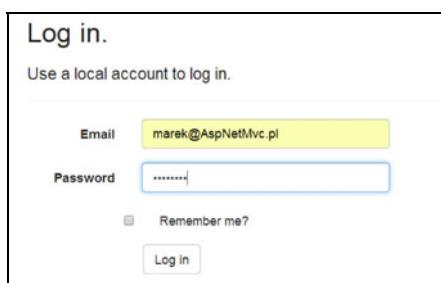
if (!context.Users.Any(u => u.UserName == "Prezes"))
{
    var user = new Uzytkownik { UserName = "prezes@AspNetMvc.pl" };
    var adminresult = manager.Create(user, "1234Abc.");
    if (adminresult.Succeeded)
        manager.AddToRole(user.Id, "Admin");
}
```

Następnie uruchom migrację komendą `Update-Database`. Jeśli wystąpią błędy, to można usunąć tabele z bazy danych i pliki z migracjami (pozostaw plik konfiguracyjny), a następnie uruchomić komendy `Add-migration` i `Update-Database`. Zostanie utworzona nowa baza danych, a metoda `Seed` doda dane startowe.

Pole `User Name` musi być e-mailem, inaczej walidacja po stronie klienta nie pozwoli się zalogować.

Po dodaniu nowych użytkowników i aktualizacji bazy danych można się zalogować na konto użytkownika Marek lub Prezes (rysunek 8.100).

Rysunek 8.100.
Logowanie na konto dowolnego użytkownika



Utworzenie tych dwóch kont było konieczne, aby zademonstrować autoryzację dostępu do poszczególnych akcji w zależności od roli użytkownika.

Zabezpieczanie akcji

Ustanowiono następujące ogólne zasady:

- ◆ admin (rola Admin, e-mail `prezes@AspNetMvc.pl`) może wszystko (Szczegóły, Edytuj i Usuń);
- ◆ pracownik (rola Pracownik, e-mail `Marek@AspNetMvc.pl`) może tylko edytować i dodawać, nie może usuwać ogłoszeń;
- ◆ zwykły użytkownik nieprzypisany do żadnej roli może wszystko (Szczegóły, Edytuj i Usuń), ale tylko dla ogłoszeń dodanych przez siebie;

- ◆ Admin i Pracownik mogą operować na wszystkich ogłoszeniach bez względu na autora.

Index

W widoku do akcji Index dla konta Admin będą widoczne wszystkie linki do akcji Szczegóły, Edytuj i Usuń. Istnieje także możliwość dodawania ogłoszenia. Dla konta Pracownik link do usuwania nie będzie widoczny. Natomiast zwykły użytkownik będzie miał dostępny tylko przycisk Szczegóły.

W pliku *Index.cshtml* znajduje się kod:

```
@Html.ActionLink("Szczegóły", "Details", new { id = item.Id }, new {
    @class = "btn btn-warning" })
<br />
@Html.ActionLink("Edytuj ", "Edit", new { id = item.Id }, new {
    @class = "btn btn-primary" })
<br />
@Html.ActionLink("Usuń", "Delete", new { id = item.Id }, new {
    @class = "btn btn-danger" })
```

Zamień go na:

```
@Html.ActionLink("Szczegóły", "Details", new { id = item.Id }, new {
    @class = "btn btn-warning" })
@if (User.Identity.IsAuthenticated && (User.IsInRole("Admin") ||
    User.IsInRole("Pracownik")))
{
    <br />
    @Html.ActionLink("Edytuj ", "Edit", new { id = item.Id },
        new { @class = "btn btn-primary" })
    if (User.IsInRole("Admin"))
    {
        <br />
        @Html.ActionLink("Usuń", "Delete", new { id = item.Id }, new
            { @class = "btn btn-danger" })
    }
}
```

Następnie kod:

```
<p>
    @Html.ActionLink("Dodaj nowe ogłoszenie", "Create", null, new {
        @class = "btn btn-primary" })
</p>
```

zamień na:

```
@if (User.Identity.IsAuthenticated)
{
    <p>
        @Html.ActionLink("Dodaj nowe ogłoszenie", "Create", null,
            new { @class = "btn btn-primary" })
    </p>
}
```

W ten sposób spełnione zostały założenia. Na kolejnych rysunkach zaprezentowano poszczególne widoki dla użytkowników: Admin (rysunek 8.101), Pracownik (rysunek 8.102), tylko zalogowanych (rysunek 8.103) oraz niezalogowanych (rysunek 8.104).

Na ten moment jedynie usuneliśmy linki ze strony. Jeśli ktoś wpisze adres w wyszukiwarkę, uzyska dostęp do edycji lub usuwania, ponieważ akcje nie zostały jeszcze zabezpieczone w kontrolerze.

Lista Ogłoszeń

[Dodaj nowe ogłoszenie](#)

UzytkownikaId	Treść ogłoszenia:	Tytuł ogłoszenia:	Data dodania:	
91825236-fdd2-49fa-82d1-ca6f4d30a225	Treść ogłoszenia1	Tytuł ogłoszenia1	2014-06-30	Szczegóły Edytuj Usuń
91825236-fdd2-49fa-82d1-ca6f4d30a225	Treść ogłoszenia2	Tytuł ogłoszenia2	2014-06-29	Szczegóły Edytuj Usuń

Rysunek 8.101. Widok dla konta Admin

Lista Ogłoszeń

[Dodaj nowe ogłoszenie](#)

UzytkownikaId	Treść ogłoszenia:	Tytuł ogłoszenia:	Data dodania:	
91825236-fdd2-49fa-82d1-ca6f4d30a225	Treść ogłoszenia1	Tytuł ogłoszenia1	2014-06-30	Szczegóły Edytuj
91825236-fdd2-49fa-82d1-ca6f4d30a225	Treść ogłoszenia2	Tytuł ogłoszenia2	2014-06-29	Szczegóły Edytuj

Rysunek 8.102. Widok dla konta Pracownik

Lista Ogłoszeń

[Dodaj nowe ogłoszenie](#)

UzytkownikaId	Treść ogłoszenia:	Tytuł ogłoszenia:	Data dodania:	
91825236-fdd2-49fa-82d1-ca6f4d30a225	Treść ogłoszenia1	Tytuł ogłoszenia1	2014-06-30	Szczegóły
91825236-fdd2-49fa-82d1-ca6f4d30a225	Treść ogłoszenia2	Tytuł ogłoszenia2	2014-06-29	Szczegóły
91825236-fdd2-49fa-82d1-ca6f4d30a225	Treść ogłoszenia3	Tytuł ogłoszenia3	2014-06-28	Szczegóły

Rysunek 8.103. Widok dla zalogowanego użytkownika

Lista Ogłoszeń				
UzytkownikaId	Treść ogłoszenia:	Tytuł ogłoszenia:	Data dodania:	
91825236-fdd2-49fa-82d1-ca6f4d30a225	Treść ogłoszenia1	Tytuł ogłoszenia1	2014-06-30	Szczegóły
91825236-fdd2-49fa-82d1-ca6f4d30a225	Treść ogłoszenia2	Tytuł ogłoszenia2	2014-06-29	Szczegóły
91825236-fdd2-49fa-82d1-ca6f4d30a225	Treść ogłoszenia3	Tytuł ogłoszenia3	2014-06-28	Szczegóły

Rysunek 8.104. Widok dla niezalogowanych użytkowników

Details

W akcji Details znajdują się tylko przyciski *Wróć* i *Edytuj*. Należy ukryć przycisk *Edytuj* dla niezalogowanych użytkowników i tych, którzy są zalogowani, ale nie są autorami ogłoszenia. Jednocześnie musi on być widoczny dla zalogowanych użytkowników, którzy nie są autorami, aby zaprezentować zabezpieczenie po stronie kontrolera.

W pliku z widokiem Details zamień następujący kod:

```
<p>
    @Html.ActionLink("Edytuj", "Edit", new { id = Model.Id }, new {
        @class = "btn btn-primary" }) |
    @Html.ActionLink("Wróć", "Index", null, new { @class =
        "btn btn-warning" })
</p>
```

na:

```
<p>
    @if (User.Identity.IsAuthenticated || User.IsInRole("Admin") ||
        User.IsInRole("Pracownik"))
    {
        @Html.ActionLink("Edytuj", "Edit", new { id = Model.Id },
            new { @class = "btn btn-primary" })
        @: |
    }
    @Html.ActionLink("Wróć", "Index", null, new { @class =
        "btn btn-warning" })
</p>
```

Teraz przycisk jest widoczny dla wszystkich (rysunek 8.105) oprócz tych, którzy nie są zalogowani (rysunek 8.106).

Rysunek 8.105.
Widok
dla zalogowanych
użytkowników

Szczegóły ogłoszenia nr: 1

Ogłoszenie

Id autora:	91825236-fdd2-49fa-82d1-ca6f4d30a225
Treść ogłoszenia:	Treść ogłoszenia1
Tytuł ogłoszenia:	Tytuł ogłoszenia1
Data dodania:	2014-06-30

[Wróć](#) | [Edytuj](#)

Rysunek 8.106.

Widok
dla niezalogowanych
użytkowników

Szczegóły ogłoszenia nr: 1

Ogłoszenie

Id autora: 91825236-fdd2-49fa-82d1-ca6f4d30a225
Treść ogłoszenia: Treść ogłoszenia1
Tytuł ogłoszenia: Tytuł ogłoszenia1
Data dodania: 2014-06-30

[Wróć](#)

Edit

Kolejnym elementem jest akcja Edit. Mimo że przyciski są niewidoczne, to w dalszym ciągu niezalogowana osoba może edytować wartość.

Przejdz pod adres (jako niezalogowany użytkownik) /Ogłoszenie/Edit/1. Mimo że klient jest niezalogowany, pokazuje się strona edycji (rysunek 8.107).

Rysunek 8.107.

Strona edycji

Edytujesz ogłoszenie nr: 1

Ogłoszenie

Treść ogłoszenia:

Tytuł ogłoszenia:

[Zapisz](#)

[Wróć](#)

Aby zabezpieczyć aplikację, musisz zaktualizować metody GET i POST dla akcji Edit. Dodaj atrybut [Authorize] do metod GET i POST dla akcji Edit, aby nie było możliwości wyświetlenia strony edycji dla niezalogowanych użytkowników. Po dodaniu atrybutu metoda GET wygląda następująco:

```
[Authorize]
public ActionResult Edit(int id)
{
    Ogloszenie ogloszenie = _repo.GetOgloszenieById(id);
    if (ogloszenie == null)
    {
        return HttpNotFound();
    }
    return View(ogloszenie);
}
```

Teraz od razu po wpisaniu adresu (jako niezalogowany użytkownik) /Ogłoszenie/Edit/1 następuje przekierowanie do akcji logowania.

Analogicznie dodaj atrybut [Authorize] do metody POST.

Załóż teraz konto zwykłego użytkownika niemającego przypisanej żadnej roli (może to być konto powiązane z Google). Na stronie szczegółów dostępny jest link do edycji. Wybierz dowolne ogłoszenie i edytuj wartość, po czym zapisz. Pokaż się informacja o pomyślanej edycji (rysunek 8.108).

Rysunek 8.108.

Pomyślnie
zakończona edycja
ogłoszenia

Teraz można edytować dowolne ogłoszenie, nawet nie swoje.

Aby zabezpieczyć metodę, trzeba sprawdzić, czy Id zalogowanego użytkownika jest takie samo jak IdUżytkownika zapisane w ogłoszeniu. Po pobraniu użytkownika z bazy danych i sprawdzeniu, czy nie została zwrócona wartość null, dodaj warunek sprawdzający:

```
else if (ogloszenie.UzytkownikId != User.Identity.GetUserId() &&
        !(User.IsInRole("Admin") || User.IsInRole("Pracownik")))
{
    return new HttpStatusCodeResult(HttpStatusCode.BadRequest);
}
```

Metoda po aktualizacji wygląda następująco:

```
[Authorize]
public ActionResult Edit(int? id)
{
    if (id == null)
    {
        return new HttpStatusCodeResult(HttpStatusCode.BadRequest);
    }
    Ogloszenie ogloszenie = _repo.GetOgloszenieById((int)id);
    if (ogloszenie == null)
    {
        return HttpNotFound();
    }
    else if (ogloszenie.UzytkownikId != User.Identity.GetUserId() &&
              !(User.IsInRole("Admin") || User.IsInRole("Pracownik")))
    {
        return new HttpStatusCodeResult(HttpStatusCode.BadRequest);
    }
    return View(ogloszenie);
}
```

Oprócz sprawdzenia Id został również dodany kod, który pozwoli adminowi i pracownikom edytować dowolne ogłoszenia. Widok zwrócony użytkownikowi, który nie jest właścicielem, a będzie próbował edytować nie swoje ogłoszenie, przedstawia rysunek 8.109.

Rysunek 8.109.
Błąd podczas edycji



Akcja Edit jest już zabezpieczona, ale w dalszym ciągu jeśli ktoś wyśle poprawne żądanie POST, będzie mógł edytować ogłoszenie. Musiałby jednak przechwycić jednorazowy token (AntiForgeryToken) i ciasneczkę przesyłane do przeglądarki po wykonaniu metody GET. Jest to możliwe, jednak dość trudne do zrobienia. Aby zabezpieczyć się przed tego typu atakami, stosowany jest protokół SSL/TSL, który nie wymaga dużych zmian w aplikacji — należy go tylko uruchomić.

Create

Akcja Create została już wcześniej zabezpieczona poprzez binding ([Bind(Include=" ")]) , atrybut [Authorize] oraz AntiForgeryToken. Jest ona dostępna dla wszystkich użytkowników oprócz tych, którzy nie są zalogowani, dlatego nie ma specjalnych wymogów co do bezpieczeństwa.

Delete

Jako ostatnia zostanie zabezpieczona akcja Delete. Usuwać ogłoszenia może tylko Admin lub właściciel. Sytuacja wygląda bardzo podobnie jak z Edytuj, jednak Pracownik nie może usuwać (edytować mógł).

Przejdź do metody GET dla akcji Delete i dodaj atrybut [Authorize]. Kolejnym krokiem jest sprawdzenie, czy aktualnie zalogowany użytkownik jest właścicielem lub adminem. Jeśli tak, to może przejść do widoku usuwania. Jeśli nie, zostanie zwrócony komunikat 400 — Bad Request.

Dodaj kod:

```
else if (ogloszenie.UzytkownikId != User.Identity.GetUserId() &&
         !User.IsInRole("Admin"))
{
    return new HttpStatusCodeResult(HttpStatusCode.BadRequest);
}
```

Po aktualizacji kod akcji wygląda następująco:

```
[Authorize]
public ActionResult Delete(int? id, bool? blad)
{
    if (id == null)
    {
        return new HttpStatusCodeResult(HttpStatusCode.BadRequest);
    }

    Ogloszenie ogloszenie = _repo.GetOgloszenieById((int)id);
```

```

if (ogloszenie == null)
{
    return HttpNotFound();
}
else if (ogloszenie.UzytkownikId != User.Identity.GetUserId() &
         ↳User.IsInRole("Admin"))
{
    return new HttpStatusCodeResult(HttpStatusCode.BadRequest);
}
if(blad !=null)
    ViewBag.Blad = true;
return View(ogloszenie);
}

```

Od teraz tylko autor i admin mogą usuwać ogłoszenia, pozostały użytkownicy oraz pracownicy dostaną odpowiedni komunikat (rysunek 8.110).

Rysunek 8.110.

Komunikat o błędzie podczas usuwania



Etap 5. Podsumowanie

Na tym etapie zabezpieczono wszystkie akcje z kontrolera. W widokach pokazują się tylko te linki, z których może korzystać dany użytkownik. W widoku Details link Edytuj został specjalnie pozostawiony dla wszystkich, aby zademonstrować zabezpieczenie po stronie akcji w kontrolerze, dlatego teraz to poprawimy.

Następujący kod:

```

<p>
@if (User.Identity.IsAuthenticated || User.IsInRole("Admin") ||
     ↳User.IsInRole("Pracownik"))
{
    @Html.ActionLink("Edytuj", "Edit", new { id = Model.Id }, new
        ↳{ @class = "btn btn-primary" })
    @: |
}
@Html.ActionLink("Wróć", "Index", null, new { @class =
    ↳"btn btn-warning" })
</p>

```

zamień na:

```

<p>
@if (User.Identity.IsAuthenticated && (User.IsInRole("Admin") ||
     ↳User.IsInRole("Pracownik") || Model.UzytkownikId ==
     ↳User.Identity.GetUserId()))
{
    @Html.ActionLink("Edytuj", "Edit", new { id = Model.Id }, new
        ↳{ @class = "btn btn-primary" })
    @: |
}

```

```
@Html.ActionLink("Wróć", "Index", null, new { @class =
    ➔"btn btn-warning" })
</p>
```

Na samej górze widoku dodaj dyrektywę `using`, dzięki której będzie można odczytać Id aktualnie zalogowanego użytkownika:

```
@using Microsoft.AspNetCore.Identity;
```

Kod widoku po dodaniu dyrektywy (początek) wygląda następująco:

```
@model Repository.Models.Ogłoszenie
@using Microsoft.AspNetCore.Identity;

 @{
    ViewBag.Tytuł = "Szczegóły ogłoszenia nr:" + Model.Id;
    ViewBag.Opis = "Szczegóły ogłoszenia nr:" + Model.Id + "
    ➔Opis do Goole";
    ViewBag.SłowaKluczowe = "Ogłoszenie, " + Model.Id +
    ➔", szczegóły";
}
```

Etap 6. Stronicowanie i sortowanie

Stronicowanie

Aplikacja jest zabezpieczona, jednak na liście ogłoszeń pobierane są wszystkie ogłoszenia. Dodamy zatem stronicowanie (paginację), aby nie pobierać wszystkich danych naraz.

Dodanie metody do repozytorium

Na początek do repozytorium trzeba dodać metodę, która będzie pobierała wybraną liczbę kolejnych elementów dla poszczególnych numerów stron. Do interfejsu `IÖgłoszenieRepo` dodaj deklarację metody `PobierzStrone()`:

```
IQueryable<Ogłoszenie> PobierzStrone(int? page, int? pageSize);
```

Do repozytorium `OgłoszenieRepo` dodaj metodę:

```
public IQueryable<Ogłoszenie> PobierzStrone(int? page = 1, int? pageSize = 10)
{
    var ogłoszenia = _db.Ogłoszenia
        .OrderByDescending(o => o.DataDodania)
        .Skip((page.Value - 1) * pageSize.Value)
        .Take(pageSize.Value);

    return ogłoszenia;
}
```

Metoda przyjmuje dwa parametry opcjonalne. Jeśli nie zostaną przekazane parametry, przypisywane są wartości domyślne, czyli pierwsza strona i 10 elementów na stronę. Elementy są sortowane malejąco według daty dodania, czyli od najnowszych ogłoszeń.

Metoda Skip służy do opuszczenia wybranej liczby elementów, czyli numer strony minus jeden razy liczba elementów na stronie. Metoda Take określa, ile elementów należy pobrać.

Aktualizacja kontrolera

Przerobimy akcję Index w kontrolerze, aby przyjmowała jeden opcjonalny parametr, czyli numer strony, oraz wywoływała nową metodę z repozytorium. Kod akcji po zmianach wygląda jak poniżej:

```
public ActionResult Index(int? page)
{
    int currentPage = page ?? 1;
    int naStronie = 5;
    var ogłoszenia = _repo.PobierzStrone(currentPage,naStronie);
    return View(ogłoszenia);
}
```

W kodzie ustawiono liczbę ogłoszeń na jednej stronie na wartość 5.

Teraz po uruchomieniu na stronie znajdą się trzy najnowsze ogłoszenia (rysunek 8.111).

Lista Ogłoszeń				
UzytkownikaId	Treść ogłoszenia:	Tytuł ogłoszenia:	Data dodania:	
d1fa895e-206f-4558-8270-3bf7b5bd45a7	XCVX	VXCV	2014-07-02	<button>Szczegóły</button> <button>Edytuj</button>
91825236-fdd2-49fa-82d1-ca6f4d30a225	Treść ogłoszenia2	Tytuł ogłoszenia2	2014-06-29	<button>Szczegóły</button> <button>Edytuj</button>
91825236-fdd2-49fa-82d1-ca6f4d30a225	Treść ogłoszenia3	Tytuł ogłoszenia3	2014-06-28	<button>Szczegóły</button> <button>Edytuj</button>
91825236-fdd2-49fa-82d1-ca6f4d30a225	Treść ogłoszenia4	Tytuł ogłoszenia4	2014-06-27	<button>Szczegóły</button> <button>Edytuj</button>
91825236-fdd2-49fa-82d1-ca6f4d30a225	Treść ogłoszenia5	Tytuł ogłoszenia5	2014-06-26	<button>Szczegóły</button> <button>Edytuj</button>

Rysunek 8.111. Pierwsza strona ogłoszeń

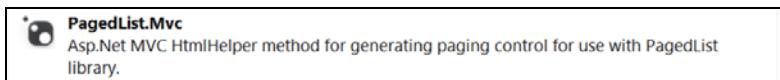
Aby zobaczyć drugą stronę ogłoszeń, odwiedź adres: /Ogłoszenie/Index?page=2.

Instalacja i użycie pakietu PagedList.Mvc

Kolejnym krokiem jest przerobienie widoku, aby można było przechodzić do następnych stron. Zainstaluj bibliotekę *PagedList.Mvc* dla projektu OGL za pomocą NuGet (rysunek 8.112).

Rysunek 8.112.

Instalacja biblioteki
PagedList



Zmień model danych dla widoku Index z:

```
@model IEnumerable<Repozytorium.Models.Ogłoszenie>
```

na:

```
@model PagedList.IPagedList<Repozytorium.Models.Ogłoszenie>
```

i dodaj dyrektywę:

```
@using PagedList.Mvc;
```

oraz plik ze stylami:

```
<link href="~/Content/PagedList.css" rel="stylesheet" type="text/css" />
```

Ostatecznie nagłówek wygląda następująco:

```
@model PagedList.IPagedList<Repozytorium.Models.Ogłoszenie>
```

```
@using PagedList.Mvc;
```

```
<link href="~/Content/PagedList.css" rel="stylesheet" type="text/css" />
```

W kolejnym kroku trzeba przerobić helper'y DisplayNameFor(), ponieważ nie radzą sobie z PagedList. Dodaj [0], aby określić, że mają wyświetlić nazwę dla pierwszego elementu z kolekcji:

```
<tr>
    <th>
        @Html.DisplayNameFor(model => model[0].UzytkownikId)
    </th>
    <th>
        @Html.DisplayNameFor(model => model[0].Tresc)
    </th>
    <th>
        @Html.DisplayNameFor(model => model[0].Tytul)
    </th>
    <th>
        @Html.DisplayNameFor(model => model[0].DataDodania)
    </th>
    <th></th>
</tr>
```

Teraz dodaj linki do stronicowania. Na samym końcu pliku wpisz następujący kod:

```
<div>
    <br />
    Strona @(Model.PageCount < Model.PageNumber ? 0 :
        ↪Model.PageNumber) z @Model.PageCount
    @Html.PagedListPager(Model, page => Url.Action("Index",
        ↪new { page }))</div>
```

Po wprowadzeniu wszystkich zmian widok wygląda jak na listingu 8.12.

Listing 8.12. Kompletny kod widoku

```
@model PagedList.IPagedList<Repozytorium.Models.Ogłoszenie>
@using PagedList.Mvc;
@link href="~/Content/PagedList.css" rel="stylesheet" type="text/css" />
{@
    ViewBag.Tytuł = "Lista ogłoszeń - metatytuł do 60 znaków";
    ViewBag.Opis = "Lista ogłoszeń z naszej aplikacji – metaopis do 160 znaków";
    ViewBag.SłowaKluczowe = "Lista, ogłoszeń, słowa, kluczowe,
        aplikacja";
}

<h2>Lista ogłoszeń</h2>
@if (User.Identity.IsAuthenticated)
{
    <p>
        @Html.ActionLink("Dodaj nowe ogłoszenie", "Create", null, new
            { @class = "btn btn-primary" })
    </p>
}
<table class="table">
    <tr>
        <th>
            @Html.DisplayNameFor(model => model[0].UzytkownikId)
        </th>
        <th>
            @Html.DisplayNameFor(model => model[0].Tresc)
        </th>
        <th>
            @Html.DisplayNameFor(model => model[0].Tytuł)
        </th>
        <th>
            @Html.DisplayNameFor(model => model[0].DataDodania)
        </th>
        <th></th>
    </tr>
    @foreach (var item in Model)
    {
        <tr>
            <td>
                @Html.DisplayFor(modelItem => item.UzytkownikId)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.Tresc)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.Tytuł)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.DataDodania)
            </td>
            <td>
                @Html.ActionLink("Szczegóły", "Details", new
                    { id = item.Id }, new { @class = "btn btn-warning" })
            </td>
        </tr>
    }

```

```
    @if (User.Identity.IsAuthenticated &&
        (User.IsInRole("Admin") ||
        User.IsInRole("Pracownik")))
    {
        <br />@Html.ActionLink("Edytuj ", "Edit", new { id =
            item.Id }, new { @class = "btn btn-primary" })
        if (User.IsInRole("Admin"))
        {
            <br />
            @Html.ActionLink("Usuń", "Delete", new { id =
                item.Id }, new { @class = "btn btn-danger" })
        }
    }
</td>
</tr>
}
</table>
<div>
<br />
Strona @(Model.PageCount < Model.PageNumber ? 0 :
    Model.PageNumber) z @Model.PageCount
@Html.PagedListPager(Model, page => Url.Action("Index", new
{
    page
}))
</div>
```

Widok jest gotowy. Teraz czas zająć się akcją Index. Przejdz do kontrolera OgloszenieController. Dodaj dyrektywę:

```
using PagedList;
```

Następnym krokiem będzie aktualizacja akcji Index. Nie trzeba już dodatkowej metody z repozytorium PobierzStrone(), ponieważ PagedList sam sobie generuje zapytania. Zamieniamy ją na metodę pobierającą wszystkie ogłoszenia o nazwie PobierzOgłoszenia(). Następnie, aby możliwa była paginacja, konieczne jest ustalenie kolejności, czyli pola, po którym będą sortowane dane. W tym przypadku będzie to data dodania (malejaco). Na końcu trzeba wywołać metodę:

```
.ToPagedList<Ogłoszenie>(currentPage, naStronie);
```

Kod akcji Index przed zmianami:

```
public ActionResult Index(int? page)
{
    int currentPage = page ?? 1;
    int naStronie = 5;
    var ogłoszenia = _repo.PobierzStrone(currentPage, naStronie);
    return View(ogłoszenia);
}
```

Kod akcji po zmianach (rysunek 8.113):

```
public ActionResult Index(int? page)
{
    int currentPage = page ?? 1;
    int naStronie = 3;
    var ogłoszenia = _repo.PobierzOgłoszenia();
```

Lista Ogłoszeń				
UzytkownikaId	Treść ogłoszenia:	Tytuł ogłoszenia:	Data dodania:	
d1fa895e-20ef-4558-8270-3bf7b5bd45a7	xcvx	vxcv	2014-07-02	Szczegóły Edytuj
91825236-fdd2-49fa-82d1-ca6f4d30a225	Treść ogłoszenia2	Tytuł ogłoszenia2	2014-06-29	Szczegóły Edytuj
91825236-fdd2-49fa-82d1-ca6f4d30a225	Treść ogłoszenia3	Tytuł ogłoszenia3	2014-06-28	Szczegóły Edytuj

Strona 1 z 4

1 2 3 4 >

Rysunek 8.113. Wygląd aplikacji z paginacją

```
ogloszenia = ogloszenia.OrderByDescending(d=>d.DataDodania);
return View(ogloszenia.ToPagedList<Ogloszenie>(currentPage,
    =>naStronie));
}
```

Zwróć teraz uwagę na zapytania, jakie są wysyłane do bazy danych. Za każdym razem wysyłane jest zapytanie:

```
SELECT
    [GroupBy1].[A1] AS [C1]
FROM (
    SELECT
        COUNT(1) AS [A1]
    FROM [dbo].[Ogloszenie] AS [Extent1]
) AS [GroupBy1]
```

Zwroca ono kompletną liczbę wierszy. Na podstawie wyniku tego zapytania obliczana jest liczba stron (liczba wierszy dzielona przez liczbę wyników na stronę).

Następnie wysyłane jest zapytanie o dane z konkretnej strony:

```
SELECT TOP (3)
    [Extent1].[Id] AS [Id],
    [Extent1].[Tresc] AS [Tresc],
    [Extent1].[Tytuł] AS [Tytuł],
    [Extent1].[DataDodania] AS [DataDodania],
    [Extent1].[UzytkownikaId] AS [UzytkownikaId]
FROM (
    SELECT [Extent1].[Id] AS [Id],
        [Extent1].[Tresc] AS [Tresc],
        [Extent1].[Tytuł] AS [Tytuł],
        [Extent1].[DataDodania] AS [DataDodania],
        [Extent1].[UzytkownikaId] AS [UzytkownikaId], row_number()
    =>OVER (ORDER BY
        [Extent1].[DataDodania] DESC) AS [row_number]
    FROM [dbo].[Ogloszenie] AS [Extent1]
```

```
) AS [Extent1]
WHERE [Extent1].[row_number] > 6
ORDER BY [Extent1].[DataDodania] DESC
```

Należy teraz dostosować do PagedList akcję o nazwie Partial z kontrolera OgloszenieController zwracającą widok PartialView, ponieważ korzysta z tego samego pliku z widokiem.

Zamień:

```
// GET: /Ogloszenie/
public ActionResult Partial()
{
    var ogloszenia = _repo.PobierzOgloszenia();
    return PartialView("Index", ogloszenia);
}
```

na:

```
// GET: /Ogloszenie/
public ActionResult Partial(int? page)
{
    int currentPage = page ?? 1;
    int naStronie = 3;
    var ogloszenia = _repo.PobierzOgloszenia();
    ogloszenia = ogloszenia.OrderByDescending(d =>
        d.DataDodania);
    return PartialView("Index", ogloszenia
        .ToPagedList<Ogloszenie>(currentPage, naStronie));
}
```

Widok PartialView wygląda teraz jak na rysunku 8.114.

UzytkownikaId	Treść ogłoszenia	Tytuł ogłoszenia	Data dodania	
d1fa895e-206f-4558-8270-3bf7b5bd45a7 xcvx	vxcv	2014-07-02	Szczegóły Edytuj	
91825236-fdd2-49fa-82d1-ca6f4d30a225	Treść ogłoszenia2	Tytuł ogłoszenia2	2014-06-29	Szczegóły Edytuj
91825236-fdd2-49fa-82d1-ca6f4d30a225	Treść ogłoszenia3	Tytuł ogłoszenia3	2014-06-28	Szczegóły Edytuj

Strona 1 z 4

1 2 3 4 »

Rysunek 8.114. Widok PartialView

Jak widać, przyciski paginacji mają przypisane style CSS. Dzieje się tak, ponieważ zamieszczono link do pliku CSS ze stylami dla PagedList na początku widoku, a nie w pliku z szablonem *_Layout.cshtml*.

Sortowanie

Sortowanie zostanie zaimplementowane w widoku *Lista ogłoszeń*, a więc w akcji Index z kontrolera OgloszenieController.

Aktualizacja kontrolera

Aby zaimplementować sortowanie, wykorzystamy ViewBag, w których będzie można przekazywać nazwę pola, po jakim mają zostać posortowane dane. Do akcji Index trzeba dodać parametr o nazwie sortOrder typu string przechowujący nazwę pola, po którym się sortuje. Poniższą linię:

```
public ActionResult Index(int? page)
```

zamień na:

```
public ActionResult Index(int? page, string sortOrder)
```

W kolejnym kroku dodaj kod:

```
ViewBag.CurrentSort = sortOrder;
ViewBag.IdSort = String.IsNullOrEmpty(sortOrder) ? "IdAsc" : "";
ViewBag.DataDodaniaSort = sortOrder == "DataDodania" ?
    ↳"DataDodaniaAsc" : "DataDodania";
ViewBag.TrescSort = sortOrder == "TrescAsc" ? "Tresc" : "TrescASC";
ViewBag.TytulSort = sortOrder == "TytulAsc" ? "Tytul" : "TytulASC";
```

Powyższy kod odpowiada za ustawienie pola, po którym się sortuje. Jeśli klikniesz ponownie to samo pole, kolejność sortowania zostanie odwrócona. Wykorzystaliśmy tutaj wyrażenie ?:.

Następnie po wywołaniu metody z repozytorium usuwamy linijkę odpowiedzialną za sortowanie po dacie i dodajemy kod:

```
switch (sortOrder)
{
    case "DataDodania":
        ogloszenia = ogloszenia.OrderByDescending(s =>
            ↳s.DataDodania);
        break;
    case "DataDodaniaAsc":
        ogloszenia = ogloszenia.OrderBy(s => s.DataDodania);
        break;
    case "Tytul":
        ogloszenia = ogloszenia.OrderByDescending(s => s.Tytul);
        break;
    case "TytulAsc":
        ogloszenia = ogloszenia.OrderBy(s => s.Tytul);
        break;
    case "Tresc":
        ogloszenia = ogloszenia.OrderByDescending(s => s.Tresc);
        break;
    case "TrescAsc":
        ogloszenia = ogloszenia.OrderBy(s => s.Tresc);
        break;
    case "IdAsc":
        ↳
```

```
    ogloszenia = ogloszenia.OrderBy(s => s.Id);
    break;
default: //id descending
ogloszenia = ogloszenia.OrderByDescending(s => s.Id);
break;
}
```

Jest to instrukcja switch, która w zależności od wybranego typu sortowania wywołuje metodę `OrderBy()` lub `OrderByDescending()` na zapytaniu wysłanym do repozytorium.

Ostatecznie kod kontrolera zaprezentowano na listingu 8.13.

Listing 8.13. Kompletny kod metody Index

```
public ActionResult Index(int? page, string sortOrder)
{
    int currentPage = page ?? 1;
    int naStronie = 3;

    ViewBag.CurrentSort = sortOrder;
    ViewBag.IdSort = String.IsNullOrEmpty(sortOrder) ? "IdAsc" : "";
    ViewBag.DataDodaniaSort = sortOrder == "DataDodania" ?
        "DataDodaniaAsc" : "DataDodania";
    ViewBag.TrescSort = sortOrder == "TrescAsc" ? "Tresc" :
        "TrescAsc";
    ViewBag.TytulSort = sortOrder == "TytulAsc" ? "Tytul" :
        "TytulAsc";

    var ogloszenia = _repo.PobierzOgloszenia();
    switch (sortOrder)
    {
        case "DataDodania":
            ogloszenia = ogloszenia.OrderByDescending(s => s.DataDodania);
            break;
        case "DataDodaniaAsc":
            ogloszenia = ogloszenia.OrderBy(s => s.DataDodania);
            break;
        case "Tytul":
            ogloszenia = ogloszenia.OrderByDescending(s => s.Tytul);
            break;
        case "TytulAsc":
            ogloszenia = ogloszenia.OrderBy(s => s.Tytul);
            break;
        case "Tresc":
            ogloszenia = ogloszenia.OrderByDescending(s => s.Tresc);
            break;
        case "TrescAsc":
            ogloszenia = ogloszenia.OrderBy(s => s.Tresc);
            break;
        case "IdAsc":
            ogloszenia = ogloszenia.OrderBy(s => s.Id);
            break;
        default: //id descending
            ogloszenia = ogloszenia.OrderByDescending(s => s.Id);
            break;
    }
}
```

```

        return View(ogloszenia.ToPagedList<Ogloszenie>(currentPage,
        ↳naStronie));
    }
}

```

Aktualizacja widoku

Mając gotowy kontroler, trzeba zaktualizować kod widoku, aby w linkach do stronowania i sortowania przekazywał wartość sortOrder.

Na początek zamień nagłówki wierszy, które będą teraz linkami i będą służyły do sortowania po klikniętym polu.

Poniższy kod:

```

<th>
    @Html.DisplayNameFor(model => model[0].UzytkownikId)
</th>
<th>
    @Html.DisplayNameFor(model => model[0].Tresc)
</th>
<th>
    @Html.DisplayNameFor(model => model[0].Tytul)
</th>
<th>
    @Html.DisplayNameFor(model => model[0].DataDodania)
</th>

```

zamień na:

```

<th>
    @Html.ActionLink("Id użytkownika", "Index", new { sortOrder =
        ↳ViewBag.IdSort })
</th>
<th>
    @Html.ActionLink("Treść", "Index", new { sortOrder =
        ↳ViewBag.TrescSort })
</th>
<th>
    @Html.ActionLink("Tytuł", "Index", new { sortOrder =
        ↳ViewBag.TytulSort })
</th>
<th>
    @Html.ActionLink("Data dodania", "Index", new { sortOrder =
        ↳ViewBag.DataDodaniaSort })
</th>

```

Kolejnym i ostatnim krokiem jest dodanie wartości sortOrder do linków do stronowania. Zamień linię:

```

@Html.PagedListPager(Model, page => Url.Action("Index",
        ↳new { page }))

```

na:

```

@Html.PagedListPager(Model, page => Url.Action("Index", new
        ↳{ page, sortOrder = ViewBag.CurrentSort }))

```

Kompletny plik z widokiem został zaprezentowany na listingu 8.14.

Listing 8.14. Kompletny kod widoku

```
@model PagedList.IagedList<Repozytorium.Models.Ogłoszenie>
@using PagedList.Mvc;
<link href="~/Content/PagedList.css" rel="stylesheet" type="text/css" />
{@
    ViewBag.Tytuł = "Lista ogłoszeń - metatytuł do 60 znaków";
    ViewBag.Opis = "Lista ogłoszeń z naszej aplikacji - 
                    ↳metaopis do 160 znaków";
    ViewBag.SłowaKluczowe = "Lista, ogłoszeń, słowa, kluczowe,
                            ↳aplikacja";
}

<h2>Lista ogłoszeń</h2>
@if (User.Identity.IsAuthenticated)
{
    <p>
        @Html.ActionLink("Dodaj nowe ogłoszenie", "Create", null,
                         ↳new { @class = "btn btn-primary" })
    </p>
}
<table class="table">
    <tr>
        <th>
            @Html.ActionLink("Id użytkownika", "Index", new
                             ↳{ sortOrder = ViewBag.IdSort })
        </th>
        <th>
            @Html.ActionLink("Treść", "Index", new { sortOrder =
                ↳ViewBag.TrescSort })
        </th>
        <th>
            @Html.ActionLink("Tytuł", "Index", new { sortOrder =
                ↳ViewBag.TytułSort })
        </th>
        <th>
            @Html.ActionLink("Data dodania", "Index", new
                             ↳{ sortOrder = ViewBag.DataDodaniaSort })
        </th>
        <th></th>
    </tr>

@foreach (var item in Model)
{
    <tr>
        <td>
            @Html.DisplayFor(modelItem => item.UzytkownikId)
        </td>
        <td>
            @Html.DisplayFor(modelItem => item.Tresc)
        </td>
        <td>
            @Html.DisplayFor(modelItem => item.Tytuł)
```

```

        </td>
        <td>
            @Html.DisplayFor(modelItem => item.DataDodania)
        </td>
        <td>
            @Html.ActionLink("Szczegóły", "Details", new
                { id = item.Id }, new { @class = "btn btn-warning" })
            @if (User.Identity.IsAuthenticated &&
                !(User.IsInRole("Admin") || User.IsInRole("Pracownik")))
            {
                <br />@Html.ActionLink("Edytuj ", "Edit", new
                    { id = item.Id }, new { @class = "btn btn-primary" })
                if (User.IsInRole("Admin"))
                {
                    <br />
                    @Html.ActionLink("Usuń", "Delete", new
                        { id = item.Id }, new { @class =
                            "btn btn-danger" })
                }
            }
        </td>
    </tr>
}
</table>

<div>
    <br />
    Strona @Model.PageCount < Model.PageNumber ? 0 :
        →Model.PageNumber) z @Model.PageCount

    @Html.PagedListPager(Model, page => Url.Action("Index", new { page,
        →sortOrder = ViewBag.CurrentSort}))
</div>

```

Uruchom aplikację — nagłówki są teraz linkami i mają inny kolor. Jeśli klikniesz nagłówek *Data dodania*, zobaczysz, że ogłoszenia są posortowane od najnowszych (rysunek 8.115).

Po kliknięciu kolejny raz nagłówka *Data dodania* widać, że ogłoszenia są teraz posortowane od najstarszych (rysunek 8.116).

Etap 6. Podsumowanie

Zaimplementowaliśmy paginację oraz sortowanie na podstawie kolumn dla widoku Index. Oprócz paginacji i sortowania można jeszcze dodać wyszukiwanie na podstawie wprowadzonego tekstu po wybranej kolumnie. Wyszukiwanie działa na podobnej zasadzie jak sortowanie — trzeba dodać jeszcze kilka parametrów do akcji i jedną instrukcję switch.

Lista Ogłoszeń				
Dodaj nowe ogłoszenie				
Id Użytkownika	Treść	Tytuł	Data Dodania	
6df2d16d-4eaa-4c4e-acb8-70f832b89cd3	gfhjgh	jfhjf	2014-07-03	Szczegóły Edytuj Usuń
6df2d16d-4eaa-4c4e-acb8-70f832b89cd3	nfgnf	nfghn	2014-07-02	Szczegóły Edytuj Usuń
6df2d16d-4eaa-4c4e-acb8-70f832b89cd3	tut	Tutu	2014-07-02	Szczegóły Edytuj Usuń

Rysunek 8.115. Posortowane ogłoszenia (od najnowszych)

Lista Ogłoszeń				
Dodaj nowe ogłoszenie				
Id Użytkownika	Treść	Tytuł	Data Dodania	
91825236-fdd2-49fa-82d1-ca6f4d30a225	Treść ogłoszenia10	Tytuł ogłoszenia10	2014-06-21	Szczegóły Edytuj
91825236-fdd2-49fa-82d1-ca6f4d30a225	Treść ogłoszenia9	Tytuł ogłoszenia9	2014-06-22	Szczegóły Edytuj
91825236-fdd2-49fa-82d1-ca6f4d30a225	Treść ogłoszenia8	Tytuł ogłoszenia8	2014-06-23	Szczegóły Edytuj

Rysunek 8.116. Posortowane ogłoszenia (od najstarszych)

Etap 7. Ogłoszenia użytkownika, kategorie, cache i ViewModel

Zakładka Moje ogłoszenia

Teraz dodamy zakładkę, która wyświetli tylko ogłoszenia dodane przez zalogowanego użytkownika. Wykorzystamy gotowy plik z widokiem `Index`. Przeniesiemy jego zawartość i dodamy nowy plik z widokiem o nazwie `MojeOgłoszenia` do folderu `Views/Ogłoszenie`, do którego wcześniej została skopiowana zawartość pliku `Index`.

W pliku zmień tytuł w znaczniku `<h2>`:

```
<h2>Lista ogłoszeń</h2>
```

na:

```
<h2>Lista moich ogłoszeń</h2>
```

Następnie zaktualizuj linki paginacji, aby kierowały do akcji MojeOgłoszenia. Poniższą linię:

```
@Html.PagedListPager(Model, page => Url.Action("Index", new
    { page }))
```

zamień na:

```
@Html.PagedListPager(Model, page => Url.Action("MojeOgłoszenia", new
    { page }))
```

Dodaj teraz akcję do kontrolera Ogłoszenie o nazwie MojeOgłoszenia:

```
public ActionResult MojeOgłoszenia(int? page)
{
    int currentPage = page ?? 1;
    int naStronie = 3;
    string userId = User.Identity.GetUserId();
    var ogłoszenia = _repo.PobierzOgłoszenia();
    ogłoszenia = ogłoszenia.OrderByDescending(d => d.DataDodania)
        .Where(o=>o.UzytkownikId == userId);
    return View(ogłoszenia.ToPagedList<Ogłoszenie>(currentPage,
        naStronie));
}
```

W porównaniu do akcji Index do zapytania został dodany jeden warunek:

```
.Where(o=>o.UzytkownikId == userId);
```

Aby po dodaniu ogłoszenia zostać przekierowywanym do akcji Index oraz aby po dodaniu otwierała się zakładka MojeOgłoszenia, w metodzie POST dla akcji Create zamień linijkę:

```
return RedirectToAction("Index");
```

na:

```
return RedirectToAction("MojeOgłoszenia");
```

Wygląd strony po przeróbkach został zaprezentowany na rysunku 8.117.

Cache

Do cachowania zostanie wykorzystany atrybut [OutputCache], który zapisuje wynik akcji w pamięci i przy kolejnych żądaniach do tej akcji przez określoną ilość czasu zwraca zapisane w pamięci dane, zamiast wykonywać akcję.

Dodaj do akcji MojeOgłoszenia atrybut:

```
[OutputCache(Duration=1000)]
```

Lista Moich Ogłoszeń				
Dodaj nowe ogłoszenie				
UzytkownikaId	Treść ogłoszenia:	Tytuł ogłoszenia:	Data dodania:	
6df2d16d-4eaa-4c4e-acb8-70f832b89cd3	new Treść ogłoszenia	new Tytuł ogłoszenia	2014-07-02	Szczegóły Edytuj Usuń
6df2d16d-4eaa-4c4e-acb8-70f832b89cd3	Moje ogłoszenie najnowsze	moje ogłoszenie najnowsze	2014-07-02	Szczegóły Edytuj Usuń
6df2d16d-4eaa-4c4e-acb8-70f832b89cd3	Moje ogłoszenie nowe	Moje ogłoszenie nowe	2014-07-02	Szczegóły Edytuj Usuń

Strona 1 z 2

1 2 >

Rysunek 8.117. Widok strony po zmianach

Akcja wygląda teraz następująco:

```
[OutputCache(Duration=1000)]
public ActionResult MojeOgłoszenia(int? page)
{
    ...
}
```

Uruchom aplikację i przejdź do zakładki *Moje ogłoszenia* (rysunek 8.118).

Usuń pierwsze ogłoszenie i przejdź ponownie do zakładki *Moje ogłoszenia*. Jak widać, ogłoszenie, które zostało prawidłowo usunięte, nadal jest na liście, a podczas odświeżania strony zapytanie do bazy danych nie zostało wysłane. Po naciśnięciu przycisku *Szczegóły* dla tego ogłoszenia pojawia się *Błąd 404 — Not Found* (rysunek 8.119).

Dzieje się tak, ponieważ dane zostały zwrócone z cache. Dopiero gdy minie czas, który podano w polu duration, a więc 1000 sekund, i odświeży się ponownie zakładkę *Moje ogłoszenia*, zostanie wykonana akcja (nowe zapytanie do bazy danych) i zwrócone z zapytania dane zostaną ponownie zapisane w pamięci na 1000 sekund.

Kategorie

Podobnie jak przy ogłoszeniach, dodaj kontroler. Wybierz opcję *MVC5 Controller with views, using Entity Framework* i nazwij go *KategoriaController*. Wybierz klasę z modelem *Kategoria* i klasę z kontekstem, czyli *OglContext*. Zostaną wygenerowane widoki i kontroler z akcjami CRUD. Tym razem usuń wszystkie akcje z kontrolera oprócz akcji Index. Nie będzie potrzeby implementacji metod Create, Details, Edit i Delete, ponieważ robi się to analogicznie jak w przypadku ogłoszeń. Zostanie tylko zaimplementowana akcja Index, aby wyświetlić wszystkie dostępne kategorie.

Lista Moich Ogłoszeń				
UzytkownikaId	Treść ogłoszenia:	Tytuł ogłoszenia:	Data dodania:	
6df2d16d-4eaa-4c4e-acb8-70f832b89cd3	new Treść ogłoszenia	new Tytuł ogłoszenia	2014-07-02	Szczegóły Edytuj Usuń
6df2d16d-4eaa-4c4e-acb8-70f832b89cd3	Moje ogłoszenie najnowsze	moje ogłoszenie najnowsze	2014-07-02	Szczegóły Edytuj Usuń
6df2d16d-4eaa-4c4e-acb8-70f832b89cd3	Moje ogłoszenie nowe	Moje ogłoszenie nowe	2014-07-02	Szczegóły Edytuj Usuń

Strona 1 z 2

1 2 >

Rysunek 8.118. Lista ogłoszeń**Rysunek 8.119.**

*Błąd w dostępie
do danych*

Błąd HTTP 404.0 — Not Found

Zasób, do którego chcesz uzyskać dostęp, został usunięty,

Kod kontrolera po usunięciu niepotrzebnych akcji:

```
public class KategoriaController : Controller
{
    private OglContext db = new OglContext();

    // GET: Kategoria
    public ActionResult Index()
    {
        return View(db.Kategorie.ToList());
    }
}
```

Implementacja interfejsu i klasy repozytorium

Dodaj teraz interfejs IKategoriaRepo do folderu *IRepo* oraz klasę KategoriaRepo do folderu *Repo*.

Do interfejsu dodaj deklarację metody PobierzKategorie(). Kod interfejsu IKategoriaRepo wygląda następująco:

```
public interface IKategoriaRepo
{
    IQueryable<Kategoria> PobierzKategorie();
}
```

Do klasy KategoriaRepo dodaj definicję metody PobierzKategorie(), która jest niemal identyczna jak metoda PobierzOgłoszenia() z repozytorium OgłoszenieRepo. Podobnie jak w repozytorium ogłoszeń, trzeba wstrzykiwać instancję kontekstu poprzez konstruktor. Klasa musi dziedziczyć po interfejsie IKategoriaRepo. Ostateczny kod klasy KategoriaRepo wygląda jak poniżej:

```
public class KategoriaRepo : IKategoriaRepo
{
    private readonly IOglContext _db;
    public KategoriaRepo(IOglContext db)
    {
        _db = db;
    }

    public IQueryable<Kategoria> PobierzKategorie()
    {
        _db.Database.Log = message => Trace.WriteLine(message);
        var kategorie = _db.Kategorie.AsNoTracking();
        return kategorie;
    }
}
```

Importujemy biblioteki dla wszystkich elementów klasy.

Implementacja kontrolera

Podobnie jak w kontrolerze OgłoszenieController, będzie można wstrzykiwać instancje repozytorium, tylko tym razem będzie to KategoriaRepo, a nie OgłoszenieRepo. W akcji Index wykorzystamy metodę PobierzKategorie() z repozytorium, aby pobrać listę kategorii.

Kod kontrolera wygląda teraz następująco:

```
public class KategoriaController : Controller
{
    private readonly IKategoriaRepo _repo;

    public KategoriaController(IKategoriaRepo repo)
    {
        _repo = repo;
    }

    // GET: Kategoria
    public ActionResult Index()
    {
        var kategorie = _repo.PobierzKategorie().AsNoTracking();
        return View(kategorie);
    }
}
```

Wstrzykiwanie implementacji dzięki IoC

Uruchom aplikację i przejdź do podstrony *Kategoria*. Zostaje wyświetlony błąd widoczny na rysunku 8.120.

Rysunek 8.120.

Błąd dla podstrony
Kategoria

Błąd serwera w aplikacji '/'.

The current type, `Repozytorium.IRepo.IKategoriaRepo`, is an interface and cannot be constructed. Are you missing a type mapping?

Aplikacja nie może utworzyć instancji interfejsu IKategoriaRepo, co jest prawdą, ponieważ nie można utworzyć instancji interfejsu. Oznacza to, że w miejsce interfejsu nie została wstrzyknięta instancja klasy repozytorium. Jest to oczywiste, ponieważ w konfiguracji pakietu *Unity* nie ustawiono wstrzykiwania klasy KategoriaRepo w miejsce interfejsu IKategoriaRepo. Przejdz do pliku *UnityConfig* z folderu *App_Start* i dodaj w metodzie RegisterTypes() następującą linijkę:

```
container.RegisterType<IKategoriaRepo, KategoriaRepo>(new
    =>PerRequestLifetimeManager());
```

Ostatecznie metoda wygląda tak:

```
public static void RegisterTypes(IUnityContainer container)
{
    container.RegisterType<AccountController>(new
        =>InjectionConstructor());
    container.RegisterType<ManageController>(new
        =>InjectionConstructor());
    container.RegisterType<IOgloszenieRepo, OgloszenieRepo>(new
        =>PerRequestLifetimeManager());
    container.RegisterType<IKategoriaRepo, KategoriaRepo>(new
        =>PerRequestLifetimeManager());
    container.RegisterType<IOglContext, OglContext>(new
        =>PerRequestLifetimeManager());
}
```

Uruchom ponownie aplikację i przejdź do podstrony *Kategoria*. Zostanie zwrócona lista kategorii (rysunek 8.121).

Index

[Create New](#)

Nazwa kategorii: Co?	Id rodzica:	Tytuł w Google:	Opis strony w Google:	Słowa kluczowe Google:	Treść strony:	
Nazwa Kategorii1	1	Tytuł kategorii1	Opis kategorii1	Słowa kluczowe do kategorii1	Treść ogłoszenia1	Edit Details Delete
Nazwa Kategorii2	2	Tytuł kategorii2	Opis kategorii2	Słowa kluczowe do kategorii2	Treść ogłoszenia2	Edit Details Delete
Nazwa Kategorii3	3	Tytuł kategorii3	Opis kategorii3	Słowa kluczowe do kategorii3	Treść ogłoszenia3	Edit Details Delete
Nazwa Kategorii4	4	Tytuł kategorii4	Opis kategorii4	Słowa kluczowe do kategorii4	Treść ogłoszenia4	Edit Details Delete
Nazwa Kategorii5	5	Tytuł kategorii5	Opis kategorii5	Słowa kluczowe do kategorii5	Treść ogłoszenia5	Edit Details Delete

Rysunek 8.121. Podstrona *Kategoria*

Można jeszcze usuwać linki *Create New*, *Edit*, *Details* i *Delete*, ponieważ nie będą używane (nie są zaimplementowane).

Wyświetlanie ogłoszeń z wybranej kategorii LINQ (Query Syntax)

Ostatnim krokiem będzie utworzenie widoku dla listy ogłoszeń z wybranej kategorii. Po kliknięciu nazwy kategorii będzie można przejść do listy ogłoszeń z tej kategorii.

Usuń linki (CRUD), ponieważ nie będą już potrzebne. Przejdź do pliku o nazwie *Index* w folderze *Views/Kategoria* i w miejsce nazwy kategorii:

```
@Html.DisplayFor(modelItem => item.Nazwa)
```

wpisz kod:

```
@Html.ActionLink(item.Nazwa, "PokazOgłoszenia", new { id = item.Id })
```

Od teraz nazwa kategorii będzie linkiem do akcji *PokazOgłoszenia* z kontrolera *Kategoria*.

Ostateczny kod widoku *Index* dla kontrolera *Kategoria* prezentuje listing 8.15, a widok strony — rysunek 8.122.

Listing 8.15. Kod widoku Index

```
@model IEnumerable<Repozytorium.Models.Kategoria>

{
    ViewBag.Tytul = "Lista kategorii - metatytuł do 60 znaków";
    ViewBag.Opis = "Lista kategorii z naszej aplikacji - 
                    ↳metaopis do 160 znaków";
    ViewBag.SłowaKluczowe = "Lista, kategorii, słowa, kluczowe,
                            ↳aplikacja";
}

<h2>Lista kategorii</h2>



| <@Html.DisplayNameFor(model => model.Nazwa)>    | <@Html.DisplayNameFor(model => model.ParentId)>  | <@Html.DisplayNameFor(model => model.MetaTytul)> |
|-------------------------------------------------|--------------------------------------------------|--------------------------------------------------|
| <@Html.DisplayNameFor(model => model.MetaOpis)> | <@Html.DisplayNameFor(model => model.MetaSłowa)> | <@Html.DisplayNameFor(model => model.Tresc)>     |



@foreach (var item in Model)
{
    <tr>
        <td>
            @Html.ActionLink(item.Nazwa, "PokazOgłoszenia", new {
                id = item.Id })
        </td>
    </tr>
}
```

Lista Kategorii					
Nazwa kategorii: Co?	Id rodzica:	Tytuł w Google:	Opis strony w Google:	Słowa kluczowe Google:	Treść strony:
Nazwa Kategorii1	1	Tytuł kategorii1	Opis kategorii1	Słowa kluczowe do kategorii1	Treść ogłoszenia1
Nazwa Kategorii2	2	Tytuł kategorii2	Opis kategorii2	Słowa kluczowe do kategorii2	Treść ogłoszenia2
Nazwa Kategorii3	3	Tytuł kategorii3	Opis kategorii3	Słowa kluczowe do kategorii3	Treść ogłoszenia3
Nazwa Kategorii4	4	Tytuł kategorii4	Opis kategorii4	Słowa kluczowe do kategorii4	Treść ogłoszenia4

Rysunek 8.122. Wygląd strony

```

<td>
    @Html.DisplayFor(modelItem => item.ParentId)
</td>
<td>
    @Html.DisplayFor(modelItem => item.MetaTytul)
</td>
<td>
    @Html.DisplayFor(modelItem => item.MetaOpis)
</td>
<td>
    @Html.DisplayFor(modelItem => item.MetaSlowa)
</td>
<td>
    @Html.DisplayFor(modelItem => item.Tresc)
</td>
</tr>
}
</table>

```

Aby wyświetlić ogłoszenia z wybranej kategorii, dodaj następujący kod akcji PokazOgłoszenia do kontrolera KategoriaController:

```

public ActionResult PokazOgłoszenia(int id)
{
    var ogłoszenia = _repo.PobierzOgłoszeniaZKategorii(id);
    return View(ogłoszenia);
}

```

Dodaj jeszcze deklarację metody PobierzOgłoszeniaZKategorii() do interfejsu IKategoriaRepo. Interfejs wygląda teraz następująco:

```

public interface IKategoriaRepo
{
    IQueryble<Kategoria> PobierzKategorie();
    IQueryble<Ogłoszenie> PobierzOgłoszeniaZKategorii(int id);
}

```

Ostatnim krokiem jest dodanie metody PobierzOgłoszeniaZKategorii() do repozytorium KategoriaRepo:

```

public IQueryble<Ogłoszenie> PobierzOgłoszeniaZKategorii(int id)
{
    _db.Database.Log = message => Trace.WriteLine(message);
}

```

```

var ogloszenia =
    ↵from o in _db.Ogloszenia
    ↵join k in _db.Ogloszenie_Kategoria on o.Id equals k.Id
    ↵      where k.KategoriaId == id
    ↵      select o;

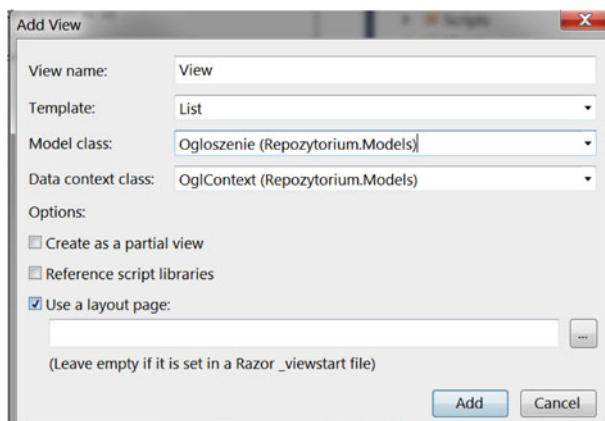
return ogloszenia;
}

```

W zapytaniu wykorzystaliśmy składnię zapytań dla LINQ (ang. *Query Syntax*) oraz operację `join`, aby pobrać te ogłoszenia, które są powiązane z wybraną kategorią.

Teraz trzeba dodać jeszcze widok dla akcji `PokazOgloszenia`. Kliknij prawym przyciskiem myszy katalog *Kategoria* w folderze *Views* i wybierz *Add/View*. Zostanie wyświetcone okno dodawania widoku (rysunek 8.123). Nazwa pozostaje taka sama jak nazwa akcji w kontrolerze, czyli *PokazOgloszenia*. Jako szablon wybierz listę w klasie z modelem *Ogloszenie*, ponieważ będzie to lista ogłoszeń z kategorii. Klasa z kontekstem to *OglContext*.

Rysunek 8.123.
Okno dodawania
widoku



Po wygenerowaniu widoku usuń linki do innych akcji, a w nagłówku <h2> dodaj tekst Ogłoszenia z kategorii: oraz zamień Email na UzytkownikId (listing 8.16, rysunek 8.124).

Listing 8.16. Ostateczny kod widoku

```

@model IEnumerable<Repozytorium.Models.Ogloszenie>

 @{
    ViewBag.Tytul = "Ogłoszenia z kategorii";
}

<h2>Ogłoszenia z kategorii</h2>

<table class="table">
    <tr>
        <th>
            @Html.DisplayNameFor(model => model.UzytkownikId)
        </th>
        <th>

```

Ogłoszenia z kategorii:			
UzytkownikId	Treść ogłoszenia:	Tytuł ogłoszenia:	Data dodania:
91825236-fdd2-49fa-82d1-ca6f4d30a225	Treść ogłoszenia4	Tytuł ogłoszenia4	2014-06-27
91825236-fdd2-49fa-82d1-ca6f4d30a225	Treść ogłoszenia5	Tytuł ogłoszenia5	2014-06-26

Rysunek 8.124. Wygląd strony dla adresu: /Kategoria/PokazOgłoszenia/5

```

@Html.DisplayNameFor(model => model.Tresc)
</th>
<th>
    @Html.DisplayNameFor(model => model.Tytul)
</th>
<th>
    @Html.DisplayNameFor(model => model.DataDodania)
</th>
</tr>

@foreach (var item in Model)
{
    <tr>
        <td>
            @Html.DisplayFor(modelItem => item.UzytkownikId)
        </td>
        <td>
            @Html.DisplayFor(modelItem => item.Tresc)
        </td>
        <td>
            @Html.DisplayFor(modelItem => item.Tytul)
        </td>
        <td>
            @Html.DisplayFor(modelItem => item.DataDodania)
        </td>
    </tr>
}
</table>

```

Zastosowanie HTML helpera — Html.Action

Kolejnym krokiem jest wygenerowanie akcji `Partial` z kontrolera `Ogłoszenie` (adres URL `/Ogłoszenie/Partial`). Domyślnie akcja zwraca widok pokazany na rysunku 8.125.

Po otworzeniu podstrony z ogłoszeniami dla wybranej kategorii zostanie wyświetlony ten sam widok `Partial` pod listą ogłoszeń. Dodaj na samym końcu pliku z widokiem `PokazOgłoszenia` linijkę:

```
@Html.Action("Partial", "Ogłoszenie", null)
```

Pod widokiem z ogłoszeniami z kategorii został wyświetlony widok `Partial` prezentujący wszystkie ogłoszenia (rysunek 8.126). Tym razem widok `Partial` wygląda lepiej, ponieważ w szablonie zostały wczytane style CSS. HTML helper `Action()` wywołuje akcję z kontrolera i wyświetla „widok w widoku”.

Lista Ogłoszeń

[Dodać nowe ogłoszenie](#)

UzytkownikaId	Treść ogłoszenia:	Tytuł ogłoszenia:	Data dodania:	
6df2d16d-4caa-4c4e-acb8-70f832b89cd3	Moje ogłoszenie najnowsze moje ogłoszenie najnowsze	moje ogłoszenie najnowsze	2014-07-02	Szczegóły Edytuj Usuń
6df2d16d-4eaa-4c4e-acb8-70f832b89cd3	Moje ogłoszenie nowe	Moje ogłoszenie nowe	2014-07-02	Szczegóły Edytuj Usuń
6df2d16d-4eaa-4c4e-acb8-70f832b89cd3	cbvncvb	ncvbnc	2014-07-02	Szczegóły Edytuj Usuń

Strona 1 z 5

[1](#) [2](#) [3](#) [4](#) [5](#) [»](#)

Rysunek 8.125. Domyślny widok Partial View

Ogłoszenia z kategorii:

UzytkownikaId	Treść ogłoszenia:	Tytuł ogłoszenia:	Data dodania:
91825236-fdd2-49fa-82d1-ca6f4d30a225	Treść ogłoszenia4	Tytuł ogłoszenia4	2014-06-27
91825236-fdd2-49fa-82d1-ca6f4d30a225	Treść ogłoszenia5	Tytuł ogłoszenia5	2014-06-26

Lista Ogłoszeń

[Dodać nowe ogłoszenie](#)

UzytkownikaId	Treść ogłoszenia:	Tytuł ogłoszenia:	Data dodania:	
6df2d16d-4eaa-4c4e-acb8-70f832b89cd3	Moje ogłoszenie najnowsze	moje ogłoszenie najnowsze	2014-07-02	Szczegóły Edytuj Usuń
6df2d16d-4eaa-4c4e-acb8-70f832b89cd3	Moje ogłoszenie nowe	Moje ogłoszenie nowe	2014-07-02	Szczegóły Edytuj Usuń
6df2d16d-4eaa-4c4e-acb8-70f832b89cd3	cbvncvb	ncvbnc	2014-07-02	Szczegóły Edytuj Usuń

Rysunek 8.126. Po uruchomieniu widać „widok w widoku”

Teraz, po sprawdzeniu działania Action(), można usunąć dodaną linię kodu i widok wróci do poprzedniego stanu.

Zastosowanie ViewModel

W przedstawionym rozwiążaniu brakuje nazwy kategorii, z której są ogłoszenia. Trzeba więc pobrać nazwę z bazy danych na podstawie id, a następnie przekazać do modelu. Wcześniej korzystaliśmy z ViewBag, aby przesyłać dane do widoku. Teraz użyjemy ViewModel, a więc specjalnej klasy z modelem, która będzie pasowała tylko do tego

widoku. W klasie znajdzie się lista ogłoszeń oraz nazwa kategorii. Na początek zostanie utworzona klasa ViewModel o nazwie OgloszeniaZKategoriiViewModels w folderze *Models/Views*.

Kod klasy wygląda następująco:

```
public class OgloszeniaZKategoriiViewModels
{
    public IList<Ogloszenie> Ogloszenia { get; set; }
    public string NazwaKategorii { get; set; }
}
```

Następnie dodaj deklarację metody *NazwaDlaKategorii* zwracającej nazwę kategorii dla podanego id kategorii. Do interfejsu *IKategoriaRepo* dodaj deklarację:

```
string NazwaDlaKategorii(int id);
```

W pliku *KategoriaRepo* dodaj metodę *NazwaDlaKategorii*:

```
public string NazwaDlaKategorii(int id)
{
    var nazwa = _db.Kategorie.Find(id).Nazwa;
    return nazwa;
}
```

W kontrolerze *KategoriaController* utwórz nowy obiekt typu *OgloszeniaZKategoriiViewModels*. Następnie przypisz do kolekcji z ViewModel o nazwie *Ogloszenia* listę ogłoszeń pobranych z bazy danych, a do pola *NazwaKategorii* przypisz wartość zwróconą z metody *NazwaDlaKategorii()*. Kod przed aktualizacją wygląda jak poniżej:

```
public ActionResult PokazOgloszenia(int id)
{
    var ogloszenia = _repo.PobierzOgloszeniaZKategorii(id);
    return View(ogloszenia);
}
```

A oto kod po aktualizacji:

```
public ActionResult PokazOgloszenia(int id)
{
    var ogloszenia = _repo.PobierzOgloszeniaZKategorii(id);
    OgloszeniaZKategoriiViewModels model = new
        OgloszeniaZKategoriiViewModels();
    model.Ogloszenia = ogloszenia.ToList();
    model.NazwaKategorii = _repo.NazwaDlaKategorii(id);
    return View(model);
}
```

Ostatnim elementem jest utworzenie opierającego się na ViewModel typowanego widoku *OgloszeniaZKategoriiViewModels*. Na początku trzeba zmienić typ widoku, ponieważ teraz będziemy pobierać dane w formie ViewModel, a nie w formie kolekcji ogłoszeń.

Poniższą linię kodu:

```
@model IEnumerable<Repozytorium.Models.Ogloszenie>
```

zamień na:

```
@model Repository.Models.Views.OgloszeniaZKategoriiViewModels
```

Następnie w nagłówku <h2> i tytule strony wyświetli nazwę kategorii przesłaną poprzez ViewModel:

```
@{  
    ViewBag.Tytul = "Ogłoszenia z kategorii: " +  
        Model.NazwaKategorii;  
}  
<h2>Ogłoszenia z kategorii: @Model.NazwaKategorii</h2>
```

Ostatnią zmianą będzie zmiana dla HTML helpera DisplayNameFor, w którym będzie teraz zwracany tytuł dla pierwszego elementu z kolekcji Ogloszenia wchodzącej w skład ViewModel. Przykład:

```
@Html.DisplayNameFor(model => model.Ogloszenia.First().UzytkownikId)
```

Pętla wyświetlająca dane będzie działać teraz na kolekcji Model.Ogloszenia.

Ostateczny kod widoku zaprezentowano na listingu 8.17.

Listing 8.17. Kod widoku

```
@model Repository.Models.Views.OgloszeniaZKategoriiViewModels  
  
{@  
    ViewBag.Tytul = "Ogłoszenia z kategorii: " +  
        Model.NazwaKategorii;  
}  
  
<h2>Ogłoszenia z kategorii: @Model.NazwaKategorii</h2>  
  
<table class="table">  
    <tr>  
        <th>  
            @Html.DisplayNameFor(model =>  
                model.Ogloszenia.First().UzytkownikId)  
        </th>  
        <th>  
            @Html.DisplayNameFor(model => model.Ogloszenia.First().Tresc)  
        </th>  
        <th>  
            @Html.DisplayNameFor(model => model.Ogloszenia.First().Tytul)  
        </th>  
        <th>  
            @Html.DisplayNameFor(model =>  
                model.Ogloszenia.First().DataDodania)  
        </th>  
    </tr>  
  
    @foreach (var item in Model.Ogloszenia)  
    {  
        <tr>  
            <td>  
                @Html.DisplayFor(modelItem => item.UzytkownikId)
```

```

        </td>
        <td>
            @Html.DisplayFor(modelItem => item.Tresc)
        </td>
        <td>
            @Html.DisplayFor(modelItem => item.Tytul)
        </td>
        <td>
            @Html.DisplayFor(modelItem => item.DataDodania)
        </td>
    </tr>
}
</table>

```

Uruchom aplikację i przejdź do zakładki *Kategoria*, po czym kliknij nazwę kategorii i przejdź do listy ogłoszeń z tej kategorii (rysunek 8.127). Teraz zarówno w tytule, jak i w nagłówku wyświetla się nazwa kategorii (rysunek 8.128).

Ogłoszenia z kategorii: Nazwa Kategorii3			
UzytkownikaId	Treść ogłoszenia:	Tytuł ogłoszenia:	Data dodania:
91825236-fdd2-49fa-82d1-ca6f4d30a225	Treść ogłoszenia2	Tytuł ogłoszenia2	2014-06-29
91825236-fdd2-49fa-82d1-ca6f4d30a225	Treść ogłoszenia3	Tytuł ogłoszenia3	2014-06-28

Rysunek 8.127. Lista ogłoszeń z nazwą kategorii, do której zostały przypisane

Rysunek 8.128.

Nazwa kategorii
w tytule strony



Etap 7. Podsumowanie

W tej części utworzyliśmy zakładkę z kategoriami, wprowadziliśmy cache i ViewModel. Wygenerowaliśmy również akcję z widokiem PartialView jako część innego widoku dzięki `Html.Action()`. Wykorzystaliśmy składnię zapytań (ang. *Query Syntax*) dla wyrażenia LINQ, aby zwrócić ogłoszenia powiązane z wybraną kategorią.



Jeśli podczas używania metody `Include` w zapytaniu Method Syntax, np.:

```

var ogloszenia = _db.Ogloszenia.Include(m =>
    m.Ogloszenie_Kategoria);

```

pojawi się błąd:

Cannot convert lambda expression to type 'string' because it is not a delegate type
należy dodać dyrektywę:

```
using System.Data.Entity;
```

Etap 8. Dane w JSON, zarządzanie relacją „wiele do wielu” i attribute routing

W tym etapie utworzymy akcję, która będzie zwracać dane kategorii w postaci JSON.

Dodaj metodę do kontrolera KategoriaController o nazwie KategorieWJson. Następnie pobierz dane za pomocą metody PobierzKategorie() z repozytorium. W odróżnieniu od standardowych metod, które zwracają widoki, ta metoda zwróci dane w formacie JSON:

```
public ActionResult KategorieWJson()
{
    var kategorie = _repo.PobierzKategorie().AsNoTracking();
    return Json(kategorie, JsonRequestBehavior.AllowGet);
}
```

Poniżej pokazano dane zwrócone po odwiedzeniu adresu /Kategoria/KategorieWJson/:

```
[{"Id":1,"Nazwa":"Nazwa kategorii1","ParentId":1,"MetaTytul":"Tytuł
↳kategorii1","MetaOpis":"Opis kategorii1","MetaSłowa":
↳"Słowa kluczowe do kategorii1","Tresc":"Treść
↳ogłoszenial","Ogłoszenie_Kategoria":[]},{ "Id":2,"Nazwa":"Nazwa
↳kategorii2","ParentId":2,"MetaTytul":"Tytuł kategorii2","MetaOpis":"Opis
↳kategorii2","MetaSłowa":"Słowa kluczowe do kategorii2","Tresc":"Treść
↳ogłoszenia2","Ogłoszenie_Kategoria":[]},
...
, {"Id":9,"Nazwa":"Nazwa kategorii9","ParentId":9,"MetaTytul":"Tytuł
↳kategorii9","MetaOpis":"Opis kategorii9","MetaSłowa":"Słowa kluczowe do
kategorii9","Tresc":"Treść
↳ogłoszenia9","Ogłoszenie_Kategoria":[]},{ "Id":10,"Nazwa":"Nazwa
↳kategorii10","ParentId":10,"MetaTytul":"Tytuł kategorii10","MetaOpis":"Opis
↳kategorii10","MetaSłowa":"Słowa kluczowe do kategorii10","Tresc":"Treść
↳ogłoszenia10","Ogłoszenie_Kategoria":[]}]
```

PartialView a dane w formacie JSON lub XML

Dane zwrócone z widoku Partial posiadają formatowanie HTML. Dane zwrócone z Web serwisu, Web API lub z akcji w formacie JSON nie mają kodu HTML, tylko strukturę JSON bądź XML. Dane w formacie HTML mogą zostać wklejone bezpośrednio na stronę, podobnie jak przy użyciu helpera `Html.Action`. Dane z Web serwisu muszą zostać przetworzone na kod HTML, np. przy użyciu `jQuery`, aby mogły być wyświetlane na stronie.

Użycie attribute routingu

Skorzystamy z *attribute routing*, aby uprościć ścieżkę do danych kategorii zwracanych w postaci JSON.

Aby włączyć *attribute routing*, należy dodać następującą linijkę w pliku *RouteConfig* w metodzie *RegisterRoutes()*:

```
routes.MapMvcAttributeRoutes();
```

Attribute routing musi być włączony przed innymi ścieżkami, ponieważ wybierana jest pierwsza pasująca ścieżka.

Kod metody RegisterRoutes() po zmianach wygląda jak poniżej:

```
public static void RegisterRoutes(RouteCollection routes)
{
    routes.IgnoreRoute("{resource}.axd/{*pathInfo}");

    routes.MapMvcAttributeRoutes();

    routes.MapRoute(
        name: "Default",
        url: "{controller}/{action}/{id}",
        defaults: new { controller = "Home", action = "Index", id =
            UrlParameter.Optional }
    );
}
```

Mając włączony *attribute routing*, dodaj atrybut [Route("JSON")] do akcji KategorieWJson z kontrolera KategoriaController.

Kod akcji z atrybutem:

```
[Route("JSON")]
public ActionResult KategorieWJson()
{
    var kategorie = _repo.PobierzKategorie().AsNoTracking();
    return Json(kategorie, JsonRequestBehavior.AllowGet);
}
```

Od teraz akcja KategorieWJson jest dostępna pod konkretnym adresem URL (rysunek 8.129).



Rysunek 8.129. Akcja KategorieWJson po włączeniu attribute routingu

Przed zastosowaniem *attribute routingu* metoda była dostępna pod adresem URL (rysunek 8.130):



Rysunek 8.130. Akcja KategorieWJson przed włączeniem attribute routingu

Jeśli sprawdzisz stary adres URL, pojawi się błąd 404 (rysunek 8.131).

Rysunek 8.131.

Brak strony



Zarządzanie relacją „wiele do wielu” i autocomplete

Aplikacja jest już praktycznie gotowa i ma zaimplementowane prawie wszystkie możliwe scenariusze, jednak brakuje zarządzania powiązaniem „wiele do wielu” pomiędzy kategorią a ogłoszeniem. To zawsze problematyczne powiązanie, ponieważ jeśli jest dużo kategorii, to po wczytaniu wszystkich powstalaby bardzo dłużna lista. Można wyświetlić checkboxy do zaznaczania, jednak wtedy trudno będzie znaleźć wybraną kategorię. Można wykorzystać podpowiadanie na podstawie tego, co się wpisze (ang. *Autocomplete*), podobnie jak robi to np. wyszukiwarka Google. Dane będą pobierane za pomocą jQuery i przesyłane w postaci JSON.

Dodaj dodatkowy przycisk w widoku MojeOgłoszenia (rysunek 8.132):

```
<br />@Html.ActionLink("Dodaj kategorie ", "DodajKategorie", new  
    { id = item.Id }, new { @class = "btn btn-success" })
```

Lista Moich Ogłoszeń				
Dodaj nowe ogłoszenie				
UżytkownikaId	Treść ogłoszenia:	Tytuł ogłoszenia:	Data dodania:	
6df2d16d-4ea-a4c4e-acb8-70f832b89cd3	nfgnf	nfghn	2014-07-02	Szczegóły Edytuj Dodaj kategorie Usuń
6df2d16d-4ea-a4c4e-acb8-70f832b89cd3	tut	Tutu	2014-07-02	Szczegóły Edytuj Dodaj kategorie Usuń
6df2d16d-4ea-a4c4e-acb8-70f832b89cd3	tyiti	yyit	2014-07-02	Szczegóły Edytuj Dodaj kategorie Usuń

Rysunek 8.132. Strona po dodaniu przycisku

Dodatek na AspNetMvc.pl

Dalsza część rozwiązania jest już dość mocno skomplikowana i nie nadaje się do książki, ponieważ składa się z setek linii kodu zarówno w HTML, CSS, jQuery, JavaScript, jak i po stronie serwera/aplikacji MVC, czyli w języku C#. Rozwiązanie będzie dostępne dla wszystkich zainteresowanych w formie PDF-a na stronie internetowej *AspNetMvc.pl*¹² jako dodatek do książki oraz na serwerze FTP wydawnictwa.

¹² Na stronie będzie można również zadawać pytania i pisać uwagi odnośnie do książki. W miarę możliwości będziemy się starali udzielać odpowiedzi na zadawane pytania.

Etap 8. Podsumowanie

W książce nie opisano wzorca UoW (ang. *Unit of Work*) ani repozytorium generycznego (przez niektórych uważanego za antywzorzec). Repozytorium generyczne polega na tworzeniu wspólnych metod, które powtarzają się dla każdego repozytorium, a więc np. Dodaj, Usuń i Edytuj, dzięki wykorzystaniu metod, klas i interfejsów generycznych. Wzorzec UoW polega na przeniesieniu kontekstu, tak aby był wspólny dla wszystkich repozytoriów. Przykładowo wywołuje się metody z repozytorium Ogłoszenie, następnie z repozytorium Kategoria, a dopiero później zapisuje zmiany jako jedną transakcję.

Jeśli nie chcesz tłumaczyć za każdym razem wygenerowanych widoków i kontrolerów, można poprawić szablony, na podstawie których generowane są pliki. Pliki z szablonami znajdują się w katalogu:

```
C:\Program Files (x86)\Microsoft Visual Studio  
12.0\Common7\IDE\Extensions\Microsoft\Web\Mvc\Scaffolding\Templates
```

Jeśli przyłączysz pliki z szablonami do projektu, to będą one zmienione tylko w tym projekcie, natomiast jeśli zmienisz je globalnie, to będą zmienione dla wszystkich projektów.

W omówionym projekcie zostały utworzone testy. W tak małym projekcie testy nie mają sensu, dlatego nie korzystaliśmy z nich, jednak można dopisać własne testy i wypróbować ich działanie.

Schemat przedstawiający zależność pomiędzy kontekstem bezpośrednio w kontrolerze, repozytorium i UoW został zaprezentowany na rysunku 8.133.

Etap 9. Dodatek

— tworzenie modelu dla podejścia Model First

W tym dodatku zaprezentowano kolejne kroki tworzenia modelu za pomocą Entity Designera i podejścia *Model First*.

Aby dodać nowy model, kliknij prawym przyciskiem myszy folder *Models* i wybierz *Add/ADO.NET Entity DataModel*, jak na rysunku 8.134.

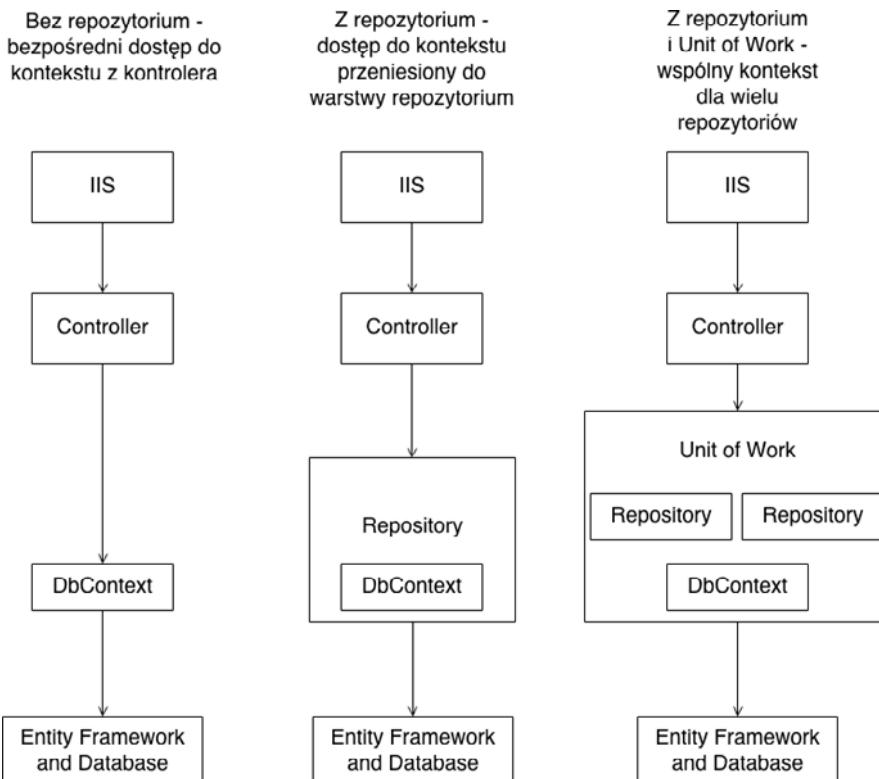
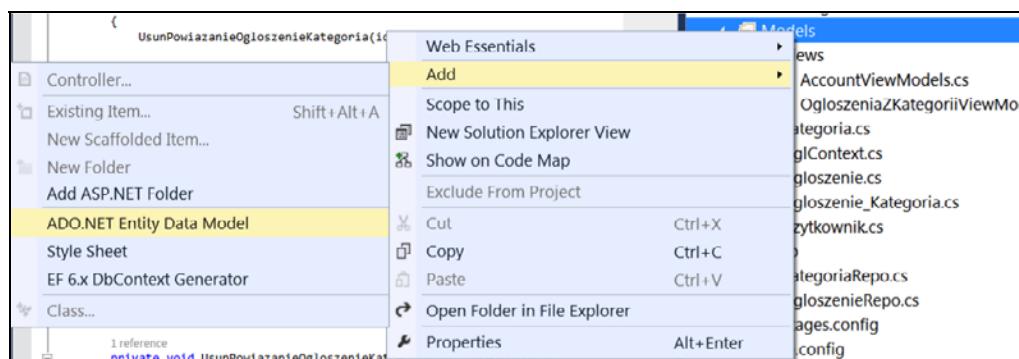
W kolejnym kroku wybierz *Empty EF Designer model* (rysunek 8.135).

Następnie dodaj encję. Kliknij prawym przyciskiem myszy i wybierz *Add New/Entity...* (rysunek 8.136).

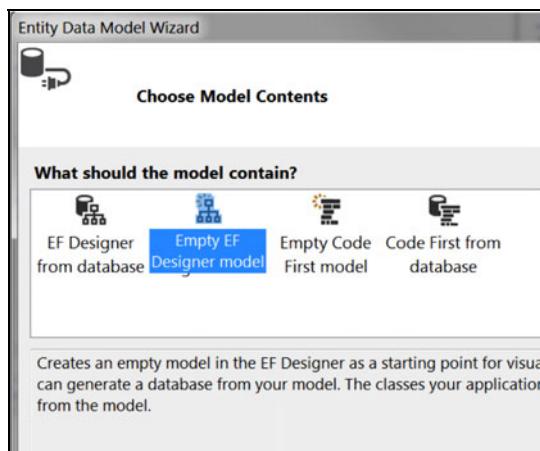
W kolejnym oknie wybierz nazwę klasy, nazwę tabeli w bazie danych, typ bazowy, jeśli chcesz wykorzystać dziedziczenie w bazie danych, oraz typ klucza podstawowego (rysunek 8.137).

Po kliknięciu przycisku *OK* otrzymasz encję, jak pokazano na rysunku 8.138.

W ten sam sposób dodaje się tabele dla ogłoszeń.

**Rysunek 8.133.** Zależności pomiędzy kontekstem w kontrolerze, repozytorium i UoW**Rysunek 8.134.** Dodawanie modelu

Rysunek 8.135.
Wybór treści modelu



Rysunek 8.136.
Dodawanie encji



Następnie dodaj powiązanie pomiędzy tabelami. Kliknij prawym przyciskiem myszy i wybierz *Add New/Association...* (rysunek 8.139).

W kolejnym oknie masz do wyboru typ powiązania i jego nazwę. Możesz również zaznaczyć, czy chcesz tworzyć relację FK (rysunek 8.140).

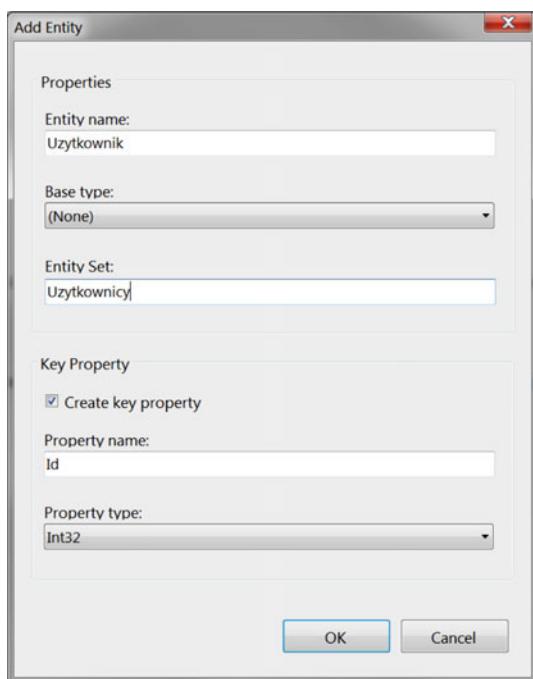
Po utworzeniu relacji diagram wygląda jak na rysunku 8.141.

Aby dodać nową kolumnę do tabeli, kliknij tabelę prawym przyciskiem myszy i wybierz *Add New/Scalar Property* (rysunek 8.142).

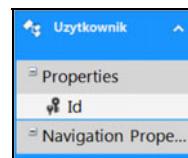
Następnie wpisz nazwę właściwości (rysunek 8.143).

Aby ustawić typ dodanych właściwości, otwórz okno *Properties* (rysunek 8.144) i w polu *Type* wybierz pożądany typ danych.

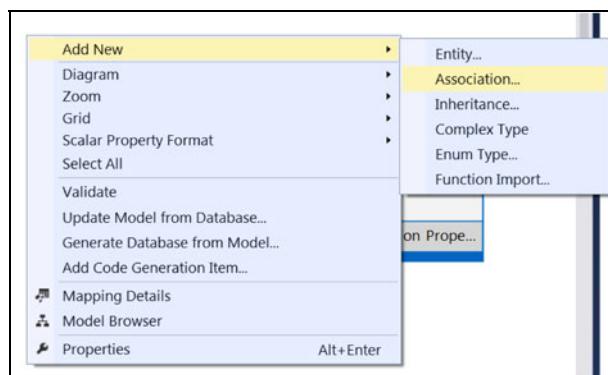
Rysunek 8.137.
Wybór opcji



Rysunek 8.138.
Encja Uzytkownik



Rysunek 8.139.
Powiązania pomiędzy tabelami



Rysunek 8.140.

Parametry powiązania pomiędzy tabelami

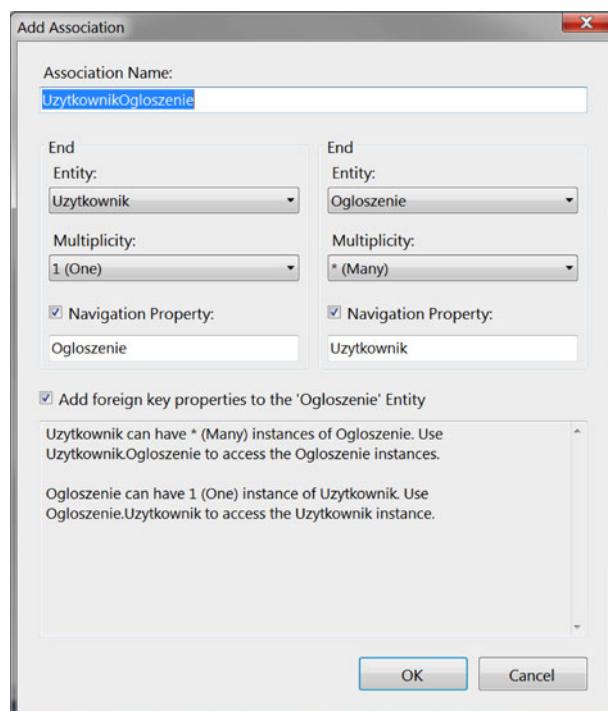
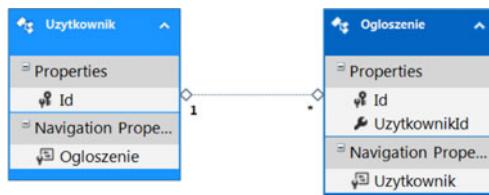
**Rysunek 8.141.**

Diagram ERD

**Rysunek 8.142.**

Dodawanie nowej kolumny do tabeli

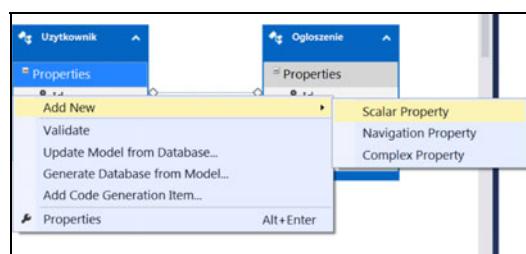
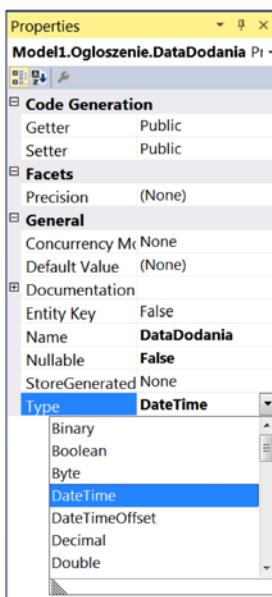
**Rysunek 8.143.**

Diagram po wpisaniu nazwy właściwości



Rysunek 8.144.
Okno Properties

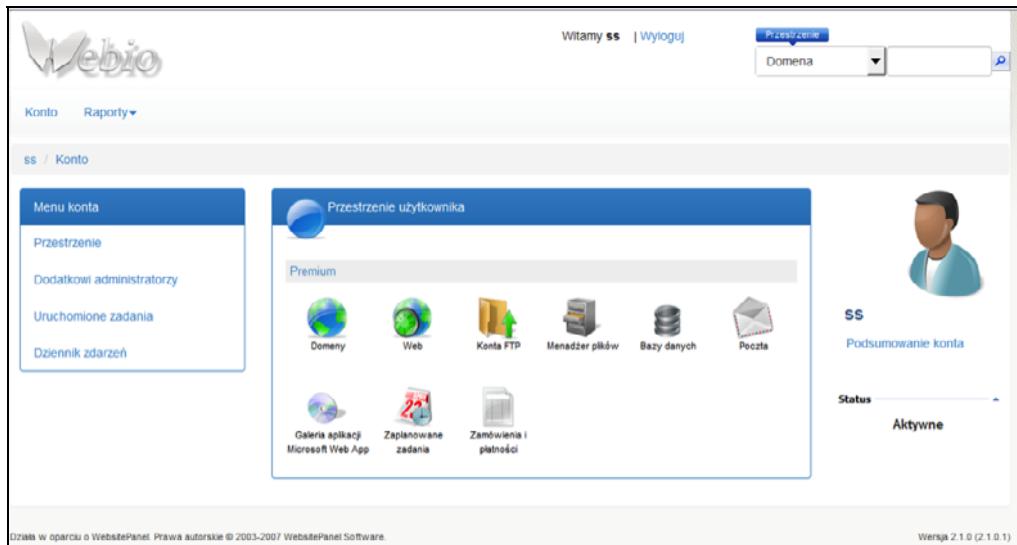


Publikacja systemu na zewnętrzny serwerze hostingowym

W tym podrozdziale zostaną opisane kolejne kroki uruchamiania systemu na hostingu Webio. Jest to najpopularniejszy polecany przez Microsoft polski hosting obsługujący aplikacje ASP.NET. Minimalnym planem hostingowym, który pozwala na uruchomienie aplikacji korzystającej z bazy danych, jest pakiet *Podstawowy*. Najtańszy pakiet, *Startowy*, nie pozwala na tworzenie bazy danych, która jest wymagana do działania systemu. Wersja bazy danych SQL Server zainstalowanej na hostingu w dniu publikacji książki to Microsoft SQL Server 2012. Webio udostępnia 14-dniowy okres testowy o parametrach konta podstawowego, a więc z bazą danych SQL Server. Po zakończeniu okresu testowego można opłacić serwer na kolejny okres lub zrezygnować z usługi. Do utworzenia przykładowej aplikacji i uruchomienia na hostingu 14 dni w zupełności wystarczy.

Na rysunku 8.145 przedstawiono panel kontrolny użytkownika Webio.

W *Menu konta* w zakładce *Przestrzenie* znajduje się lista wykupionych pakietów hostingu. Po kliknięciu nazwy przestrzeni otworzy się *Menu przestrzeni* (rysunek 8.146). W *Menu przestrzeni* znajdują się następujące elementy: *Domeny*, *Witryny*, *Adresy IP*, *Konta FTP*, *Poczta*, *Baza danych*, *Udostępnione foldery SSL*, *Statystyki witryn*, *ODBC DSNs*, *Menadżer plików*, *Galeria aplikacji Microsoft Web App*, *Zaplanowane zadania* oraz *System Hard Quota*. Pierwszą rzeczą, którą trzeba zrobić, aby uruchomić system na tym hostingu, jest dodanie domeny dla aplikacji.



Rysunek 8.145. Panel użytkownika Webio

Dodawanie domeny

Aby dodać domenę do przestrzeni, przejdź na zakładkę *Domeny*, a następnie po odświeżeniu się strony kliknij przycisk *Dodaj domenę* (rysunek 8.147).

W kolejnym oknie wybierz typ tworzonej domeny (rysunek 8.148). Do wyboru są dwie opcje:

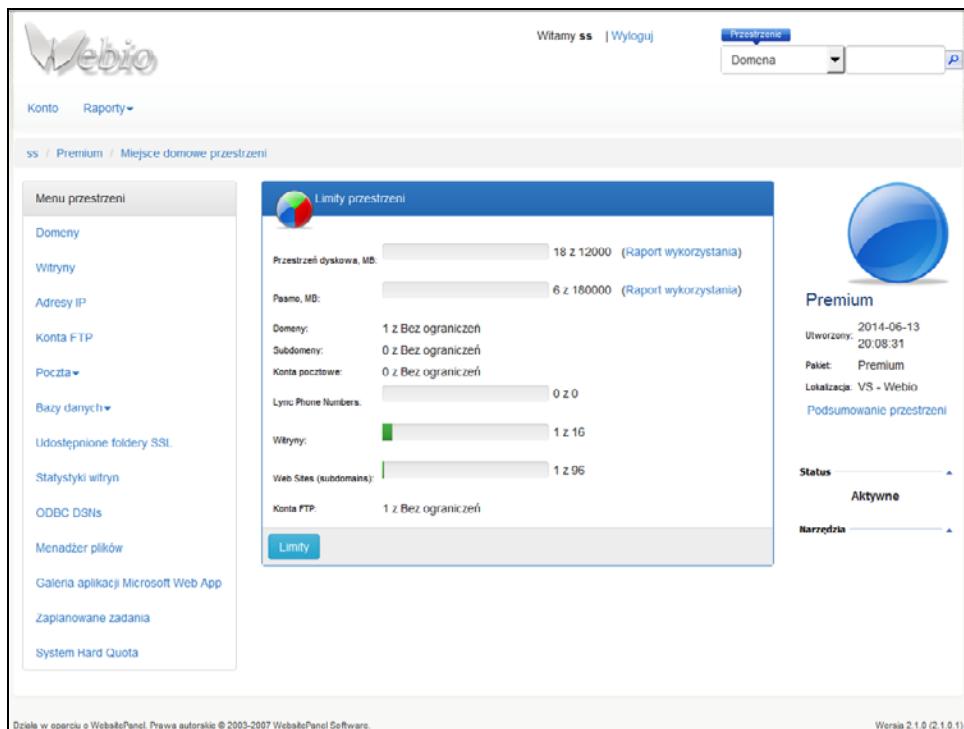
- ◆ *Domena* — tworzy nową domenę;
- ◆ *Poddomena* — tworzy poddomenę w wybranej domenie głównej.

W tym przypadku wybierz opcję *Domena*, ponieważ tworzona jest nowa domena.

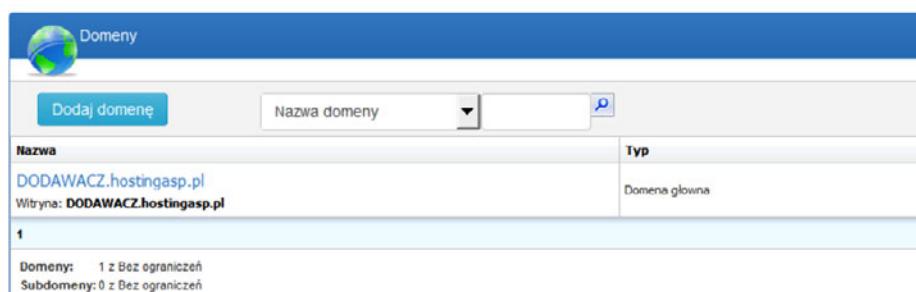
W kolejnym kroku (rysunek 8.149) wpisz nazwę domeny (w tym przypadku będzie to domena *AspNetMvc.pl*). Tu także jest kilka opcji do wyboru:

- ◆ *Utwórz witrynę* — tworzy witrynę przypisaną do domeny;
- ◆ *Wskazuj na istniejącą witrynę* — pozwala na utworzenie przekierowania z tej domeny na stronę zamieszczoną w innej domenie;
- ◆ *Dodaj strefę DNS dla tej domeny* — tworzy strefę DNS na serwerze Webio;
- ◆ *Utwórz alias tymczasowy* — pozwala na dostęp do witryny poprzez domenę tymczasową, czyli w tym wypadku *aspnetmvc.pl.hostingasp.pl*.

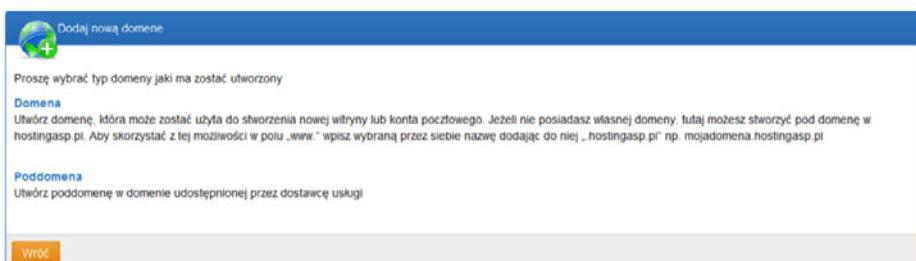
Zaznacz opcje *Utwórz witrynę* i *Dodaj strefę DNS dla tej domeny*, a następnie naciśnij przycisk *Dodaj domenę*. Po dodaniu domeny będzie ona widoczna na liście domen (rysunek 8.150).



Rysunek 8.146. Menu przestrzeni Webio



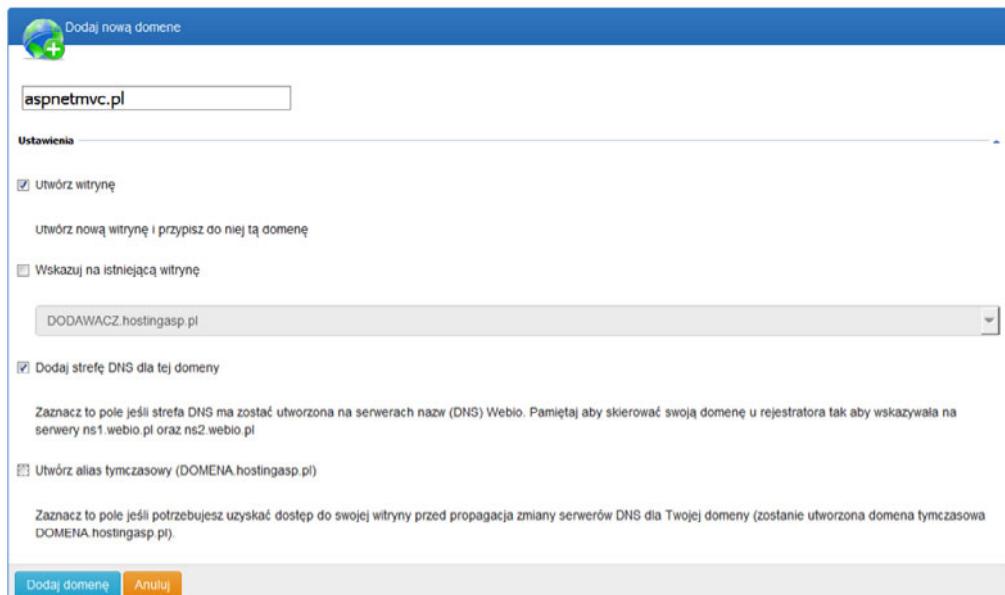
Rysunek 8.147. Okno dodawania domeny w panelu administratora



Rysunek 8.148. Okno wyboru typu domeny



Po przekierowaniu domeny `aspnetmvc.pl` u rejestatora na serwery nazw Webio (`ns1.webio.pl`, `ns2.webio.pl`) adresy DNS na serwerach zostaną zaktualizowane dopiero po pewnym czasie, przez co nie będzie możliwości natychmiastowego przeglądania witryny z adresu `aspnetmvc.pl`. Aby mieć możliwość przeglądania witryny od razu po przekierowaniu domeny, można skorzystać z aliasu tymczasowego lub skierować domenę na IP w systemie Windows. Aby przypisać domenę do IP, należy dodać wpis w formacie IP DOMENA (np. `91.219.122.70 aspnetmvc.pl`) w pliku `hosts` znajdującym się w folderze `C:/Windows/System32/drivers/etc/` (w systemie Windows).



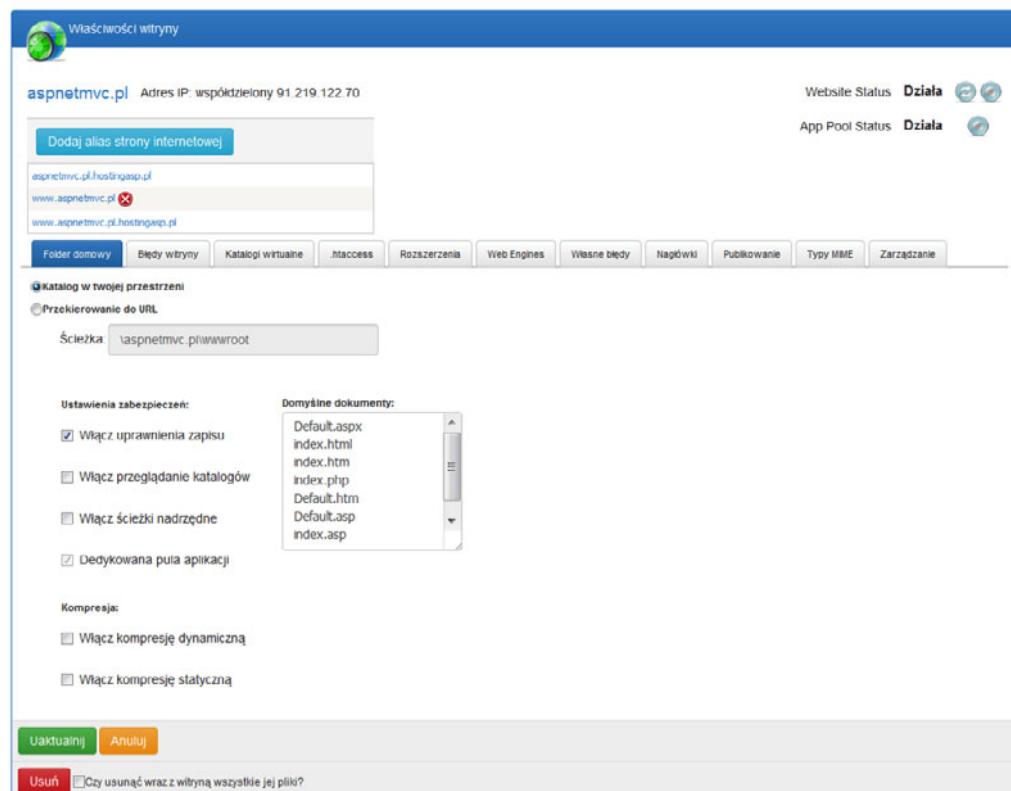
Rysunek 8.149. Nazwa domeny

Domeny	
Dodaj domenę	
Nazwa	TYP
aspnetmvc.pl Witryna: <code>aspnetmvc.pl</code>	Domena główna

Rysunek 8.150. Lista dostępnych domen

Konfiguracja witryny

Po dodaniu domeny wraz z witryną w przestrzeni Webio przejdź na zakładkę witryny i kliknij nazwę nowo dodanej witryny. Teraz możesz już zarządzać witryną poprzez panel (rysunek 8.151).



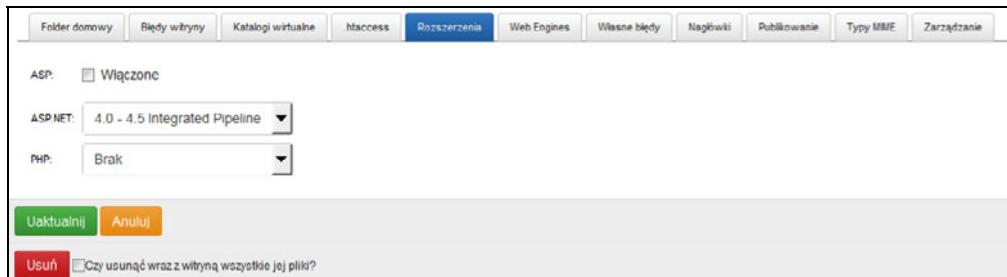
Rysunek 8.151. Okno zarządzania witryną

Na głównej stronie panelu zarządzania witryną można wybrać *Folder domowy*, w jakim będzie się znajdować aplikacja, a także ścieżki nadzędne, przeglądanie katalogów, uprawnienia do zapisu czy rodzaj uwierzytelniania. Dodatkowo można określić, jakie pliki będą domyślnymi plikami przy dostępie do witryny. W tym przypadku pozostawimy standardowe ustawienia.

Aby uruchomić aplikację ASP.NET MVC5, wymagane jest włączenie trybu *4.0-4.5 Integrated Pipeline*. Aby to zrobić, przejdź do zakładki *Rozszerzenia* (rysunek 8.152), gdzie z listy rozwijanej *ASP.NET* wybierz wcześniej wspomniany tryb. WebGo oferuje także obsługę PHP, jednak nie jest nam ona potrzebna, dlatego można ją wyłączyć.

W pozostałych zakładkach można przeglądać logi z błędami witryny, a także uruchomić tryb *Web Deploy*, który pozwala na łatwe przenoszenie aplikacji z Visual Studio wprost na serwer. Pozostałe zakładki są w tym przypadku mniej istotne, dlatego nie będą opisywane.

Aby uruchomić *Web Deploy*, wystarczy w zakładce *Publikowanie* wpisać nazwę użytkownika, hasło i kliknąć przycisk *Włącz* (rysunek 8.153).



Rysunek 8.152. Okno wyboru trybu działania witryny



Rysunek 8.153. Uruchamianie Web Deploy

Po włączeniu Web Deploy można pobrać plik, na podstawie którego Visual Studio uruchomi procedurę publikacji bez podawania hasła i nazwy użytkownika. Kliknij *Download Publishing Profile for this web site* i zapisz plik na dysku (rysunek 8.154). W dowolnym momencie można wyłączyć tę opcję lub zmienić hasło.



Rysunek 8.154. Włączony Web Deploy

Tworzenie bazy danych

Aby utworzyć bazę danych, na głównej stronie panelu kliknij ikonę *Bazy danych* i z menu rozwijanego wybierz *SQL Server 2012*. W nowym oknie (rysunek 8.155) masz do wyboru utworzenie bazy danych oraz użytkownika.

The screenshot displays two windows side-by-side. The top window is titled 'SQL Server 2012 Databases' and shows a list of databases. It has a button 'Utwórz bazę danych' (Create database) and a search bar 'Nazwa elementu'. One database, 'ss_linkidb', is listed with a count of 1. The bottom window is titled 'SQL Server 2012 Logins' and shows a list of logins. It also has a button 'Utwórz użytkownika' (Create user) and a search bar 'Nazwa elementu'. One login, 'ss_linkidb', is listed with a count of 1. Both windows have a status bar at the bottom indicating 'Databases: 1 z 16' or 'Users: 1 z 16' respectively.

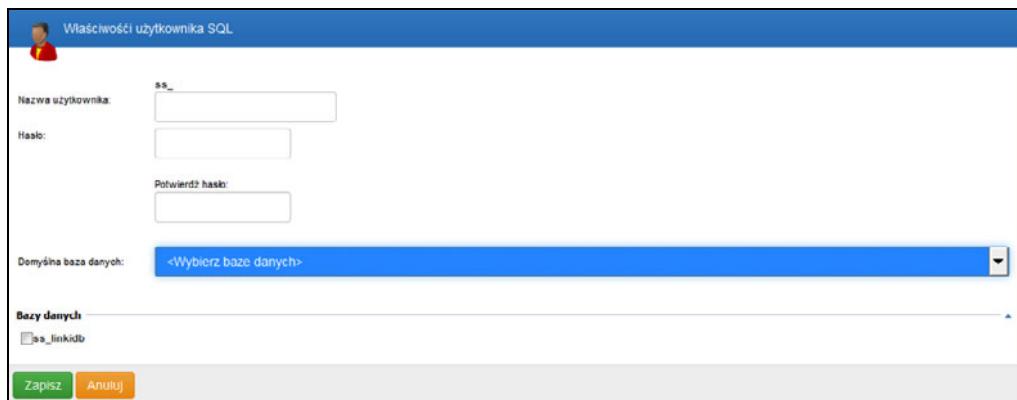
Rysunek 8.155. Okno zarządzania bazą danych i użytkownikami bazy danych

Podczas tworzenia bazy danych (rysunek 8.156) wpisz nazwę bazy oraz wybierz użytkownika, który ma do niej dostęp (jeśli został jakiś wcześniej dodany).

The screenshot shows the 'Właściwości bazy danych SQL' (SQL Database Properties) dialog box. It contains fields for 'Nazwa bazy danych:' (Database name: ss_linkidb) and a list of 'Użytkownicy' (Users) where 'ss_linkidb' is selected. Under 'Pliki bazy danych:' (Data files), there is a dropdown menu. At the bottom are 'Zapisz' (Save) and 'Anuluj' (Cancel) buttons.

Rysunek 8.156. Okno tworzenia bazy danych

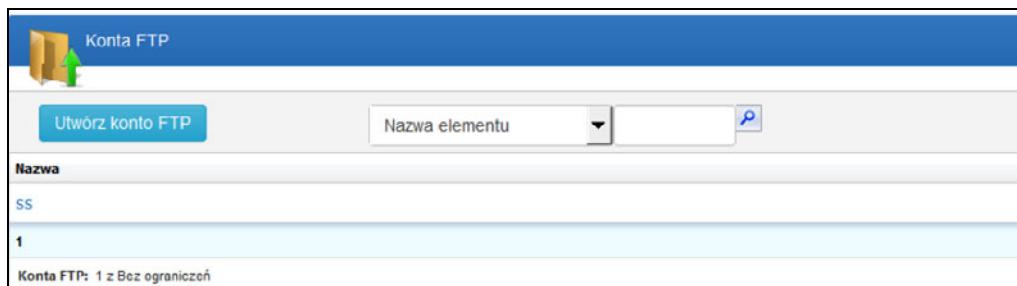
Analogicznie podczas tworzenia użytkownika (rysunek 8.157) wpisz nazwę i hasło oraz wybierz bazę danych, do której będzie miał dostęp (jeśli została jakaś wcześniej dodana).



Rysunek 8.157. Okno tworzenia użytkownika bazy danych

Tworzenie konta FTP

Konto FTP jest niezbędne, aby przesyłać pliki na serwer w standardowy sposób, np. za pomocą Total Commandera. Jeśli nie będziesz przesyłać żadnych dodatkowych plików przez FTP, to nie ma potrzeby tworzenia tego konta. Aby utworzyć konto FTP, na stronie głównej panelu wybierz zakładkę *Konta FTP*. W nowo otwartym oknie (rysunek 8.158) kliknij przycisk *Utwórz konto FTP*.

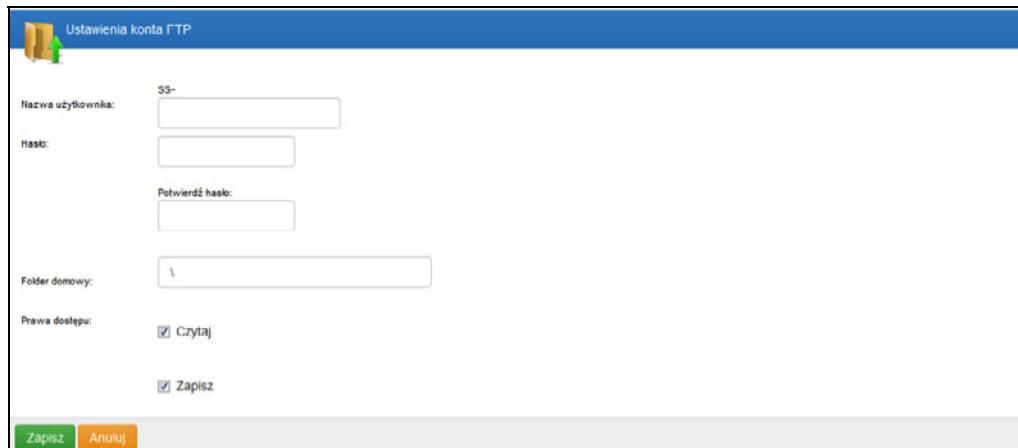


Rysunek 8.158. Okno zarządzania kontami FTP

Kolejnym krokiem (rysunek 8.159) jest wprowadzenie nazwy użytkownika, hasła, określenie folderu domowego (miejscia, w którym będzie się znajdował użytkownik po zalogowaniu do serwera) oraz praw dostępu.

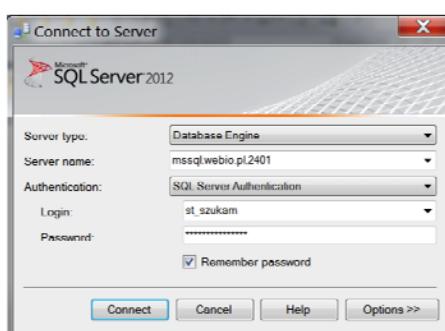
Połączenie z bazą danych poprzez SQL Server Management Studio

Gdy utworzysz użytkownika oraz bazę danych, będziesz się mógł z nią połączyć za pomocą programu do zarządzania bazą danych Microsoft SQL Server Management Studio (rysunek 8.160). Aby to zrobić, podaj nazwę serwera, która jest znana po zakupieniu hostingu, oraz login i hasło do bazy danych. W przypadku braku Management Studio można skorzystać z Visual Studio, gdzie w oknie *Server Explorer* jest dostęp do bazy danych.



Rysunek 8.159. Okno tworzenia konta FTP

Rysunek 8.160.
Okienko połączenia
z SQL Server
Management Studio



Po skonfigurowaniu hostingu można przejść do Visual Studio i zacząć wgrywać aplikację na serwer.

Wdrażanie aplikacji na serwer za pomocą Microsoft Visual Studio

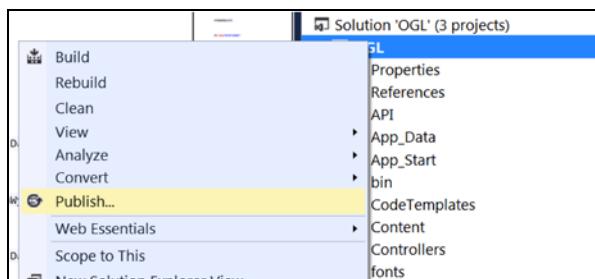
Visual Studio 2013 pozwala wdrożyć aplikację na kilka sposobów. Można wysłać pliki bezpośrednio na serwer poprzez połączenie FTP, utworzyć paczkę plików i wgrać samodzielnie lub skorzystać z narzędzia Web Deploy, które automatycznie wdraża aplikację i jest najwygodniejsze. Aby opublikować witrynę na hostingu, kliknij prawym przyciskiem myszy nazwę projektu i wybierz opcję *Publish...* (rysunek 8.161).

Pokaże się okno (rysunek 8.162) pozwalające na wybór miejsca, w którym chce się opublikować witrynę. Aby skorzystać z Web Deploy, wybierz opcję *Import*, która automatycznie zaimportuje ustawienia na podstawie pliku, który pobrano przy uruchamianiu Web Deploy.

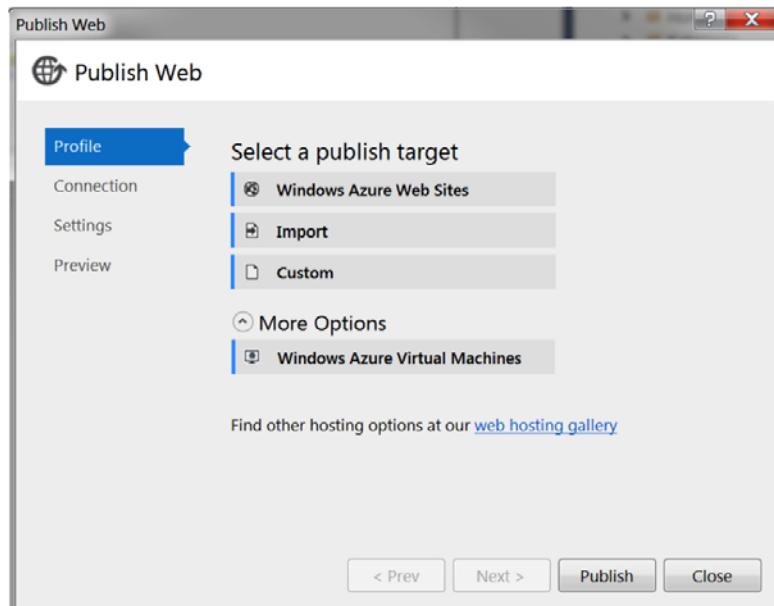
Po wybraniu pliku pokaże się okno z ustawieniami pobranymi z pliku (rysunek 8.163). Aby rozpocząć przesyłanie aplikacji na serwer, wystarczy kliknąć przycisk *Publish*.

Rysunek 8.161.

Publikowanie witryny na hostingu zewnętrznym

**Rysunek 8.162.**

Wybór miejsca publikacji



W trakcie przesyłania aplikacji na serwer pokaże się okno *Web Publish Activity*, w którym można śledzić postęp wysyłania plików (rysunek 8.164).

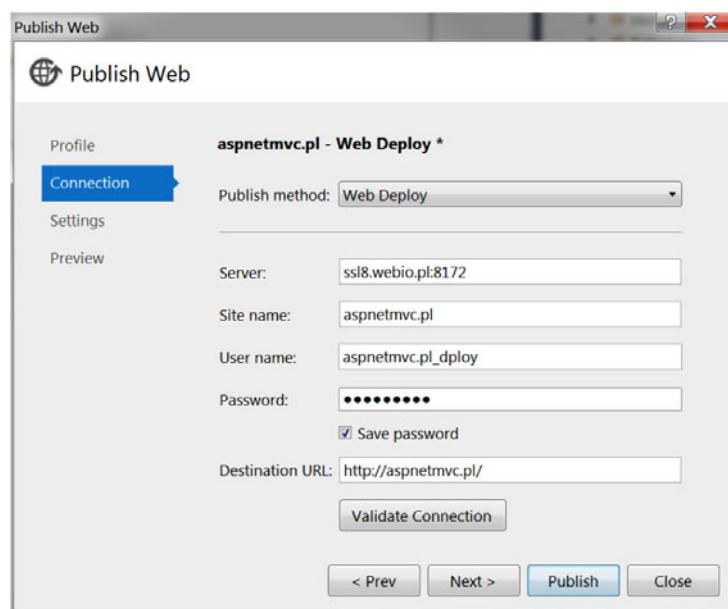
Po przesłaniu wszystkich plików automatycznie zostanie otworzona domyślna przeglądarka i wyświetlona zostanie przesyłana strona. Po przejściu do zakładki korzystającej z bazy danych wystąpi błąd, ponieważ nie zmieniliśmy *ConnectionString* w pliku *Web.config*. *ConnectionString* zawiera informacje o adresie serwera, użytkowniku i haśle do bazy danych. Domyślnie Visual Studio korzysta z lokalnej bazy danych *LocalDb*, która nie jest dostępna na serwerze.

Przykład *ConnectionString* dla bazy SQL Server:

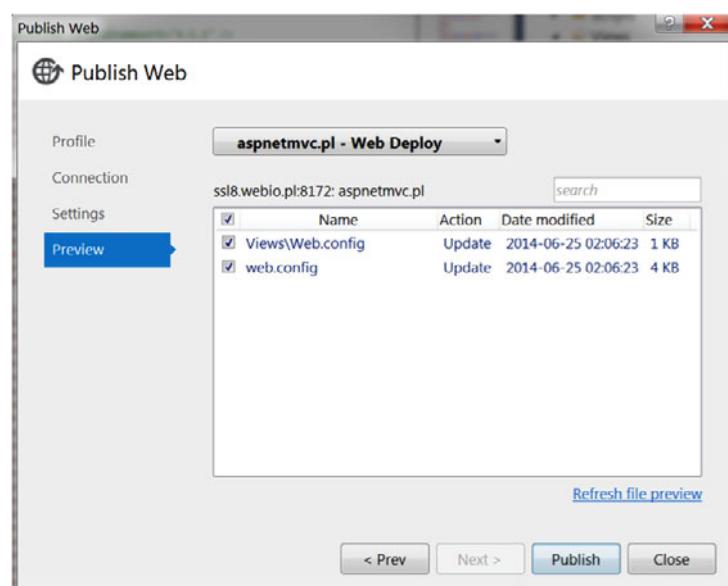
```
<connectionStrings>
  <add name="DefaultConnection"
    connectionString="Server=mssql15.webio.pl,2401;Database=NAZWABAZY;
    Uid=USER;Password=HASŁO;" providerName="System.Data.SqlClient"/>
</connectionStrings>
```

Rysunek 8.163.

Rozpoczęcie publikacji

**Rysunek 8.164.**

Postęp publikacji



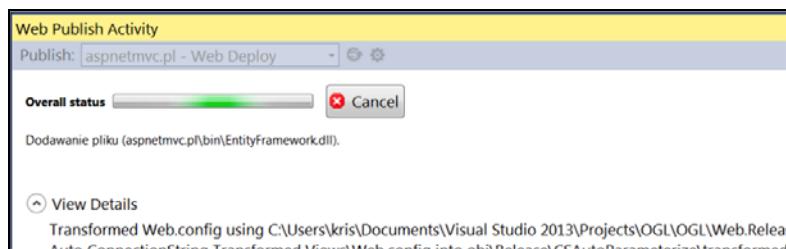
Po prawidłowym uruchomieniu aplikacji na serwerze i zmianieConnectionString na ten z serwera zarówno w internecie w domenie *aspnetmvc.pl*, jak i po uruchomieniu aplikacji w Visual Studio będzie realizowany dostęp do tej samej bazy danych (już nie lokalnej, tylko tej z hostingu).

Jeśli wprowadzisz zmiany w plikach i będziesz chciał opublikować nową wersję na serwerze, narzędzie Web Deploy nie będzie przesyłać całej aplikacji, jak miało to miejsce za pierwszym razem, tylko te pliki, które edytowano od czasu ostatniego

przesyłania na serwer. Po kliknięciu *Publish* pokaże się okno, w którym można otworzyć podgląd oraz sprawdzić, jakie pliki były edytowane i zostaną przesłane na serwer (rysunek 8.165).

Rysunek 8.165.

Edytowane pliki



Dodatek A

Zasady i metodologie w programowaniu

Programowanie powinno być usystematyzowane. Służą do tego zasady, których warto przestrzegać w czasie pisania kodu.

Zasady

SOLID

SOLID to zbiór zasad opisujący podstawowe założenia programowania obiektowego. Zasady SOLID składają się z pięciu założeń:

- ◆ S — zasada pojedynczej odpowiedzialności — SRP (ang. *Single Responsibility Principle*),
- ◆ O — zasada otwarte-zamknięte — OCP (ang. *Open-Closed Principle*),
- ◆ L — zasada podstawienia Liskov — LSP (ang. *Liskov Substitution Principle*),
- ◆ I — zasada separacji interfejsów — ISP (ang. *Interface Segregation Principle*),
- ◆ D — zasada odwrócenia zależności — DIP (ang. *Dependency Inversion Principle*).

Zasada pojedynczej odpowiedzialności (SRP)

Zasada pojedynczej odpowiedzialności mówi, że każda klasa (obiekt) powinna odpowiadać za jak najmniejszy fragment logiki programu. Zasada SRP definiuje zbyt dużą odpowiedzialność jako powód do zmiany (ang. *Reason to Change*). Jeżeli jedna klasa wykonuje obliczenia oraz wyświetla wynik obliczeń, nie spełnia zasady SRP. Odpowiada za dwa procesy, a więc mogą wystąpić dwa powody do zmian. Może się zmienić zasada obliczania lub sposób wyświetlania wyniku. Zasada pojedynczej odpowiedzialności

nakazuje w tym wypadku oddzielenie części obliczeń od części wyświetlania wyniku. Oba procesy powinny być zaimplementowane w dwóch osobnych klasach, które komunikują się między sobą za pomocą publicznych interfejsów. Interfejsy pozwalają na łatwiejsze zarządzanie powiązaniami pomiędzy klasami. W ten sposób maleje ilość odpowiedzialności dla każdej klasy oraz rośnie prawdopodobieństwo ponownego użycia każdej z klas.

Zasada otwarte-zamknięte (OCP)

Zasada otwarte-zamknięte mówi, że klasy, moduły, funkcje itp. powinny być otwarte na rozszerzenia, a zamknięte na modyfikacje. Powinna zatem istnieć możliwość uzywania nowych funkcjonalności poprzez rozszerzenie elementu, a nie zmianę już istniejącego. Zasada ta jest bardzo ważna w czasie zmian w kodzie produkcyjnym, ponieważ zmiana kodu w jednym miejscu może powodować problemy w kilku innych miejscach dostosowanych do wcześniejszej wersji elementu. Dzięki trzymaniu się tej zasady zwiększa się modularność systemu i zmniejsza ryzyko pojawienia się błędów w czasie rozbudowy programu.

Zasada podstawienia Liskov (LSP)

Zasada podstawienia Liskov mówi, że: „Funkcje, które używają wskaźników lub referencji do klas bazowych, muszą być w stanie używać również obiektów klas dziedziczących po klasach bazowych, bez dokładnej znajomości tych obiektów”¹. Mówiąc skrótnie, klasa dziedzicząca po klasie bazowej nie powinna zmieniać jej funkcjonalności, tylko rozszerzać możliwości. Najpopularniejszym przykładem na złamanie tej zasady jest dziedziczenie klasą Kwadrat po klasie Prostokat (listing A.1).

Listing A.1. Kod klas Kwadrat i Prostokat

```
public class Prostokat
{
    public virtual int Szerokosc { get; set; }
    public virtual int Wysokosc { get; set; }
}

public class Kwadrat : Prostokat
{
    public override int Wysokosc
    {
        get { return base.Wysokosc; }
        set { UstawBoki(value); }
    }

    public override int Szerokosc
    {
        get { return base.Szerokosc; }
        set { UstawBoki(value); }
    }
}
```

¹ Robert C. Martin, *The Liskov Substitution Principle*, <http://www.objectmentor.com/resources/articles/lsp.pdf>.

```
// Równe boki dla kwadratu
private void UstawBoki(int value)
{
    base.Wysokosc = value;
    base.Szerokosc = value;
}
```

Dla kwadratu wywołuje się metodę `UstawBoki()`, która ustawia obydwa boki na tę samą długość (listing A.2).

Listing A.2. Kod głównej klasy programu

```
public class Program
{
    public static void Main()
    {
        // Dla kwadratu wypisze 10×10
        Prostokat kwadrat = new Kwadrat();
        kwadrat.Szerokosc = 4;
        kwadrat.Wysokosc = 10;

        Console.WriteLine("{0} x {1}",
                           kwadrat.Szerokosc,
                           kwadrat.Wysokosc);

        // Dla prostokąta wypisze 4×10
        Prostokat prostokat = new Prostokat();
        prostokat.Szerokosc = 4;
        prostokat.Wysokosc = 10;

        Console.WriteLine("{0} x {1}",
                           prostokat.Szerokosc,
                           prostokat.Wysokosc);

        Console.ReadLine();
    }
}
```

Program w zależności od typu utworzonego obiektu zwraca inne wyniki pomimo ustawienia w ten sam sposób długości boków. Jest to przykład zmiany zachowania klasy bazowej. Klasa `Kwadrat` dziedziczy po klasie `Prostokat`, a pomimo to w zależności od tego, do której klasy posiada referencje, otrzymuje inny wynik, chociaż typ obiektu jest cały czas taki sam, czyli `Prostokat`.

Zasada separacji interfejsów (ISP)

Zasada separacji interfejsów mówi, że tworzone interfejsy muszą być odpowiedzialne za jak najmniejszą funkcjonalność. Jeśli na przykład jeden interfejs opisujący metody dostępne w klasie chce rozszerzyć funkcjonalność aplikacji o nowy moduł, który wymaga tylko części metod dostępnych w interfejsie, należy zaimplementować wszystkie metody z interfejsu, mimo że tylko część z nich będzie wykorzystywana. Taki interfejs nazywa

się „tłustym” (ang. *Fat*) lub „zanieczyszczonym” (ang. *Polluted*). „Tłuste” interfejsy nie są dobrym rozwiązańiem i mogą powodować niewłaściwe zachowanie systemu. Najpopularniejszym przykładem obrazującym problem „tłustych” interfejsów jest problem pracownika i robota (listingi A.3 i A.4). Jak widać, interfejs jest „tłustym” interfejsem, ponieważ robot nie musi jeść ani pić.

Listing A.3. Kod interfejsu

```
interface IPracownik
{
    void Pracuj();
    void Jedz();
    void Pij();
}
```

Listing A.4. Kod klas

```
class Pracownik : IPracownik
{
    public void Pracuj()
    {
    }
    public void Jedz()
    {
    }
    public void Pij()
    {
    }
}

class Robot : IPracownik
{
    public void Pracuj()
    {
    }
    public void Jedz()
    {
        throw new NotImplementedException();
    }
    public void Pij()
    {
        throw new NotImplementedException();
    }
}
```

Aby zapobiec „tłustym” interfejsom, stosuje się refaktoryzację kodu i rozbicie interfejsów na kilka mniejszych (listing A.5). Klasa Robot zamiast dziedziczyć po „tłustym” interfejsie i implementować niepotrzebne metody, dziedziczy tylko po jednym mniej rozbudowanym interfejsie. W przypadku klasy Pracownik dziedziczy ona po dwóch interfejsach.

Listing A.5. Kod po refaktoryzacji

```
interface IPracuj
{
    void Pracuj();
}

interface ISpozywaj
{
    void Jedz();
    void Pij();
}

class Pracownik : ISpozywaj, IPracuj
{
    public void Pracuj()
    {}
    public void Jedz()
    {}
    public void Pij()
    {}
}

class Robot : IPracuj
{
    public void Pracuj()
    {}
}
```

Zasada odwrócenia zależności (DIP)

Zasada odwrócenia zależności mówi, że:

- ♦ kod z warstw z wyższego poziomu nie powinien zależeć od kodu z niższych warstw; obie warstwy powinny być za to zależne od abstrakcji;
- ♦ abstrakcje nie powinny zależeć od szczegółów, a więc konkretnej implementacji; szczegółły, czyli implementacja, powinny zależeć od abstrakcji.

Głównym celem stosowania DIP jest oddzielenie wysokopoziomowych komponentów aplikacji od komponentów niskopoziomowych. Pomiędzy komponentami wysokopoziomowymi a niskopoziomowymi powinna się znajdować warstwa abstrakcji, pozwalająca na uniezależnienie warstw. Dzięki takiemu rozwiązaniu możliwa jest zamiana jednej warstwy bez wprowadzania zmian na innym poziomie. Przykładowo warstwa biznesowa aplikacji korzysta z warstwy dostępu do danych. Aby usunąć bezpośrednie zależności pomiędzy warstwami, część biznesowa nie może mieć referencji bezpośrednio do warstwy dostępu do danych. Referencje powinny być wstrzykiwane przez konstruktora, co w zależności od wstrzykniętego typu pozwala nam operować na innej warstwie dostępu do danych.

W przykładzie z listingu A.6 wstrzykujemy obiekt implementujący interfejs `IDataContext`. Nie ma znaczenia, jaki konkretnie jest to obiekt — może to być obiekt dostępu do bazy danych lub do plików tekstowych. Ważne jest tylko, aby implementował interfejs dostępu do danych.

Listing A.6. Przykładowy kod wykorzystujący wstrzyknięcie przez konstruktor

```
class Biznesowa
{
    private IDataProvider _context = null;
    public Biznesowa(IDataProvider context)
    {
        _context = context;
    }
    public void DodajDoBazy()
    {
        _context.Baza.Add(new Wpis());
    }
}
```

Obok wstrzykiwania zależności przez konstruktor istnieje również możliwość wstrzykiwania przez właściwości lub parametry metod.

GRASP

GRASP (ang. *General Responsibility Assignment Software Patterns*) to zbiór dziewięciu zasad określających, jaką odpowiedzialność powinno się przypisywać określonym obiektom i klasom w systemie. Każda zasada wyjaśnia inny problem dotyczący ustalania odpowiedzialności obiektu. Struktura GRASP wygląda następująco:

- ◆ *Creator* — kiedy obiekt powinien tworzyć inny obiekt?
- ◆ *Information Expert* — jak ustalać zakres odpowiedzialności obiektu?
- ◆ *Controller* — jaka część systemu odpowiedzialna za logikę biznesową programu powinna się komunikować z interfejsem użytkownika?
- ◆ *Low Coupling* — jak ograniczyć zakres zmian w systemie w momencie zmiany fragmentu systemu?
- ◆ *High Cohesion* — jak ograniczać zakres odpowiedzialności obiektu?
- ◆ *Polymorphism* — co zrobić, gdy odpowiedzialność różni się w zależności od typu?
- ◆ *Pure Fabrication* — komu delegować zadania, gdy nie można zidentyfikować, do kogo należy podany zbiór operacji?
- ◆ *Indirection* — jak zorganizować komunikację między obiektami, aby mocno ich nie wiązać?
- ◆ *Protected Variations* — jak przypisywać odpowiedzialność, aby zmiana w jednej części systemu nie powodowała niestabilności innych elementów?

Creator

Problem: określa, kiedy pierwszy obiekt powinien tworzyć inny obiekt.

Rozwiązanie: należy przypisać klasie B odpowiedzialność za tworzenie obiektów klasy A, gdy:

- ◆ klasa B zawiera (lub agreguje) obiekty klasy A,
- ◆ B „rejestruje życie” instancji klasy A,
- ◆ B blisko współpracuje z A,
- ◆ B ma dane inicjalizacyjne potrzebne przy tworzeniu A.

Information Expert

Problem: której klasie przypisać daną odpowiedzialność/zadanie?

Rozwiązanie: należy przypisać to zadanie tej klasie, która ma informacje niezbędne do tego, aby je móc wypełnić.

Komentarz: programista powinien delegować nową odpowiedzialność do klasy zawierającej najwięcej informacji potrzebnych do zrealizowania nowej funkcjonalności. Konieczne jest wcześniejsze określenie, jakie dane są niezbędne.

Controller

Problem: który obiekt poza GUI (interfejsem użytkownika) powinien obsługiwać żądania systemu?

Rozwiązanie: należy przydzielić odpowiedzialność obiektowi spełniającemu jeden z warunków:

- ◆ klasa reprezentuje cały system,
- ◆ klasa reprezentuje przypadek użycia systemu, w którym wykonywana jest dana operacja (NazwaHandler, NazwaController).

Komentarz: zadania interfejsu użytkownika powinny być obsługiwane przez kontroler. Kontroler powinien odebrać informacje od GUI, wykonać niezbędne operacje i zwrócić wynik do GUI. Bardzo dobrym przykładem na spełnienie tej zasady jest wzorzec MVC.

Low Coupling

Problem: jak ograniczyć zakres zmian w systemie w momencie zmiany fragmentu systemu?

Rozwiązanie: należy delegować odpowiedzialności, tak aby zachować jak najmniejszą liczbę powiązań pomiędzy klasami.

Komentarz: w przypadku wielu powiązań zmiana w jednej klasie powoduje wiele zmian w innych klasach. Klasy powinny być od siebie jak najbardziej niezależne.

Klasa, która ma za dużo powiązań, prawdopodobnie jest „przeciążona” obowiązkami, co implikuje złamanie kolejnej zasady, czyli *High Cohesion*.

Klasy A i B są ze sobą powiązane, gdy:

- ◆ obiekt A ma atrybut typu B lub typu C związanego z B,
- ◆ obiekt A wywołuje metody obiektu typu B,
- ◆ obiekt A ma metodę związaną z typem B (zmienna lokalna, typ wartości/parametru),
- ◆ obiekt A dziedziczy po B.

High Cohesion

Problem: jak sprawić, by obiekty miały jasny cel, były zrozumiałe i łatwe w utrzymaniu?

Rozwiązanie: należy przypisać odpowiedzialności do obiektu, tak aby spójność była jak największa.

Komentarz: każdy obiekt powinien się skupiać tylko na jednej odpowiedzialności. Nie należy tworzyć obiektów odpowiedzialnych za dużą część logiki aplikacji. W przypadku dużej klasy należy podzielić odpowiedzialność na kilka mniejszych klas.

Polymorphism

Problem: co zrobić, gdy odpowiedzialność różni się w zależności od typu?

Rozwiązanie: przy użyciu polimorfizmu należy przydzielić zobowiązania typom, dla których to zachowanie jest różne.

Komentarz: zbiór obiektów może mieć zbiór zachowań wspólnych oraz zbiór zachowań odmiennych. Dla zachowań odmiennych należy wykorzystać mechanizm polimorfizmu. Podobny efekt można uzyskać, korzystając z konstrukcji *if-else*, jednak nie jest to dobra praktyka, ponieważ w przypadku zmian trzeba odszukać i zaktualizować wszystkie miejsca w programie zawierające te konstrukcje. Prowadzi to do powstania trudnego i drogiego w utrzymaniu kodu.

Pure Fabrication

Problem: jak przydzielić odpowiedzialność, by nie naruszyć zasad *High Cohesion* i *Low Coupling*, gdy nie odpowiada nam rozwiązanie sugerowane przez *Information Expert*?

Rozwiązanie: należy przypisać zakres odpowiedzialności sztucznej lub pomocniczej klasie, która nie reprezentuje żadnego problemu domenowego. Nie narusza zasad *High Cohesion* i *Low Coupling*.

Komentarz: w pewnych przypadkach nie ma możliwości zachowania wszystkich wymienionych zasad, ponieważ pewne zasady mogą być ze sobą sprzeczne. Należy

wówczas stworzyć nową klasę, która nie reprezentuje żadnego problemu, udostępnia tylko i wyłącznie metody dostępne dla innych obiektów. Pozwoli to zachować zgodność z podstawowymi zasadami.

Indirection

Problem: komu przydzielić zobowiązanie, jeśli zależy nam na uniknięciu bezpośredniego powiązania między obiektami?

Rozwiązanie: należy przypisać te odpowiedzialności do nowego pośredniego obiektu. Obiekt ten będzie służył do komunikacji innych klas/komponentów/usług/pakietów, dzięki czemu nie będą one zależne bezpośrednio od siebie.

Komentarz: obiekty powinny być jak najmniej ze sobą powiązane. Aby zmniejszyć liczbę powiązań, można utworzyć klasę pośrednią, która będzie się komunikować z pozostałyimi obiektami. Klasa pośrednia może odpowiadać paru innym klasom, co powoduje, że jedna klasa nie musi mieć żadnych informacji na temat innych klas. Przykładem takiego zachowania może być odczyt danych z pliku lub z bazy danych. Klasa, która potrzebuje tych informacji, nie musi wiedzieć, skąd je wziąć oraz czy pochodzą z bazy danych, czy z pliku. Takimi rzeczami zajmuje się klasa pośrednia.

Protected Variations

Problem: jak projektować obiekty, by ich zmiana nie wywierała szkodliwego wpływu na inne elementy?

Rozwiązanie: należy zidentyfikować punkty przewidywanego zróżnicowania czy niestabilności i przypisać odpowiedzialność do wspólnego stabilnego interfejsu.

Komentarz: należy tak zaprojektować system, aby wymienianie elementów systemu na alternatywne było jak najprostsze. Podczas zmian powinna zostać dostarczona inna implementacja interfejsu, dlatego interfejsy powinny być tworzone dla punktów niestabilności, czyli miejsc w programie, które mogą wymagać różnych zachowań.

DRY

DRY (ang. *Don't Repeat Yourself* — nie powtarzaj się) to wzorzec, który mówi, że należy unikać powtarzania tych samych części kodu w różnych miejscach. Pozwala to na uniknięcie kopiowania lub kopiowania i delikatnego zmieniania części kodu w inne miejsca. Główną zaletą stosowania wzorca DRY jest możliwość uniknięcia błędów popełnianych w trakcie kopiowania powtarzających się fragmentów kodu. Pozwala to zaoszczędzić czas, który traci się, aby wprowadzić mało znaczące zmiany do kopiowanego kodu, a następnie czas tracony na szukanie zmian, które przeoczono w trakcie kopiowania. W zasadzie DRY chodzi również o unikanie powtórzeń czynności, które są wykonywane przez programistów. Takie czynności powinny być wykonywane przez generatory kodu lub skrypty automatyzujące pracę. Sposoby realizacji założeń DRY w językach programowania:

- ◆ funkcje,
- ◆ szablony lub makra,
- ◆ struktury,
- ◆ klasy,
- ◆ stałe,
- ◆ moduły,
- ◆ biblioteki.

KISS

KISS (ang. *Keep It Simple, Stupid*) to reguła mówiąca, że nie należy na siłę komplikować prostych rzeczy, aby przykładowo wykorzystać niepotrzebne w danym momencie wzorce projektowe. Nie należy tracić czasu na szukanie lepszego rozwiązania, gdy czas wykorzystany do opracowania rozwiązania będzie dłuższy niż ten potrzebny na wykonanie zadania w sposób standardowy. Do czasu pracy należy doliczyć czas poświęcony na szukanie lepszego rozwiązania. Należy go porównać z przewidywanym czasem przeznaczonym na rozwiązanie danego problemu w tradycyjny, najprostszy sposób.

Rule of Three

Zasada KISS jest ściśle powiązana z zasadą DRY. W niektórych sytuacjach łatwiej przekopiować kilka linijek, niż stosować regułę DRY. Dlatego została wymyślona zasada RoT (ang. *Rule of Three*). Mówiąc o niej, że jeśli istnieją trzy kopie tego samego kodu, należy zastosować regułę DRY i połączyć kod. Pozwala to uchronić program przed wprowadzeniem zmian w trzech miejscach, co może powodować dużą liczbę nowych problemów.

Separation of Concern

Separacja zagadnień (ang. *Separation of Concern*, SoC) polega na podziale programu na odrębne moduły, których funkcjonalność pokrywa się tak mało, jak to tylko możliwe. Taką budowę programu nazywa się modułową. Każdy element systemu powinien mieć swoje rozłączne i osobliwe zastosowanie. Celem SoC jest utworzenie systemu, w którym każda część pełni znaczącą rolę, przy zachowaniu możliwości maksymalnej adaptacji do zmian. SoC nie odnosi się tylko do architektury systemu, ale do różnych zagadnień, np. do podziału aplikacji na warstwy (prezentacji, logiki biznesowej, dostępu do danych, bazy danych). Zalety separacji:

- ◆ upraszcza pracę grupową, każdy pracownik pracuje nad swoim modułem;
- ◆ budowa modułowa pozwala na łatwiejszą rozbudowę systemu;
- ◆ poprawia czytelność budowy programu, pozwala łatwiej zrozumieć działanie systemu;
- ◆ poprawia reużywalność elementów systemu.

Separacja zagadnień i zasada pojedynczej odpowiedzialności wydają się bardzo podobne, jednak są pomiędzy nimi pewne różnice. SRP każe wydzielać każdą funkcjonalność do osobnej klasy, która ma się zajmować tylko jedną rzeczą, natomiast SoC każe dzielić zagadnienie (niekoniecznie program) na moduły, które w jak najmniejszym stopniu pokrywają się funkcjonalnością. W pewnych sytuacjach SoC kłoci się z SRP — w takiej sytuacji należy wybrać odpowiednie rozwiązanie w zależności od rodzaju problemu. Nie ma tu idealnego rozwiązania, dlatego należy wybrać takie, które jest najlepsze z dostępnych.

YAGNI

Zasada YAGNI (ang. *You Ain't Gonna Need It* — nie będziesz tego potrzebował) mówi, że nie należy pisać oprogramowania, które nie jest potrzebne, ale wydaje się, że może się kiedyś przydać. Nie należy patrzeć zbyt daleko w przyszłość, ponieważ po pewnym czasie może się okazać, że traci się czas na coś, co się nie przyda. W trakcie rozwoju projektu praktyczne zawsze zmieniają się założenia, co powoduje, że wcześniejsze koncepcje należy zweryfikować lub całkowicie odrzucić.

MoSCoW

MoSCoW (ang. *M — Must have it, S — Should have it, C — Could have if not affecting other things, W — Won't have this time*) to metoda, która zaleca podział wymagań klienta na kilka podgrup:

- ◆ te, które musi mieć (*M*),
- ◆ te, które powinien mieć (*S*),
- ◆ te, które może mieć, jeżeli nie zaszkodzą tworzeniu gotowych funkcjonalności (*C*),
- ◆ te, których nie uda się zaimplementować w danym terminie (*W*).

Przestrzegając zasady YAGNI, musisz pamiętać, że dobrą praktyką jest wykorzystanie metody MoSCoW dla uporządkowania priorytetów zadań.

Metodologie

Metodologia to inaczej sposób, schemat lub podejście do tworzenia oprogramowania.

Manifest Agile

Mianem programowania zwinnego określa się grupę metodyk wytwarzania oprogramowania, opartych na technice programowania iteracyjnego. Pojęcie programowania zwanego zostało wprowadzone w 2001 r. w manifeście Agile. Manifest Agile to

deklaracja wspólnych zasad dla zwinnych metodów tworzenia oprogramowania. Został opracowany w 2001 r. W spotkaniu uczestniczyli reprezentanci nowych metodów tworzenia oprogramowania.

W treści można przeczytać: „Wytwarzając oprogramowanie i pomagając innym w tym zakresie, odkrywamy lepsze sposoby wykonywania tej pracy. W wyniku tych doświadczeń przedkładamy:

- ◆ Ludzi i interakcje ponad procesy i narzędzia.
- ◆ Działające oprogramowanie ponad obszerną dokumentację.
- ◆ Współpracę z klientem ponad formalne ustalenia.
- ◆ Reagowanie na zmiany ponad podążanie za planem.

Doceniamy to, co wymieniono po prawej stronie, jednak bardziej cenimy to, co po lewej”.

Założenia manifestu Agile:

- ◆ osiągnięcie satysfakcji klienta poprzez szybkość wytwarzania oprogramowania,
- ◆ działające oprogramowanie jest dostarczane okresowo (raczej tygodniowo niż miesięcznie),
- ◆ podstawową miarą postępu jest działające oprogramowanie,
- ◆ późne zmiany w specyfikacji nie mają destrukcyjnego wpływu na proces wytwarzania oprogramowania,
- ◆ bliska, dzienna współpraca pomiędzy klientem a deweloperem,
- ◆ bezpośredni kontakt jako najlepsza forma komunikacji w zespole i poza nim,
- ◆ ciągła uwaga nastawiona na aspekty techniczne oraz dobry projekt (design),
- ◆ prostota,
- ◆ samozarządzalność zespołów,
- ◆ regularna adaptacja do zmieniających się wymagań.

Do metodów zwinnych należą:

- ◆ *Scrum*,
- ◆ *eXtreme Programming*,
- ◆ *Dynamic Systems Development Method*,
- ◆ *Adaptive Software Development*,
- ◆ *Crystal Clear*,
- ◆ *Feature Driven Development*,
- ◆ *Pragmatic Programming*.

Scrum

Scrum to metodyka (lub według niektórych podejścia do wytwarzania oprogramowania) iteracyjna (iteracje zwane *sprintami*) prowadzenia projektów. Każdy *sprint* trwa od tygodnia do miesiąca. Efektem każdego *sprintu* powinno być dostarczenie kolejnej działającej wersji produktu.

Przed przystąpieniem do prac niezbędne jest stworzenie ogólnej wizji produktu wraz z opisem funkcjonalności — ta część jest nazywana *Product Backlog*. Kolejnym etapem jest spotkanie zwane *Sprint Planning*, na którym ustalany jest cel *sprintu*. Na podstawie tego spotkania opracowywany jest *Sprint Backlog*, czyli lista zadań wraz z określeniem ich pracochłonności. Każdego dnia organizowane są spotkania *Daily Scrum*, na których omawiane są zadania zrealizowane poprzedniego dnia, zadania do realizacji w dniu spotkania oraz problemy występujące przy realizacji zadań. Każdy *sprint* kończy się spotkaniem *Sprint Review*, będącym podsumowaniem przebiegu tej części prac, na którym prezentowany jest produkt. Role w metodzie Scrum:

- ◆ *scrum master* — dba o proces oraz rozwiązuje problemy zewnętrzne,
- ◆ *product owner* — to osoba, która decyduje, co ma być zrobione w danym *sprintie*,
- ◆ *the team* — członkowie zespołu, w zespole powinny się znajdować osoby o zróżnicowanych kompetencjach,
- ◆ *others* — wszyscy inni, a więc przyszli użytkownicy, zarząd firmy itp.

eXtreme Programming

Programowanie ekstremalne — XP (ang. *eXtreme Programming*) — to metodologia tworzenia oprogramowania, która kładzie nacisk na zmieniające się wymagania klienta. Stosuje się ją zazwyczaj w małych i średnich projektach, w których nie wiadomo dokładnie, co i jak ma być zrobione. XP dzieli się na trzy zasadnicze elementy:

- ◆ część programowania,
- ◆ część pracy grupowej,
- ◆ część pracy z klientem.

Podstawowe praktyki wchodzące w skład metodyki XP:

- ◆ planowanie — wdrażanie kolejnych wydań (wersji) programu;
- ◆ małe wydania — częste łączenie małych ilości kodu;
- ◆ metafora systemu — każdy zespół programistyczny musi się kontaktować z klientem;
- ◆ prosty projekt — nie planuje się rozwiązań na wyrost, ponieważ nie wiadomo, czy zostaną wdrożone;
- ◆ ciągłe testowanie — programista przed napisaniem metody musi utworzyć kod, który ją będzie testował;

- ◆ *refactoring* (refaktoryzacja/przerabianie) — poprawianie działającej, przetestowanej już metody, aby uzyskać większą wydajność lub bardziej przejrzysty kod;
- ◆ programowanie w parach — poprawia jakość i szybkość tworzenia kodu;
- ◆ standard kodowania — narzucenie wszystkim programistom wspólnego standardu kodowania i dokumentowania;
- ◆ wspólna odpowiedzialność — wszyscy są odpowiedzialni tak samo, poprawki może nanieść dowolna osoba z zespołu;
- ◆ ciągłe łączenie — integracja programu tak często, jak to tylko możliwe;
- ◆ 40-godzinny tydzień pracy — należy ustalić granicę obciążenia grupy programistów;
- ◆ ciągły kontakt z klientem — klient powinien być ciągle dostępny w ramach konsultacji.

TDD

TDD (ang. *Test Driven Development*) to technika tworzenia oprogramowania sterowana przez testy. Należy do metodyk zwinnych. Dawniej była częścią metodyki XP, aktualnie jest samodzielną techniką. TDD polega na wielokrotnym powtarzaniu trzech kroków:

- ◆ utworzenie możliwie najprostszego testu jednostkowego, aby uniknąć błędu w samym teście; test służy do sprawdzenia funkcjonalności, która będzie implementowana w kolejnym kroku;
- ◆ implementacja funkcjonalności, która powinna spełniać założenia testu jednostkowego, a przeprowadzenie testu jednostkowego powinno się zakończyć sukcesem;
- ◆ refaktoryzacja kodu, czyli uporządkowanie kodu w taki sposób, aby spełniał standardy; refaktoryzacja nie może zmienić wyniku testu jednostkowego.

Największą wadą TDD jest to, że wymaga dodatkowej ilości czasu na utworzenie testów jednostkowych. Zanim przystąpi się do implementacji funkcjonalności, trzeba tworzyć testy, które będą sprawdzały poprawność implementacji. Tworzenie testów wydłuża dość znacznie wykonywanie tych samych zadań. Kolejną wadą jest konieczność utrzymywania testów, a więc podezas zmian w systemie należy przepisać bądź zaktualizować testy jednostkowe. Wprowadza to dość znaczny narzut pracy. Pisanie testów wymaga od programistów dodatkowych umiejętności, ponieważ nieprawidłowo napisane testy nie dość, że nie pomogą w odnalezieniu błędów, to mogą jeszcze spowodować wydłużenie czasu pisania aplikacji.

Podstawową zaletą TDD jest szybkość wychwytywania błędów. Testy jednostkowe pozwalają na bardzo szybkie odnalezienie błędów, co zmniejsza późniejsze koszty wynikające z szukania błędów w oprogramowaniu. Im później znajdzie się błąd, tym więcej on kosztuje. Błąd znaleziony za pomocą testów jednostkowych pozwala programistie na natychmiastowe poprawienie błędu. Gdy błąd zostaje odnaleziony w późniejszym

czasie, zazwyczaj musi się tym zajmować inna osoba, która nie zna tak dobrze tego problemu jak ta osoba, która pisała tę część kodu. Kolejną zaletą jest bardziej przeomyślany kod oraz możliwość testowania części funkcjonalności bez uruchamiania całego programu. Przy większych systemach ma to bardzo duże znaczenie.

Z pewnością nie należy stosować TDD do małych projektów. TDD przydaje się przy większych projektach, przy których pracuje większa liczba osób nieznających działania wszystkich elementów systemu.

Dodatek B

HTTP i SSL/TLS

Systemy internetowe to nadzbiór składający się z aplikacji internetowych, w skład których wchodzą witryny internetowe i strony internetowe. Ich budowa związana jest z podstawowymi technologiami webowymi opisanymi w tym dodatku. Aplikacja WWW to właściwie zbiór plików (*.html, *.htm, *.jsp, *.php, *.aspx, *.css czy pliki graficzne) i związanych z nimi komponentów (pliki binarne) składowanych w określonym katalogu (i jego opcjonalnych podkatalogach) serwera WWW. Z reguły wystarczy skopiować plik tekstowy na serwer HTTP, aby był dostępny dla wszystkich użytkowników internetu.

Aplikacje internetowe bazują głównie na rozszerzonej architekturze WWW składającej się z kilku warstw. Architektura trójwarstwowa składa się z warstwy prezentacji (interfejs użytkownika), warstwy aplikacji (logika biznesowa) oraz warstwy danych (baza danych). W każdej z tych warstw można umieścić odpowiednie elementy sprzętowo-programowe. W warstwie prezentacji jest to przeglądarka WWW zlokalizowana na dowolnym sprzęcie, w warstwie aplikacji znajduje się serwer WWW i/lub serwer aplikacji, a w ostatniej serwer bazy danych. Technologie powiązane z wymienionymi warstwami to dla pierwszej HTML, CSS, JavaScript czy Flash. W drugiej znajduje się przykładowo PHP, Java Servlets, JSP czy EJB. Bazy danych mogą być różne (relacyjne, np. MySQL, PostgreSQL, Oracle, lub nierelacyjne, np. MongoDB). Czasami rozdziela się drugą warstwę, zlokalizowaną na serwerze aplikacji, na dwie: warstwę logiki prezentacji oraz warstwę logiki biznesowej. Kolejnym możliwym podejściem jest architektura SOA polegająca na grupowaniu funkcjonalności wokół procesów biznesowych i udostępnianiu funkcjonalności w postaci usług.

HTTP

Składnikami architektury WWW są: serwer HTTP (serwer WWW), protokół HTTP oraz klient HTTP (przeglądarka WWW).

Zadaniem serwera WWW/Web jest obsługa żądań HTTP, rejestracja żądań, uwierzytelnianie i kontrola dostępu, kryptografia czy wybór wersji językowej wysyłanych plików. Do najbardziej popularnych serwerów WWW można zaliczyć: Apache, IIS, Apache Tomcat, IBM HTTP Server, Oracle WebLogic Server, nginx lub lighttpd.

Protokół HTTP jest zestawem reguł przesyłania plików hipertekstowych (komendy tekstowe). Celem jego działania jest transmisja plików HTML, a przez to zawartości stron WWW z serwera do przeglądarki. Protokół HTTP jest protokołem warstwy siódmej, a do transmisji wykorzystuje połączenia TCP. Ponieważ jest bezstanowy i bezsesyjny, każde żądanie jest traktowane zupełnie osobno. Działa w klasycznej architekturze klient-serwer, gdzie przeglądarka jest klientem. Służy do transmisji zasobów wskazywanych przez adresy URL (ang. *Uniform Resource Locator*)¹. Zasoby udostępniane poprzez HTTP są identyfikowane poprzez identyfikatory URI (ang. *Uniform Resource Identifiers*)² bądź URL. Zasadobem jest nie tylko plik, ale również np. dynamicznie generowana treść. Transportowana witryna WWW (strona internetowa) składa się z obiektów (plików). Możliwe jest przesyłanie dowolnego typu danych określonych poprzez nagłówek MIME (ang. *Multipurpose Internet Mail Extensions*). Typy MIME są istotne w kontekście przesyłania poczty elektronicznej. Protokół HTTP wymaga również, aby dane były transmitowane, podobnie jak wiadomości e-mail. MIME określa więc sposób opisu formatów plików (typ danych). Pole Content-Type nagłówka zawartosci określa ten typ (tabela B.1). Przykładowo wszystkie obsługiwane typy MIME dla serwera Apache można znaleźć w pliku konfiguracyjnym `mime.types`.

Tabela B.1. Typy danych MIME

Rozszerzenia plików	Typ danych
<code>.html</code>	<code>text/html</code>
<code>.css</code>	<code>text/css</code>
<code>.xml</code>	<code>text/xml</code>
<code>.js</code>	<code>application/x-javascript</code>
<code>.gzip</code>	<code>application/x-gzip</code>
<code>.zip</code>	<code>application/zip</code>
<code>.pdf</code>	<code>application/pdf</code>
<code>.png</code>	<code>image/png</code>
<code>.jpg</code>	<code>image/jpeg</code>
<code>.avi</code>	<code>video/avi</code>
<code>.mpeg</code>	<code>video/mpeg</code>
<code>.mid</code>	<code>audio/midi</code>
<code>.mpg</code>	<code>audio/mpeg</code>

Właściwy dialog między serwerem a klientem poprzedza zestawienie połączenia TCP („potrójny uścisnąć ręki”) na określonym porcie (domyślnie 80, ale również 8080 czy 443). Serwer po otrzymaniu komunikatu żądania generuje odpowiedź (status + specyficzna informacja). Struktura komunikatu rozpoczyna się od nagłówka zakodowanego w ASCII. Nagłówek zawiera:

¹ URL składa się z identyfikatora protokołu, adresu serwera oraz ścieżki dostępu do zasobu.

² URI zawiera możliwość przekazywania wartości zmiennych. URI = URL+URN (ang. *Uniform Resource Name*).

- ◆ rodzaj żądania,
- ◆ URI zasobu (względne lub bezwzględne),
- ◆ parametry żądania (pola).

Opcjonalnie po nagłówku mogą zostać przesłane dane związane z komunikatem HTTP po znaku CRLF. Każdy zasób żądany przez stronę HTML jest pobierany za pomocą osobnego żądania HTTP. Jak każdy protokół, również HTTP posiada nagłówek z określonymi polami definiującymi zachowanie.

Transmisja danych za pomocą HTTP to proste pobranie pliku z serwera, pobranie zawartości generowanej na żądanie lub przesłanie treści z przeglądarki do serwera. Typy żądań/metod HTTP:

- ◆ HEAD — żądanie nagłówka,
- ◆ GET — żądanie pobrania określonego zasobu,
- ◆ POST — żądanie akceptacji przetwarzanych danych (np. formularzy),
- ◆ PUT — transfer określonego zasobu,
- ◆ DELETE — żądanie usunięcia zasobu/dokumentu,
- ◆ TRACE — podgląd otrzymanego żądania (możliwość przejrzenia informacji dołączanych do żądań),
- ◆ OPTIONS — identyfikacja metod,
- ◆ TRACE — testowanie,
- ◆ LINK/UNLINK — relacje.

Przesłanie treści z przeglądarki do serwera bazuje na metodach GET i POST. Metoda GET przesyła dane w URI, natomiast metoda POST przesyła dane w treści żądania, po nagłówkach, jako wiadomość. Metoda:

- ◆ GET reprezentuje żądanie klienta HTTP (listing B.1). Następuje pobranie zasobu z serwera (dokument HTML, CSS, JavaScript, grafika). Parametrem żądania jest URI zasobu na serwerze. Żądanie może służyć również do przekazania danych formularza do serwera jako parametrów URI, ale operacja GET nie może zmieniać stanu serwera.
- ◆ POST reprezentuje żądanie klienta HTTP (listing B.2), ale do żądania dołączone jest ciało, które reprezentuje dane wysypane przez klienta HTTP do serwera HTTP (np. parametry, plik). Metoda ta — w przeciwieństwie do GET — pozwala na przetworzenie załączonych danych przez serwer WWW. Może służyć do przesyłania danych formularza, aktualizacji pliku lub bazy danych, wysyłania dużej ilości danych do serwera (nie ma ograniczeń rozmiaru, jak jest w przypadku GET), wysyłania danych przez użytkownika (które mogą zawierać nieznane znaki), ponieważ jest bardziej niezawodna i bezpieczniejsza, a zatem może zmienić stan serwera.

Dane wrażliwe powinny być wysyłane za pomocą żądania typu POST, a nie GET. Przy użyciu metody GET dane umieszczone są w adresie URL żądania, przez co są widoczne w przeglądarce.

Listing B.1. Żądanie HTTP — metoda GET

```
GET / HTTP/1.1
Accept: image/gif, image/x-bitmap, image/jpeg, image/pjpeg,
        ↳application/x-shockwave-flash, application/vnd.ms-
        ↳excel, application/vnd.ms-powerpoint,
        ↳application/msword, /*/
Accept-Language: pl
Accept-Encoding: gzip, deflate
User-Agent: Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1; SV1; .NET CLR
2.0.50727)
Connection: Keep-Alive
Host: www.onet.pl
```

Listing B.2. Żądanie HTTP — metoda POST

```
POST /registration/quicklogin.php HTTP/1.1
Accept: image/gif,/*fragment wycięty.../,/*
Referer: http://www.jakasStrona.com
Accept-Language: pl
Content-Type: application/x-www-form-urlencoded
Accept-Encoding: gzip, deflate
User-Agent: Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1;
        ↳SV1; .NET CLR 2.0.50727)
Host: www.jakasStrona.com
Content-Length: 38
Connection: Keep-Alive
Cache-Control: no-cache email=mail@test.com&password=alamakota
```

W odpowiedzi protokół HTTP generuje odpowiedni status:

- ◆ *1xx* — informacje (np. otrzymano żądanie, kontynuacja procesu),
- ◆ *2xx* — operacja wykonana pomyślnie (np. zrozumiana bądź zaakceptowana),
- ◆ *3xx* — przekierowanie (np. klient musi podjąć dodatkową akcję w celu zakończenia żądania),
- ◆ *4xx* — błąd klienta (np. zła składnia),
- ◆ *5xx* — błąd serwera (np. błąd wykonania zapytania).

Z drugiej strony celem działania przeglądarki internetowej jest odczytanie dokumentu HTML i wyświetlenie go w przyjaznej dla użytkownika formie. Klient też realizuje pewne zadania:

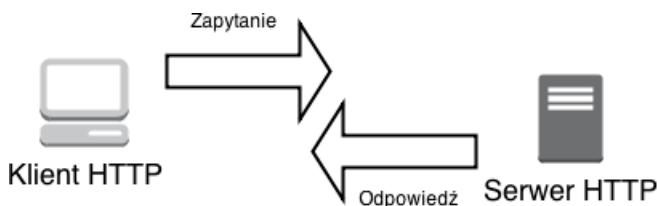
- ◆ inicjowanie połączenia HTTP,
- ◆ pobieranie interfejsu użytkownika,
- ◆ prezentacja interfejsu użytkownika,

- ◆ interakcja z użytkownikiem,
- ◆ buforowanie odpowiedzi,
- ◆ kryptografia.

Użycie HTTP do generowania zapytań (rysunek B.1):

- ◆ klient wysyła zapytanie o stronę WWW do serwera,
- ◆ serwer przygotowuje odpowiedź,
- ◆ serwer wysyła odpowiedź do klienta.

Rysunek B.1.
Standardowe
zapytania HTTP



SSL/TLS

Na wielu stronach internetowych znajduje się np. panel użytkownika lub administratora. W miejscu tym dokonywana jest autoryzacja w celu uzyskania dostępu do zasobów. Podawanie danych w dowolnym formularzu nie zawsze jest bezpieczne. Gwarantem bezpieczeństwa jest protokół SSL (ang. *Secure Socket Layer*) — certyfikat SSL. Celami szyfrowania są:

- ◆ ochrona poufności — tj. udostępnianie informacji wyłącznie tym osobom, które posiadają uprawnienia;
- ◆ ochrona autentyczności.

Protokół SSL odpowiada za szyfrowanie danych między warstwą transportową a warstwą aplikacji oraz autoryzację tożsamości serwera względem klienta. Protokół szyfruje wszystkie dane wychodzące z systemu internetowego i deszyfruje dane przychodzące. Stosowana jest technologia potwierdzania tożsamości. Dzięki zastosowaniu algorytmów kryptograficznych (np. RSA) przesyłane dane są szyfrowane. Im większa liczba bitów, tym trudniej rozszyfrować dane. Obecnie powszechnie korzysta się z szyfrowania 1024- lub 2048-bitowego.

RSA to asymetryczny, blokowy algorytm szyfrowania ze zmienną długością bloków. Posiada klucz szyfrujący o długości 1024, 2048 lub 4096 bitów. Polega na użyciu dwóch kluczy:

- ◆ publicznego — który jest dostępny dla każdego użytkownika i służy do zaszyfrowania transmisji oraz do weryfikacji podpisu;

- ◆ prywatnego — który jest przechowywany w bezpiecznym miejscu i służy do rozszyfrowywania danych zaszyfrowanych kluczem publicznym oraz do tworzenia podpisu elektronicznego.

Opis autoryzacji RSA:

- ◆ połączenie jest inicjowane po stronie programu klienta;
- ◆ po połączeniu się z serwerem klient otrzymuje od niego jego klucz publiczny;
- ◆ klucz ten jest porównywany z zachowanym w wewnętrznej bazie danych klienta z poprzednich połączeń;
- ◆ klient przekazuje serwerowi swój klucz publiczny, generuje losową liczbę, szyfruje ją za pomocą swojego klucza prywatnego oraz klucza publicznego serwera;
- ◆ serwer po otrzymaniu tak zakodowanej liczby rozszyfruje ją przy użyciu swojego klucza prywatnego i klucza publicznego klienta;
- ◆ liczba ta jest następnie używana jako klucz do kodowania podczas dalszej komunikacji.

Bezpieczne połączenie jest tworzone z wykorzystaniem kluczy publicznego i prywatnego (wiarygodność obu stron). Dodatkowo generowany jest klucz sesyjny używany jednorazowo dla konkretnego połączenia, który jest przesyłany za pomocą bezpiecznego już połączenia. Dalsza transmisja odbywa się z użyciem klucza symetrycznego.

TLS (ang. *Transport Layer Security*) to nowa nazwa dla SSL. Do wersji 3.0 stosowano nazwę SSL, a dla wersji 3.1 nazwa to TLS 1.0. HTTPS (ang. *HyperText Transfer Protocol Secured*) to HTTP z SSL/TLS. SSL/TLS ustawia bezpieczny tunel pomiędzy hostami. HTTP jest używany do transmisji zapytań i odpowiedzi. Przebieg procesu połączenia SSL/TLS:

- ◆ Przeglądarka internetowa, nawiązując połączenie z serwerem, wysyła do niego swój identyfikator. Wysyła ona ponadto swój unikalny klucz publiczny wygenerowany w momencie instalowania.
- ◆ Po otrzymaniu wiadomości serwer deszyfruje dane zaszyfrowane kluczem publicznym. Serwer wysyła do przeglądarki swój identyfikator oraz swój klucz publiczny, zaszyfrowane za pomocą klucza publicznego.
- ◆ Po otrzymaniu odpowiedzi przeglądarka deszyfruje dane kluczem publicznym oraz wysyła szyfrogram zaszyfrowany kluczem publicznym zawierający swój identyfikator i dodatkowe informacje o parametrach połączenia. Następnie wysyła żądanie przesłania klucza sesji używanego przez obie strony podczas wymiany informacji.
- ◆ Serwer wysyła klucz sesji zaszyfrowany za pomocą klucza publicznego.
- ◆ Po otrzymaniu od serwera klucza sesji klient wysyła kolejne żądania. Jeżeli serwer korzysta z protokołu SSL i HTTP, klient będzie szyfrował wysyłane żądania HTTP kluczem sesji wygenerowanym za pomocą protokołu SSL.

Do każdej transmisji danych stosowany jest inny klucz sesji, co uniemożliwia osobom niepowołanym użycie danych zdobytych podczas jednej transmisji w celu zaatakowania kolejnej transmisji.

Stosowany w aplikacjach internetowych certyfikat SSL jest odpowiednikiem dokumentu tożsamości dla serwera WWW. Zawiera on następujące składniki:

- ◆ nazwę właściciela certyfikatu,
- ◆ nazwę wydawcy certyfikatu,
- ◆ publiczny klucz właściciela dla algorytmu asymetrycznego,
- ◆ cyfrowy podpis wystawcy certyfikatu (np. VeriSign lub Comodo),
- ◆ okres ważności,
- ◆ numer seryjny (tzw. *fingerprint*).

Certyfikaty są wydawane przez niezależne i zaufane urzędy CA (ang. *Certification Authorities*). Wydanie certyfikatu jest poprzedzone sprawdzeniem autentyczności danego wnioskodawcy — czy taki ktoś istnieje i czy rzeczywiście jest tym, za kogo się podaje.

Rodzaje certyfikatów

Do wyboru są różne ich rodzaje służące do różnych celów. Koszt zakupu takiego certyfikatu SSL to od dwudziestu do nawet kilku tysięcy złotych za rok. Wszystko zależy jednak od rodzaju systemu internetowego.

Sposoby klasyfikacji certyfikatów SSL to: typ walidacji, liczba chronionych domen oraz firma wystawiająca certyfikat (centrum akredytacyjne).

Typy walidacji:

- ◆ DV (ang. *Domain Validation*) to certyfikaty tanie i najszybciej przyznawane. Weryfikacja odbywa się elektronicznie w oparciu o dane zawarte w rejestrze domen. Oferują podstawowe metody szyfrowania, które w przypadku większości zastosowań są wystarczające.
- ◆ OV (ang. *Organization Verification*) to certyfikaty o rozszerzonym sposobie weryfikacji. Zanim nastąpi wystawienie certyfikatu, musi dojść do weryfikacji poprzez okazanie szeregu dokumentów potwierdzających informacje na temat firmy i witryny internetowej. Certyfikaty z tej grupy zapewniają wyższy stopień bezpieczeństwa i dodatkowo potwierdzają wiarygodność korzystających z nich firm czy instytucji.
- ◆ EV (ang. *Extended Validation*) to certyfikaty z maksymalnym stopniem bezpieczeństwa. Przyznanie takiego certyfikatu wiąże się z przesaniem szczegółowych dokumentów w języku angielskim celem ich weryfikacji, a także nieradko z bezpośrednim kontaktem z centrum akredytacyjnym.

Liczba chronionych domen:

- ◆ pojedyncza domena,
- ◆ dzika karta, czyli ochrona wielu subdomen w obrębie tej samej domeny,
- ◆ wiele różnych domen, a domeny te mogą działać na różnych adresach IP.

Firmy wystawiające certyfikaty to np.: Comodo, GeoTrust, Thawte czy VeriSign.

Zakup certyfikatu SSL

W wielu przypadkach wystarczy najprostszy certyfikat SSL DV dla pojedynczej domeny³. Jaka firma będzie go dostarczać, nie ma w tym przypadku większego znaczenia.

Aktywacja, walidacja i instalacja certyfikatu SSL

Pierwszym krokiem jest aktywacja zakupionego certyfikatu. Należy wybrać rodzaj serwera sieciowego, np. cPanel. Konieczne jest podanie żądania certyfikatu CSR (ang. *Certificate Signing Request*). CSR zawiera wszystkie informacje niezbędne do wystawienia certyfikatu SSL przez CA. Należy pamiętać o wpisaniu poprawnych informacji, które są później niezbędne do prawidłowego wygenerowania żądania. Kolejnym krokiem jest wybranie powiązanego z domeną konta pocztowego w celu potwierdzenia własności domeny. Należy uzupełnić dane kontaktowe właściciela certyfikatu SSL.

Na podany wcześniej adres pocztowy powiązany z domeną jest wysyłana wiadomość dotycząca walidacji certyfikatu SSL. Należy kliknąć link i skopiować podany w mailu kod do odpowiedniego pola. Na podany wcześniej adres mailowy zostanie wysłany gotowy, zakodowany certyfikat SSL.

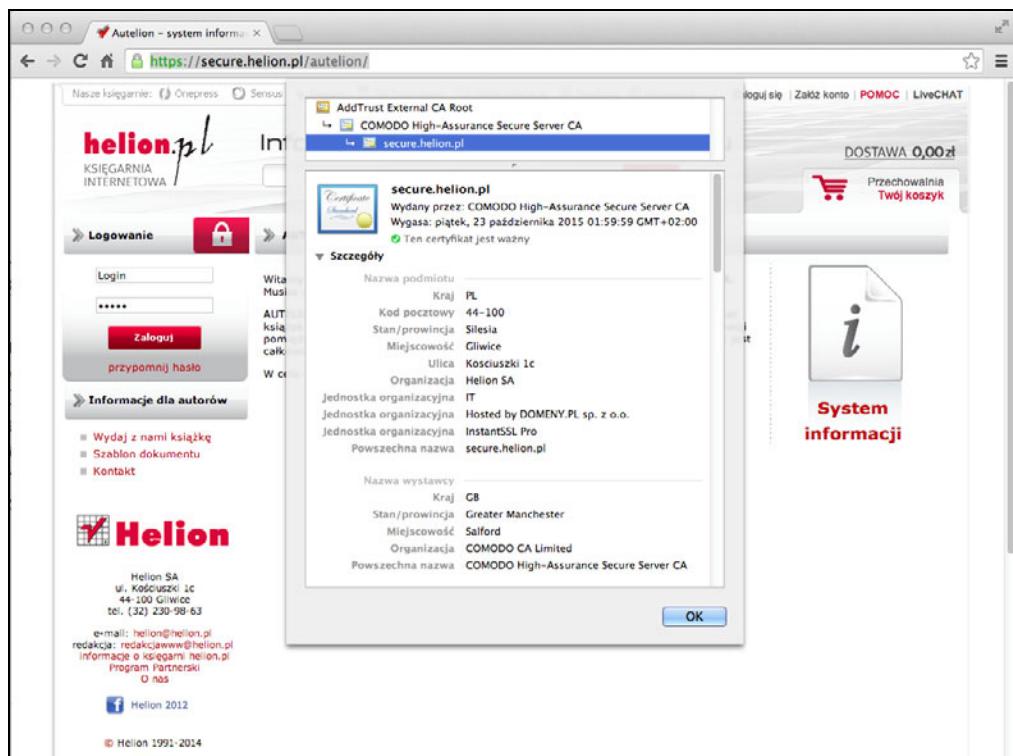
Uzyskany certyfikat SSL wraz z kluczem RSA w postaci zakodowanej należy wysłać dostawcy usług hostingowych, łącznie z informacją na temat adresu URL, do którego dostęp ma zostać zaszyfrowany. Dostawca dokończy proces instalacji certyfikatu SSL na konkretnym serwerze. Instalacja certyfikatu na stronie internetowej wymaga dedykowanego adresu IP.

Certyfikat w praktyce

Warto jeszcze przeprowadzić weryfikację certyfikatu SSL. Wystarczy wejść na jedną z popularnych witryn, które w kilka sekund dostarczą wszystkich niezbędnych informacji o stanie certyfikatu SSL.

³ Zaleca się opuszczenie strony internetowej, która wymusza połączenie szyfrowane, ale nie posiada żadnego zabezpieczenia lub działa ono niepoprawnie. Można również zaakceptować zagrożenie, dodać wyjątek i podjąć próbę odczytania zawartości strony. W tym przypadku żadne szyfrowanie nie jest realizowane.

Poza zmianą protokołu w adresie URL ważnym elementem jest szary symbol kłódki po lewej stronie od adresu w przeglądarce Mozilla Firefox bądź zielonego znaku kłódki i zmienionego koloru protokołu w samym adresie w przypadku Google Chrome. Oznacza to, że połączenie między stroną a serwerem jest już szyfrowane. Za pomocą pojedynczego kliknięcia myszką symbolu kłódki można poznać szczegóły dotyczące szyfrowania⁴. W zakładce *Connection* można zobaczyć, z jakiego certyfikatu korzysta witryna oraz jakiego rodzaju szyfrowanie zostało zastosowane, wraz ze szczegółami dotyczącymi użytej wielkości klucza czy rodzaju samego algorytmu. Można również zweryfikować zakres ważności certyfikatu, a także sprawdzić, przez kogo i dla kogo został on wygenerowany (rysunek B.2).



Rysunek B.2. Certyfikat wydawnictwa Helion

⁴ Gdy zamiast symbolu kłódki po lewej stronie od paska adresu ukazuje się pełna nazwa firmy lub witryny internetowej napisana zielonym kolorem, to oznacza to efekt zastosowania certyfikatu z grupy EV. Gdy pojawia się symbol wykrzyknika lub przekreślonej kłódki, sygnalizuje to problem związany z tym, że pomimo szyfrowanego połączenia ścieżka dostępu do konkretnego elementu strony wskazuje na protokół HTTP.

Dodatek C

HTML 5 i CSS 3

Pisanie stron internetowych należy rozpocząć od znaczników HTML i formatowania wyników z użyciem stylów CSS.

HTML 5

W trójwarstwowym modelu dokumentu (struktura, prezentacja, zachowanie) HTML znajduje się w pierwszej warstwie.

Podstawowym szkieletem każdej strony WWW jest język znaczników HTML. Istnieje kilka typów znaczników:

- ◆ strukturalne — opisujące strukturę dokumentu, np. `<body>`,
- ◆ prezentacyjne — opisujące wygląd poszczególnych elementów, np. `<i>`,
- ◆ hipertekstowe — tworzące odnośniki, np. `<a> `.

Język skryptowy HTML 5 jest najnowszą wersją języka HTML. Nowa wersja zawiera zmiany i udogodnienia dla projektantów witryn internetowych. HTML 5 ma stanowić jeden standard, który pozwala na jednoznaczna interpretację kodu HTML we wszystkich przeglądarkach, eliminuje dodatkowe wtyczki zewnętrzne i jest niezależny od urządzenia. Poprawnie napisana strona ma działać bezproblemowo niezależnie od rodzaju urządzenia. Podczas gdy specyfikacje HTML 4.01 oraz XHTML posiadają trzy wersje: *Strict* (brak tu znaczników oraz atrybutów prezentacyjnych, rolę formatowania przyjmuje arkusz stylów CSS), *Transitional* (pełna lista znaczników i atrybutów) oraz *Frameset* (pozwala zastosować ramkę `<frameset>` zamiast elementu `<body>`), w HTML 5 jest dostępny tylko jeden typ dokumentu. HTML 5 usuwa niezalecane znaczniki oraz wprowadza:

- ◆ znaczniki semantyczne,
- ◆ nowe elementy formularzy oraz podstawowe elementy walidacji,
- ◆ nowe elementy interaktywne i multimedialne,

- ◆ rysowanie za pomocą JavaScriptu po wirtualnym płótnie,
- ◆ możliwość osadzenia MathML i SVG bezpośrednio w dokumencie,
- ◆ *Web Storage*,
- ◆ szereg nowych API dla JavaScriptu, m.in. *Drag and Drop* czy geolokalizacja.

Pierwszą zauważalną różnicą jest zatem zmieniona składnia dokumentu (listing C.1).

Listing C.1. Podstawowy szablon dokumentu HTML 5

```
1 <!DOCTYPE html>
2 <html>
3     <head>
4         <meta charset="UTF-8">
5         <title></title>
6     </head>
7 <body>
8
9 </body>
10 </html>
```

Definicja typu dokumentu DTD (deklaracja `<!DOCTYPE>`) nie jest znacznikiem, a instrukcją dla przeglądarki na temat wersji HTML, w której została napisana strona, i musi wystąpić przed znacznikiem `<html>`.

Podstawą języka HTML są znaczniki:

- ◆ nagłówki,
- ◆ linie rozdzielające,
- ◆ komentarze,
- ◆ paragraf,
- ◆ definiujące formatowanie,
- ◆ cytowania,
- ◆ znaki specjalne,
- ◆ linki,
- ◆ zakotwiczenia,
- ◆ obrazy,
- ◆ listy (numerowana, nienumerowana, definicji),
- ◆ tabele (składają się z wierszy, a wiersze z kolumn),
- ◆ formularze.

Każdy znacznik składa się z elementu i atrybutów. Parametry dotyczące pliku HTML są definiowane w nagłówku strony (`<head>`), np.:

- ◆ <title>Tytuł strony</title>;
- ◆ <meta> — określenie np. autora, kodowania znaków, języka strony: <meta name="Author" content="Gall Anonim" /> czy <meta http-equiv="Content-Type" content="text/html; charset=iso-8859-2" />;
- ◆ <base> — lokalizacja dodatkowych zasobów (adres bazowy dla odnośników na stronie): <base href="http://serwer.pl/" />;
- ◆ <style> — definicja stylów CSS;
- ◆ <script> — skrypty na stronie;
- ◆ <link> — definicja powiązań pomiędzy dokumentami (np. CSS); typowe argumenty to: href — adres zasobu, type — typ zawartości, rel — typ wskazywanego dokumentu:
<link rel="stylesheet" type="text/css" href="style.css" />.

Właściwa zawartość strony znajduje się w sekcji <body>. Przykładowo formularze pozwalają zdobyć wiele informacji o użytkownikach odwiedzających stronę WWW, a wykorzystywane są np. do:

- ◆ rejestracji użytkowników,
- ◆ zamawiania produktów w sklepach,
- ◆ wypełniania ankiet.

Tworzenie formularza przy użyciu znacznika <form> zostało zaprezentowane na listingu C.2.

Listing C.2. Schemat formularza

```
<form action="gdzie_wyslac" method="post">
    <!-- pola formularza -->
</form>
```

Atrybutami znacznika <form> są:

- ◆ id — identyfikator formularza,
- ◆ name — definiuje nazwę formularza,
- ◆ action — określa, gdzie mają zostać wysłane dane z formularza (URI),
- ◆ method — metoda wysyłania formularza (GET lub POST¹ — większe możliwości i bezpieczeństwo),
- ◆ target — definiuje okno docelowe dla otwieranego dokumentu (po przesłaniu formularza),

¹ Przekazywanie zmiennych z formularza bazuje na metodach GET i POST (bezpieczne przesyłanie) oraz mechanizmie sesji i ciasteczkach.

- ◆ enctype — definiuje typ wartości wysyłanej przez formularz, stosowany dla metody POST atrybutu method:
 - ◆ "application/x-www-form-urlencoded" — wartość domyślna (dla GET zawsze),
 - ◆ "multipart/form-data" — powinien być stosowany przy wysyłaniu plików dla pola input typu file,
 - ◆ "text/plain" — powoduje zamianę spacji na znaki + i pozostawienie znaków specjalnych bez kodowania.

Każdy formularz składa się z różnego typu pól, dzięki którym użytkownik może wprowadzać dane:

- ◆ pole input (do umieszczania w formularzu kilku różnych rodzajów kontrolek):
 - ◆ tekstowe,
 - ◆ typu password,
 - ◆ wyboru (wielokrotnego),
 - ◆ opcji (wyboru jednokrotnego),
 - ◆ wyboru plików,
 - ◆ listy rozwijane,
 - ◆ obszar tekstowy,
 - ◆ przyciski.

HTML 5 dostarcza szereg nowych atrybutów (tabela C.1) i elementów, które przedstawiono w tabeli C.2.

Tabela C.1. Tabela z nowymi atrybutami globalnymi HTML 5

Atrybut	Opis
contenteditable	Określa, czy element posiada obszar do edycji
Contextmenu	Menu kontekstowe dla wybranego znacznika
Draggable	Określa, czy dany element można przeciągnąć
dropzone	Określa, czy przeciągnięta rzecz ma być skopiowana, przesunięta, czy tworzony jest do niej odnośnik po jej upuszczeniu
spellcheck	Określa, czy przeglądarka ma sprawdzać poprawność wpisywanego tekstu przez użytkownika

Tabela C.2. Tabela z nowymi elementami HTML 5

Element	Opis
<article>	Reprezentuje niezależną część dokumentu, np. artykuł, wpis w blogu czy post na forum
<aside>	Część dokumentu marginalnie powiązana ze stroną

Tabela C.2. Tabela z nowymi elementami HTML 5 — ciąg dalszy

Element	Opis
<audio>	Kontener do dodawania treści dźwiękowych
<canvas>	Kontener, w którym możliwe jest rysowanie grafiki
<command>	Komenda, która może być wywołana przez użytkownika
<menu>	Wewnątrz osadzane są poszczególne polecenia i ukrywające się za nimi funkcje (element <command>)
<datalist>	Generator listy używany wraz z elementem <input> z atrybutem list
<details>	Stosowany do prezentacji szczegółów danego elementu na stronie
<summary>	Zawarty pomiędzy znacznikiem <details>
<embed>	Używany do prezentacji interaktywnych animacji
<figcaption>	Podpis elementu figure
<figure>	Stosowany do umieszczenia grupy treści multimedialnych wraz z podpisem
<footer>	Stopka na stronie lub w danym jej fragmencie
<header>	Zawiera nagłówek fragmentu strony lub całej strony
<hgroup>	Grupuje elementy <h1> ... <h6>
<keygen>	Określa pole generatora par kluczów używane w formularzach
<mark>	Wyróżnienie ciągu znaków w dokumencie
<meter>	Kontener, który określa wartość z danego przedziału
<nav>	Sekcja odpowiadająca za nawigację
<output>	Element ukazujący wartość liczbową zmian danego procesu
<progress>	Informuje o postępie zadania
<ruby>	Umożliwia wprowadzenie adnotacji nad fragmentem tekstu
<rp>	Pozwala na wprowadzenie wymowy słowa umieszczonego w znaczniku <ruby>
<rt>	Pozwala na wprowadzenie adnotacji w przeglądarce, która nie obsługuje znacznika <ruby>
<section>	Element grupujący treść, który reprezentuje wybraną, zwartą część dokumentu — sekcję
<source>	Kontener zasobów multimedialnych
<time>	Zwraca datę i czas
<video>	Zawiera odnośnik do pliku wideo
<wbr>	W przypadku długich słów lub ciągów znaków daje możliwość ich podziału

Mimo nowych elementów języka, te z poprzednich wersji są nadal aktualne i można je stosować poza kilkoma wyjątkami.

Sekcje

W wersji piątej języka HTML zmieniono podejście do projektowania stron internetowych. Obecnie cały projekt strony jest oparty na sekcjach. Samo utworzenie strony już nie wystarczy. Trzeba też zadbać o jej pozycjonowanie, aby strona została zaindeksowana

przez wyszukiwarki internetowe. Wspomniane znaczniki semantyczne odpowiadają za konkretną sekcję strony oraz określają rodzaj i rolę treści na stronie. Elementami sekcji są:

- ◆ body,
- ◆ section,
- ◆ nav,
- ◆ article,
- ◆ aside,
- ◆ hgroup,
- ◆ header,
- ◆ footer,
- ◆ address.

W zależności od miejsca w pozycji drzewa dokumentu dany znacznik może mieć różne znaczenie semantyczne.

W technologii HTML 5 wprowadzono szereg zmian, aby usprawnić interakcję użytkownika z systemem (formularze). Dodano nowe typy elementu `input`, jak również szereg atrybutów, dzięki którym validacja formularzy stała się łatwiejsza i większą jej część można teraz przeprowadzić za pomocą samego języka, bez konieczności wykorzystania JavaScriptu. Nowe typy pól formularzy zostały omówione w tabeli C.3.

Tabela C.3. Nowe typy pól formularza HTML 5

Typ	Opis
range	Suwak, za pomocą którego możliwe jest zwiększenie bądź zmniejszenie wartości liczbowej; na osi znajdują się wartości całkowite
date/datetime	Pole pozwalające na wybór daty bądź daty i godziny
number	Pole, gdzie za pomocą odpowiednich strzałek góra – dół zmienia się wartość liczbowa; mogą to być liczby rzeczywiste
url/email/tel	Podobne do pola tekstopowego, w jego wnętrzu należy wpisać łańcuchy znakowe odpowiadające kolejno adresowi URL, adresowi e-mail i numerowi telefonu; w przeciwnym wypadku formularz nie przejdzie validacji
color	Pole służące do wybrania określonego koloru z panelu użytkownika
search	Służy do wpisania tekstu, który ma zostać wyszukany

Nowe typy pól formularza

Nowe typy pól formularza pozwalają wybierać różne formaty daty i czasu, suwak umożliwiający określenie wartości oraz widget z wyborem koloru. Wyróżniono pola tekstopowe, które mają pełnić specjalne funkcje (np. dla e-maila, adresu URL, telefonu, pola wyszukiwania oraz pozwalającego na wprowadzanie liczb). Wszystkie pola są tworzone za pomocą znacznika `<input>`, któremu nadaje się odpowiednią wartość dla atrybutu `type`.

Atrybuty dla formularza

Z nowymi elementami języka związane są również nowe rodzaje atrybutów zaprezentowane w tabeli C.4. Atrybuty te pozwalają na określenie specyficznych ustawień dla wybranego pola formularza, co ułatwia walidację tych pól.

Tabela C.4. Nowe atrybuty dla pól formularza HTML 5

Atrybut	Opis	Wartość
autocomplete	Autouzupełnianie formularza	on, off
form	Identyfikator formularza	form_id, form1_id, form2_id
height	Wysokość w pikselach	
max	Maksymalna wartość	liczba, data
min	Minimalna wartość	liczba, data
required	Wymagane pole	required
width	Szerokość w pikselach	
autofocus	Uaktywnianie pola	autofocus
list	Lista danych	datalist_id
pattern	Wyrażenie regularne	wyrażenie regularne
step	Liczba kroków	liczba, any
placeholder	Informacja o danych	tekst
formenctype	Ustala sposób kodowania danych dołączonych do zapytania (metoda POST)	
multiple	Element, który może przyjmować kilka wartości (oddzielonych przecinkiem); dla pól typu email i file oraz dla znacznika <select>	multiple

Pola, które są wymagane do uzupełnienia, otrzymują atrybut required. Atrybut autofocus pozwala ustawić kursor myszy na określonym polu. W przypadku zezwolenia na autouzupełnianie dodaje się atrybut autocomplete. Atrybut placeholder pozwala wstawić tekst początkowy, który znika po kliknięciu danego pola, przykładowo dający użytkownikowi wskazówkę na temat tego, co oznacza dane pole lub jakie dane należy wpisać. Atrybut multiple zastosowany np. dla pola typu file pozwala na dodanie więcej niż jednego pliku. W formularzu dzięki zastosowaniu atrybutów min i max można określić konkretny przedział dat.

Znaczniki

Znacznik <keygen> ma zapewnić autoryzację użytkownika. Posłuży on jako generator kluczy dla formularzy. Przeglądarka wygeneruje dwa klucze: prywatny zostanie po stronie klienta, a drugi z formularzem trafi do serwera.

Poza zmianami związanymi z budową struktury dokumentu HTML 5 wprowadzono nowe znaczniki, które w bardzo łatwy sposób umożliwiają zamieszczanie na stronie

internetowej elementów multimedialnych, dźwięków, muzyki (znacznik `<audio>`) oraz plików wideo (znacznik `<video>`) bez konieczności stosowania dodatkowych wtyczek. Ze znacznikami związane są nowe atrybuty:

- ◆ `autoplay` — umożliwia automatyczne włączenie filmu bądź muzyki zaraz po załadowaniu strony internetowej,
- ◆ `controls` — odpowiada za dodanie panelu kontrolnego,
- ◆ `preload` — ładuje dźwięk bądź film przed jego odtworzeniem,
- ◆ `loop` — powoduje odtwarzanie pliku dźwiękowego lub filmu w pętli,
- ◆ `poster` — używany, gdy ładowany plik ma duży rozmiar — w tym czasie wyświetla się użytkownikowi obraz informacyjny, aby nie oglądał pustej sekcji strony,
- ◆ `type` — zastosowany wraz z elementem `source` umożliwia załączenie kilku formatów plików, z których wybrany będzie plik obsługiwany przez przeglądarkę.

Kolejnym bardzo ważnym elementem języka HTML 5 jest znacznik `<canvas>`. Pozwala on na dynamiczne — za pomocą skryptów — renderowanie kształtów i obrazów bit-mapowych. `<canvas>` to wirtualne płótno, po którym można rysować (piksele, linie, prostokąty, elipsy, gradienty, cienie i tekst), przeskakowywać i obracać obrazy, odczytywać jako `jpg/.png`, jak również kopiować plik z grafiką na płótno. `<canvas>` stanowi dwuwymiarową siatkę, w której lewy górny wierzchołek płotna posiada współrzędne (0,0). Element `<canvas>` jest implementowany w stylu DOM (ang. *Document Object Model*). Oznacza to, że istnieje możliwość użycia języka JavaScript, aby odwołać się do obiektów, ich metod i właściwości. Rysowanie na płótnie odbywa się za pośrednictwem kontekstu. Kontekst to obiekt powiązany z konkretnym płotnem, zdolny do realizacji operacji rysowania. Definiowanie płotna jako przezroczystego elementu pokazano na listingu C.3.

Listing C.3. Tworzenie elementu canvas

```
<canvas id="canvas" width="600px" height="600px">  
</canvas>
```

Znacznik `<svg>` pozwala na osadzanie grafiki wektorowej. SVG (ang. *Scalable Vector Graphics*) jest plikiem tekstowym, którego zawartość tworzy zestaw znaczników XML opisujących pozycję, rozmiary, wygląd dwuwymiarowych kształtów, obrazów oraz tekstu tworzonego dynamicznie. Podstawowe elementy, które można rysować, to: prostokąt, koło, elipsa i linia. Grafikę SVG można dodać bezpośrednio na stronie lub z zewnętrzneego pliku. Bardziej złożone kształty rysuje się za pomocą kombinacji linii łamanych i krzywych, czyli ścieżek. Aby narysować ścieżkę dla elementu `<path>`, należy określić atrybut `d`, który zawiera kolejne kroki rysowania. Dodatkowo istnieje możliwość korzystania z filtrów pozwalających na:

- ◆ manipulowanie kolorem: nasyceniem, jasnością, kontrastem itd.,
- ◆ łączenie obrazów,
- ◆ rozmywanie i wyostrzenia,

- ◆ generowanie tekstur,
- ◆ symulację oświetlenia.

Web Storage

Kolejnym ważnym mechanizmem jest *Web Storage*. Przy użyciu powszechnie znanych ciasteczek zapamiętanie choćby imienia użytkownika jest dość skomplikowane. Ciasteczko musi mieć ustaloną datę ważności i może zostać w łatwy sposób usunięte oraz posiada ograniczone miejsce na zapis (4 kB). Witryna może korzystać jednocześnie z kilkudziesięciu takich ciasteczek. *Web Storage* to API pozwalające na przechowywanie danych z serwisów internetowych w przeglądarce internetowej. W przeciwieństwie do ciasteczek przechowuje on dane tylko w pamięci podręcznej przeglądarki i nie wysyła tych informacji przy każdym żądaniu. W *Web Storage* nie ma konieczności ustawiania dat ważności informacji czy ograniczeń miejsca. *Web Storage* pozwala na przechowywanie nawet do 10 MB danych. Wyróżnia się dwa rodzaje *Web Storage*: `localStorage` i `sessionStorage`. Są to obiekty będące właściwościami obiektu `window`. Różnią się one od siebie tym, że dane `sessionStorage` zostają usunięte wraz z zamknięciem przeglądarki lub karty oraz po określonym czasie bezczynności użytkownika, chyba że przeglądarka ma ustaloną opcję zapamiętywania sesji. Dane z `localStorage` są trwale powiązane, dlatego możliwe jest odwołanie się do nich z dowolnego miejsca, okna czy zakładki. Zapisywanie i odczytywanie takich danych jest bardzo proste, co zostało pokazane na listingu C.4.

Listing C.4. Zapisywanie i odczytywanie danych z `sessionStorage`

```
1 // Zapisywanie
2 sessionStorage.liczba = 100;
3 // Odczytywanie
4 alert(sessionStorage.liczba);
```

Do obiektu `localStorage` można się odwołać jak do tablicy asocjacyjnej. Czyszczenie przechowywanych danych realizuje funkcja `localStorage.clear()`.

Server Side Events

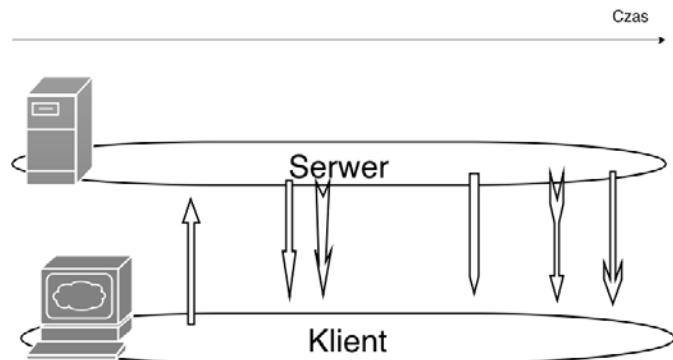
Technika HTML 5 SSE (ang. *Server Sent Events*), nazywana też EventSource (rysunek C.1), polega na tym, że:

- ◆ klient wysyła zapytanie o stronę WWW do serwera, tak jak przy użyciu HTTP;
- ◆ żądana strona wykonuje kod JavaScript, który otwiera połączenie z serwerem;
- ◆ serwer wysyła informacje do klienta, gdy pojawiają się nowe dane.

Cechy połączenia:

- ◆ ruch generowany w czasie rzeczywistym z serwera do klienta;
- ◆ w pętli są wywoływanie zdarzenia, które automatycznie wysyłają dane;
- ◆ nie ma możliwości połączenia z serwerem z innej domeny.

Rysunek C.1.
HTML 5 SSE

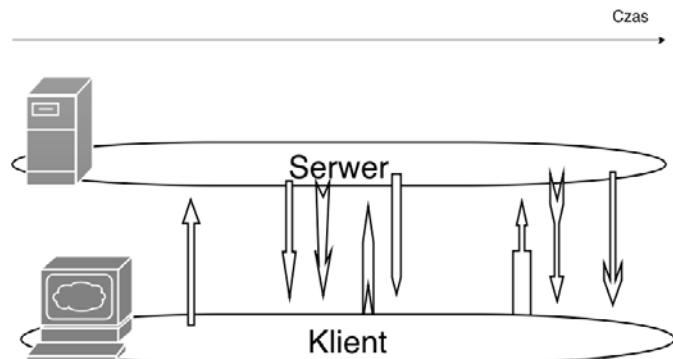


WebSockets

Technika HTML 5 WebSockets (rysunek C.2) polega na tym, że:

- ◆ klient wysyła zapytanie o stronę WWW do serwera, tak jak przy użyciu HTTP;
- ◆ żądana strona wykonuje kod JavaScript, który otwiera połączenie z serwerem;
- ◆ klient i serwer mogą wysyłać wiadomości, gdy pojawiają się nowe dane po obu stronach.

Rysunek C.2.
HTML 5 WebSockets



Cechy połączenia:

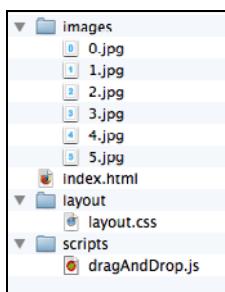
- ◆ ruch generowany w czasie rzeczywistym z serwera do klienta i odwrotnie;
- ◆ w pętli są wywoływanie zdarzenia, które automatycznie wysyłają dane;
- ◆ istnieje możliwość połączenia z serwerem z innej domeny;
- ◆ istnieje możliwość użycia strony trzeciej („gospodarza”) jako serwera WebSockets, co powoduje jedynie konieczność wdrożenia strony klienta.

Drag and Drop

Poza wszystkimi zaprezentowanymi do tej pory innowacjami, jakie dostarcza kolejna już wersja języka HTML, istnieje jeszcze kilka ciekawych mechanizmów, jakie mogą być implementowane na stronach pisanych właśnie w języku tej technologii. Jednym z nich jest bardzo użyteczny i w znacznym stopniu ułatwiający komunikację z systemem mechanizm *Drag and Drop* (przeciagnij i upuść). API *Drag and Drop* pozwala na wykonywanie takich operacji jak: chwytyanie, opuszczanie i przenoszenie wybranego elementu. Każdemu elementowi, który ma być chwytyany, należy nadać atrybut *draggable* o wartości true. Dodatkowo domyślnie obrazki (znacznik ** z określonym atrybutem *src*) oraz linki (z określonym atrybutem *href*) są elementami, które można chwytać. Aby zaprezentować w praktyce, jak korzystać z API, utworzono przykład (rysunek C.3), który pozwoli na uszeregowanie liczb od najmniejszej do największej oraz sprawdzi poprawność wykonanej operacji. Aplikacja umożliwia sortowanie cyfr od najmniejszej do największej (rysunek C.4). Każda z cyfr ma ustaloną wartość atrybutu *draggable* (listing C.5). Dzięki temu możliwe jest przenoszenie wybranego elementu.

Rysunek C.3.

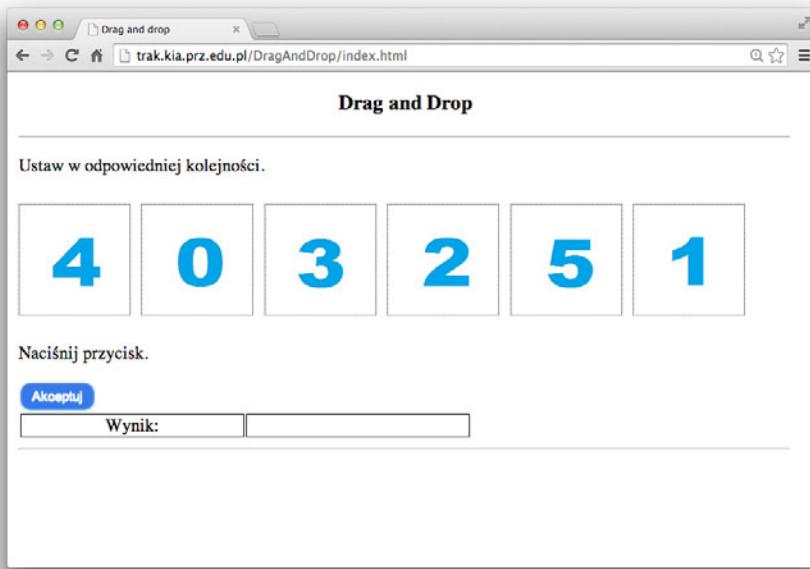
Struktura katalogów
i plików



Listing C.5. Ustawienie atrybutu (index.html)

```
<div class="image">
  
</div>
```

Na początku dwie zmienne reprezentujące obiekty będą zamieniane pozycjami (listing C.6). Gdy najedzie się na dowolny obrazek, zmianie ulega kształt kurSORA myszki. Świadczy to o tym, że dany element można przenosić. Podczas przenoszenia zostanie wywołana funkcja *drag(event)*. Pobierany w niej obiekt jest przenoszony i zapisywany w jednej ze zmiennych. Podczas upuszczenia obiektu wywoływana jest funkcja *drop(event)*. W niej pobierany jest obiekt, z którym ma być zamieniony obiekt pierwszy. Następnie dokonywana jest podmiana. Operacja ta jest wykonywana za pomocą HTML DOM i zmiany węzłów rodzica. *preventDefault()* zapobiega otwieraniu nowego okna przeglądarki podczas upuszczenia obiektu. Funkcja *allowDrop(event)* pokazuje, gdzie dany element może zostać upuszczony. Po naciśnięciu przycisku sprawdzany jest efekt sortowania (rysunek C.5). Za tę akcję odpowiada funkcja *isCorrect()*.



Rysunek C.4. Sortowanie cyfr z użyciem Drag and Drop

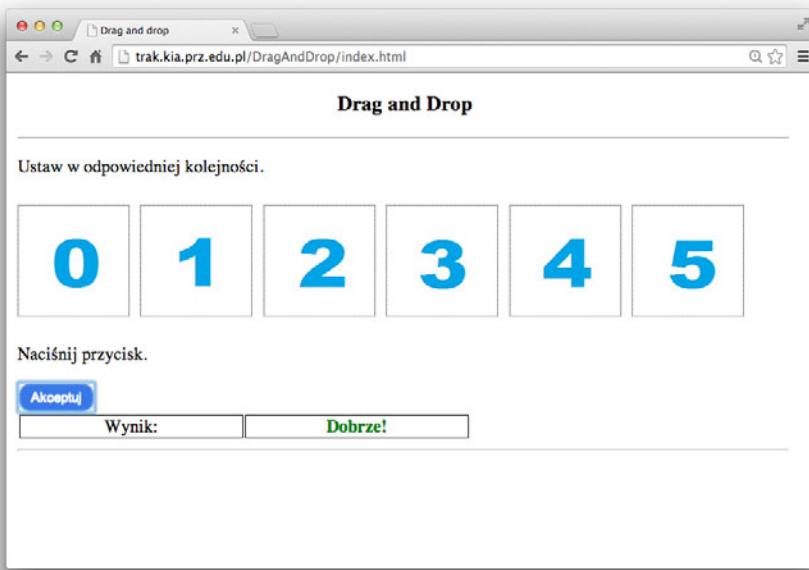
Listing C.6. Zdarzenia obsługiwane przez metody umieszczone w pliku (dragAndDrop.js)

```
var transferImageFrom = null;
var transferImageTo = null;

function drag(event){
    transferImageFrom = event.target;
}

function drop(event){
    if(event.preventDefault){
        event.preventDefault();
    }
    transferImageTo = event.target;
    parentNodeFrom = transferImageFrom.parentNode;
    transferImageTo.parentNode.appendChild(transferImageFrom);
    parentNodeFrom.appendChild(transferImageTo);
    transferImageFrom = null;
    transferImageTo = null;
}

function allowDrop(event){
    if(event.preventDefault){
        event.preventDefault();
    }
}
```



Rysunek C.5. Wynik sortowania

Geolokalizacja

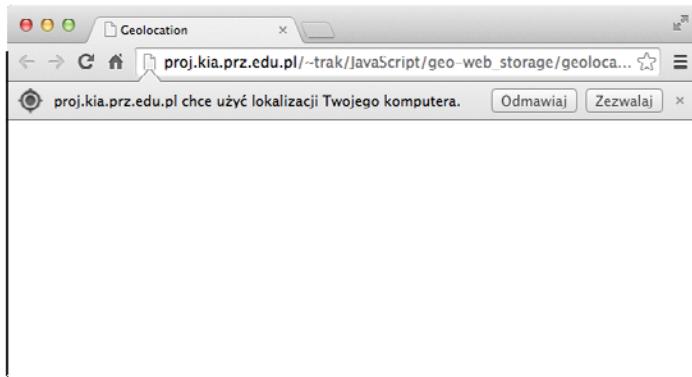
Funkcjonalność geolokalizacji polega na zlokalizowaniu aktualnego położenia. Istnieje możliwość przekazania tych informacji innym osobom. Niezależnie od wersji przeglądarki obsługującej geolokalizację użytkownik jest zawsze pytany o zgodę na użycie usługi (rysunek C.6). W zależności od możliwości lokalizacji można dokonać poprzez:

- ◆ adres IP — miejscowości (rysunek C.7);
- ◆ połączenie z wieżą telefonii komórkowej — miejscowości, osiedle, ulica;
- ◆ połączenie z siecią bezprzewodową — miejscowości, osiedle, ulica, budynek;
- ◆ dedykowane urządzenie GPS — dokładne miejsce pobytu.

Przykładowo za pomocą właściwości `coords.latitude` i `coords.longitude` pobiera się współrzędne geograficzne (listing C.7). Realizację zadania prezentuje rysunek C.8.

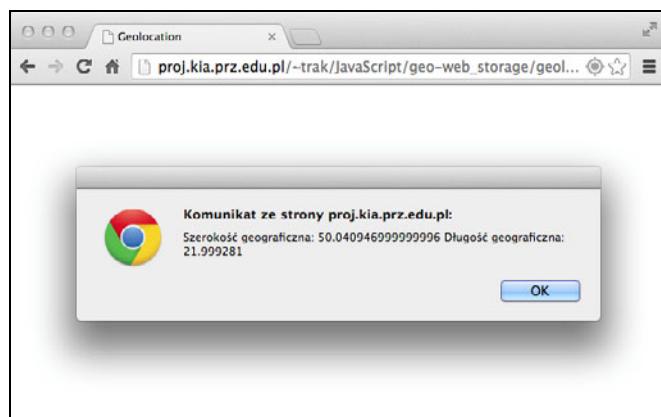
Listing C.7. Funkcja pobierająca i wyświetlająca informacje na temat położenia geograficznego

```
function geo(position)
{
    var lat = position.coords.latitude;
    var lng = position.coords.longitude;
    alert('Szerokość geograficzna: '+lat+' Długość geograficzna:
          ↵ '+lng);
}
```



Rysunek C.6. Pytanie o zgodę użytkownika na poznanie pozycji

Rysunek C.7.
Określenie pozycji



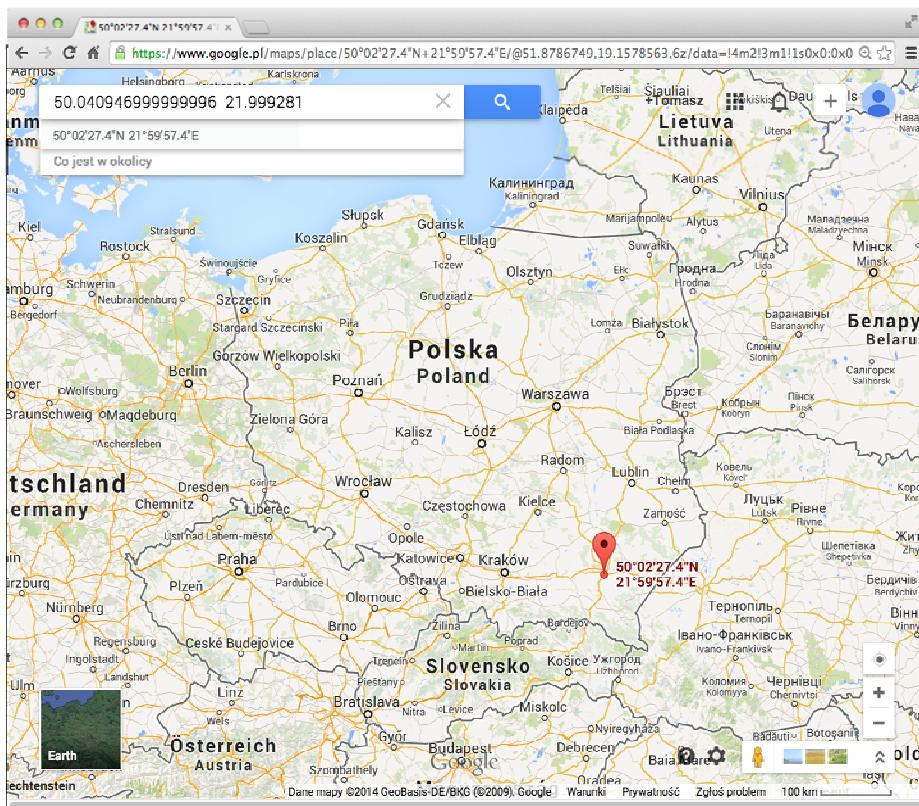
Walidacja

Po utworzeniu strony w HTML można zweryfikować jej poprawność przy użyciu dostępnych validatorów kodu (rysunek C.9).

CSS 3

W warstwie prezentacji modelu dokumentu należy umieścić CSS (ang. *Cascading Style Sheets*). Jak wynika z powyższego, HTML powinien być stosowany tylko do tworzenia szkieletu strony i jej zawartości.

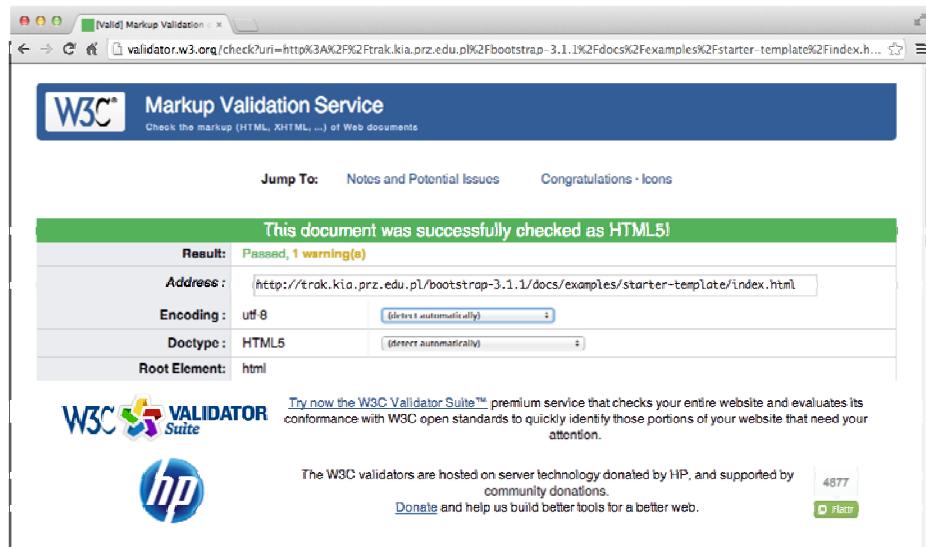
Podstawą każdej strony internetowej jest estetyczny i zwracający na siebie uwagę wygląd. Może on przyciągnąć lub zniechęcić. Za wygląd strony (kolorystykę i tła,



Rysunek C.8. Weryfikacja położenia i jego dokładności

rozmiar i krój czcionki, wyrównanie tekstu czy marginesy) odpowiadają kaskadowe arkusze stylów CSS. Rozszerzają one możliwości twórców witryn i wprowadzają rozwiązania, do których wcześniej konieczne było stosowanie języka JavaScript. CSS określa układ graficzny dokumentów HTML. Kaskadowe arkusze stylów to język opisu prezentacji (wyświetlania), czyli formatowania treści strony WWW. Dzięki CSS stało się możliwe odseparowanie struktury dokumentu opisanego za pomocą języka HTML oraz zdefiniowanie części odpowiedzialnej za określenie wyglądu poszczególnych elementów. Arkusz stylów CSS składa się więc z listy reguł określających, w jaki sposób strona (wybrane elementy) ma być wyświetlana przez przeglądarkę internetową.

Listing C.8 zawiera podstawową regułę CSS. Selektor określa, do jakich znaczników HTML stosuje się właściwość. Można tutaj określić nazwę znacznika, identyfikator lub jego klasę. Właściwość oznacza na przykład kolor tła (background-color), wielkość czcionki (font-size), jej grubość (font-weight) czy wyrównanie tekstu (text-align). Wartość jest zależna od właściwości, do której się ją przypisuje. W przypadku koloru tła jest to po prostu kolor, np. white, #ff32a4.



Rysunek C.9. Validator W3C

Listing C.8. Ogólna reguła CSS

```
selektor{
    właściwość: wartość;
    właściwość: wartość;
    ...
}
```

Rodzaje arkuszy stylów:

- ◆ importowany (ang. *Imported*) — zewnętrzny arkusz stylów dołączony do innego arkusza;
- ◆ zewnętrzny (ang. *External*) — plik tekstowy z rozszerzeniem *.css* zawierający definicje stylów (odwołania w dowolnym pliku HTML);
- ◆ osadzony (ang. *Embedded*) — w nagłówku pliku HTML (działa tylko w danym dokumencie);
- ◆ wpisany (ang. *In-line*) — dołączony do jednego znacznika, określa sposób jego wyświetlania.

Działanie arkuszy stylów cechuje hierarchia źródeł: wpisany, osadzony, zewnętrzny i importowany.

CSS 3 nie posiada jednej specyfikacji, a jest ona podzielona na moduły, które dzielą ją na kilkanaście części ściśle powiązanych z danym zagadnieniem. Podział specyfikacji na moduły jest dużym ułatwieniem dla producentów przeglądarek internetowych,

którzy będą mogli szybciej wdrażać obsługę kaskadowych arkuszy stylów, opierając się na już zatwierdzonych modułach. Niektóre moduły mają status CR (ang. *Candidate Recommendation*), który określa, że dany moduł jest stabilny.

Nie sposób wymienić wszystkich selektorów, ale należy wspomnieć o selektorze uniwersalnym w postaci wzorca *, opisującym wszystkie elementy, oraz o selektorze typu A, opisującym elementy tego typu, czy selektorze atrybutu A[attr], opisującym element A z atrybutem attr. Pozostałe typy to pseudoklasy (strukturalne, łączy, działań użytkownika), pseudoelementy czy kombinatory.

Atrybuty HTML służą do prostej obsługi zdarzeń i za pomocą JavaScriptu pozwalają modyfikować obsługę zdarzeń w locie. Zdarzenie określa stan strony WWW, dla którego może zostać uruchomiony skrypt definiujący określona funkcję zmiany jej stanu. Funkcje JavaScript związanego ze zdarzeniami są uruchamiane po wystąpieniu danego zdarzenia. Zdarzenia są związane z poszczególnymi elementami strony. Możliwe zdarzenia to zdarzenia: myszy, klawiatury, formularza, kontrolek formularza oraz dokumentu. Rejestracja zdarzenia (addEventListener) polega na określeniu typu zdarzenia i wywoływanej funkcji oraz włączeniu lub wyłączeniu mechanizmu bąbelkowania².

W tym miejscu koniecznie należy wspomnieć o obiektowym modelu dokumentu. DOM definiuje właściwości i metody wszystkich obiektów HTML na stronie oraz określa, w jaki sposób dostać się do elementu HTML na stronie od strony programistycznej. Metody DOM są dostępne nie tylko w CSS, ale również np. w JavaScriptie. CSS i JavaScript używają DOM do operowania na dokumentach. Elementy dokumentu posiadają hierarchię drzewa:

- ◆ dziecko (ang. *Child*) — element będący o jeden szczebel niżej w drzewie w stosunku do danego elementu;
- ◆ potomek (ang. *Descendant*) — element będący o jeden lub więcej szczebli niżej w drzewie w stosunku do danego elementu;
- ◆ rodzic (ang. *Parent*) — element o jeden szczebel wyżej w drzewie w stosunku do danego elementu;
- ◆ przodek (ang. *Ancestor*) — element będący o jeden lub więcej szczebli wyżej w drzewie w stosunku do danego elementu;
- ◆ brat (ang. *Sibling*) — element mający tego samego rodzica co inny element; jeśli znajduje się obok niego, to jest to brat przylegający (ang. *Adjacent Sibling*).

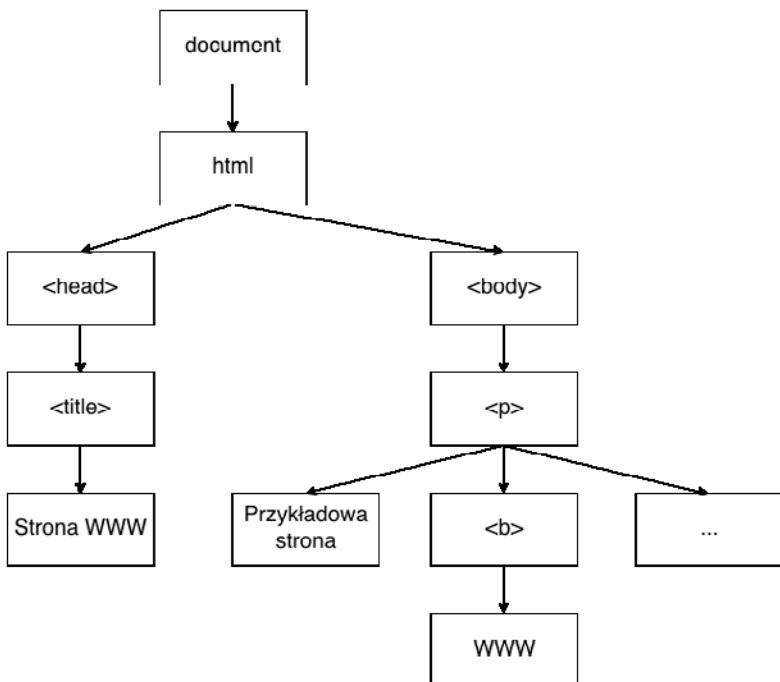
Przykładową strukturę dokumentu przedstawia listing C.9. Postać drzewa dokumentu zaprezentowano na rysunku C.10.

² Zdarzenia zachowują się tak samo jak bąbelki — lecą ku górze. W fazie bąbelkowania zdarzenie zostaje odebrane przez wszystkich przodków celu zdarzenia. „Bąbelkowe” wywoływanie zdarzeń polega więc na przekazywaniu zdarzenia do kolejnych elementów w górę hierarchii dokumentu — aż do elementu document.

Listing C.9. Przykładowa struktura dokumentu

```
<html>
  <head>
    <title>
      Strona WWW
    </title>
  </head>
  <body>
    <div>
      <p>
        Przykładowa strona <b>WWW</b>...
      </p>
    </div>
  </body>
</html>
```

Rysunek C.10.
Struktura dokumentu
w postaci drzewa



Wyszukiwanie elementów odbywa się po nazwie znacznika elementu (np. `document.getElementsByTagName('p')`), identyfikatorze (np. `document.getElementById("id_elementu")`) lub według klasy. Nie trzeba korzystać z metody `getElementsByName` do pobrania wszystkich elementów na stronie oraz odwoływania się do nich z tablicy przez numer indeksu. Każdy węzeł (czyli element, fragment tekstu, komentarz itp.) posiada pola wskazujące na jego sąsiednie węzły. Przechodzenie po drzewie dokumentu może być również realizowane na bazie atrybutów. Atrybuty odczytuje się za pomocą `nazwa`, a ustawia za pomocą `nazwa = wartość`. Można również tworzyć, dodawać i przenosić elementy.

Dodatkowo każdy element generuje w dokumencie prostokątny obszar zwany pudełkiem (ang. *Box Model*). Pudełko składa się z:

- ◆ zawartości (ang. *Content*), np. tekstu, obrazka,
- ◆ otaczających marginesów wewnętrznych (ang. *Padding*),
- ◆ obramowania (ang. *Border*),
- ◆ marginesów (ang. *Margin*).

Nowe selektory

Selektory pozwalają odnieść się do wybranych elementów drzewa dokumentu i powiązać z nimi pewne własności. Dodane do kodu strony selektory, pseudoklasy i pseudoelementy umożliwiają w efektywniejszy sposób dotarcie do elementów strony. Lista nowych selektorów została zawarta w tabeli C.5.

Tabela C.5. Lista nowych selektorów w CSS 3

Wzorzec	Opis
A[attr^="string"]	Element A, którego atrybut attr ma wartość zaczynającą się od łańcucha string
A[attr\$="string"]	Element A, którego atrybut attr ma wartość kończącą się łańcuchem string
A[attr*="string"]	Element A, którego atrybut attr ma wartość zawierającą podłańcuch string
A:root	Element A, element główny dokumentu (w dokumencie HTML jest to element html)
A:nth-child(n)	Element A, n-te dziecko jego rodzica
A:nth-last-child(n)	Element A, n-te dziecko jego rodzica, licząc od końca
A:nth-of-type(n)	Element A, n-ty element siostrzany tego typu
A:nth-last-of-type(n)	Element A, n-ty element siostrzany tego typu, licząc od końca
A:last-child	Element A, ostatnie dziecko jego rodzica
A:first-of-type	Element A, pierwszy element siostrzany tego typu
A:last-of-type	Element A, ostatni element siostrzany tego typu
A:only-child	Element A, jedyne dziecko jego rodzica
A:only-of-type	Element A, jedyny element siostrzany tego typu
A:empty	Element A, który nie ma dzieci (wliczając węzły tekstowe)
A:target	Element A będący celem identyfikatora URI
A:enabled, A:disabled	Należący do interfejsu użytkownika element A, który jest włączony lub wyłączony
A:checked	Na przykład pole typu <i>radio button</i> (przelącznik opcji) lub pole wyboru
A:not(p)	Element A, który nie pasuje do selektora prostego p
A ~ B	Element B, przed którym znajduje się element A

Nowe właściwości

Właściwości obiektów można definiować w klasie. Każdemu elementowi można zatem przyporządkować odpowiednią klasę, jednak klasy pozwalają definiować style tylko wybranych elementów HTML (a nie całości dokumentu). Element lub elementy HTML, które mają mieć własność wybranej klasy, muszą mieć nazwę tej klasy zdefiniowaną jako wartość atrybutu `class`. Style CSS normalnie są dodawane do elementów na podstawie ich pozycji w drzewie dokumentu. Pseudoklasy klasyfikują elementy inaczej niż po ich nazwie, atrybutach czy zawartości, tzn. w zasadzie nie są ustalane na podstawie drzewa dokumentu, a mogą być dynamiczne. Element może „nabywać” lub „tracić” pseudoklasę podczas interakcji z użytkownikiem.

Standardowe właściwości w CSS dotyczą głównie: marginesów, wypełnienia, obramowania, specyfikacji czcionki, tekstu, koloru i tła czy pozycjonowania elementu.

Nowe właściwości odpowiadają:

- ◆ zaokrągleniu narożników — właściwość `border-radius`,
- ◆ przecięciu tła — właściwość `background-clip`,
- ◆ definiowaniu liczby kolumn (szpalt) — właściwość `column-count`,
- ◆ definiowaniu stopnia nieprzezroczystości — właściwość `opacity`,
- ◆ płynnemu efektowi przejścia,
- ◆ transformacjom — skalowanie, obracanie elementów itd.,
- ◆ nowemu układowi strony („elastyczne pudełka”).

Nowe zdarzenia:

- ◆ obiektu `window`,
- ◆ formularza,
- ◆ myszy,
- ◆ mediów.

Kaskadowe arkusze stylów oprócz zmian w selektorach wprowadziły nowe właściwości i wartości w formatowaniu czcionki i tekstu. Możliwe jest zatem łamanie długich wyrazów, jeśli nie mieszczą się one w danym kontenerze strony, cieniowanie tekstu, dodawanie obrysu wyrazom czy określenie liczby kolumn tekstu, który ma się znajdować w danym kontenerze. Zmiany są również widoczne, jeśli chodzi o kolory, tło obramowania i gradient poszczególnych elementów. Wprowadzono kilka sposobów definiowania przezroczystości poprzez RGBA, HSLA i za pomocą właściwości `opacity`. Kolory mogą być definiowane przy użyciu nowej przestrzeni barw CMYK. Zaprojektowano nową stylistykę tła (grafika może dynamicznie dopasowywać się do rozmiaru strony). Oprócz tego dla kontenerów wprowadzono możliwość zaokrąglania narożników, cieniowania i projektowania własnego obramowania. Jednak bez wątpienia najważniejszą i najbardziej widoczną zmianą w CSS 3 jest wprowadzenie transformacji (translacji, skalowania, odbijania i pochylenia), przejść i animacji. Do wykonania prostego

ruchu elementów na stronie, nadania ruchowi płynności, opóźnienia, czasu, różnego rodzaju funkcji nie potrzeba już JavaScriptu, gdyż to wszystko oferują kaskadowe arkusze stylów.

Twitter Bootstrap

Jest to framework kaskadowych arkuszy stylów dostępny na wolnej licencji (Apache License 2.01). Pozwala na tworzenie graficznego interfejsu, a także aplikacji webowych. Wykorzystywany jest zestaw narzędzi (gotowych elementów) HTML, CSS oraz JavaScript. Zawiera m.in. takie elementy jak menu rozwijane, przyciski, pola wprowadzania, nawigacja, dzielenie na strony, powiadomienia, wskaźniki postępu itd. Bootstrap wykorzystuje w większości określone klasy CSS, dzięki czemu pozwala na podpięcie bez żadnych zbędnych i skomplikowanych zabiegów ze strony projektanta interfejsu.

Framework można pobrać ze strony <http://getbootstrap.com/>. Pliki można dołączyć do aplikacji webowych, wykorzystując bootstrap.* (skompilowane) oraz bootstrap.min.* (skompilowane, bez zbędnych dodatków). Dołączenie następuje standardowo przez dopisanie informacji na temat CSS i JavaScriptu (listing C.10).

Listing C.10. Dodanie plików bootstrap

```
<link href=". /sciezka/do/bootstrap/css/bootstrap.min.css"
      ↪rel="stylesheet">
<script src=". /sciezka/do/bootstrap/bootstrap.min.js"></script>
```

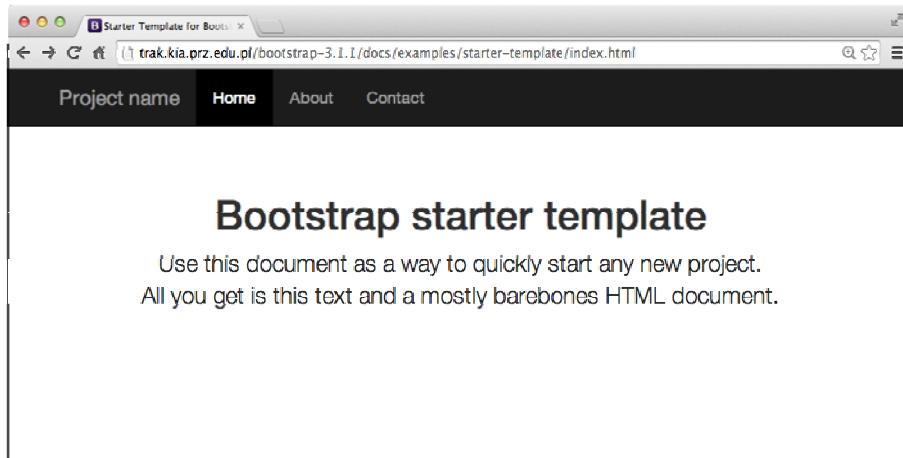
Podstawowe elementy zaprezentowano na stronie <http://getbootstrap.com/getting-started/#examples>, jednak pełna ich lista znajduje się pod adresem <http://getbootstrap.com/components/> i zawiera wszystkie niezbędne elementy przy tworzeniu standaryzowanych stron i aplikacji webowych. Dokument, którego można użyć do szybkiego utworzenia szablonu dla projektu, znajduje się pod adresem [examples/starter-template/](http://getbootstrap.com/examples/starter-template/) (rysunek C.11). Widoki strony responsywnej zaprezentowano na rysunkach C.12 (zwinięty) i C.13 (rozwinięty).

CSS 4

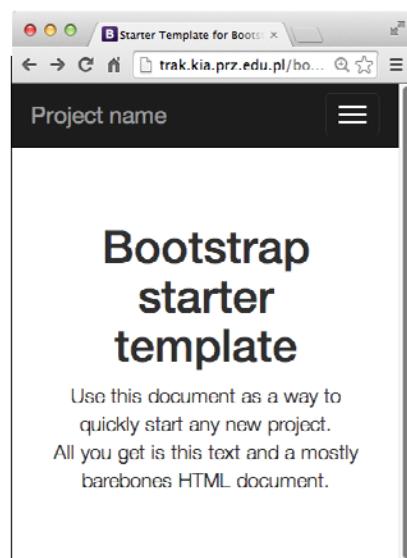
Kolejny poziom arkuszy stylów zawiera m.in. nowe pseudoklasy.

Pseudoklasy logiczne:

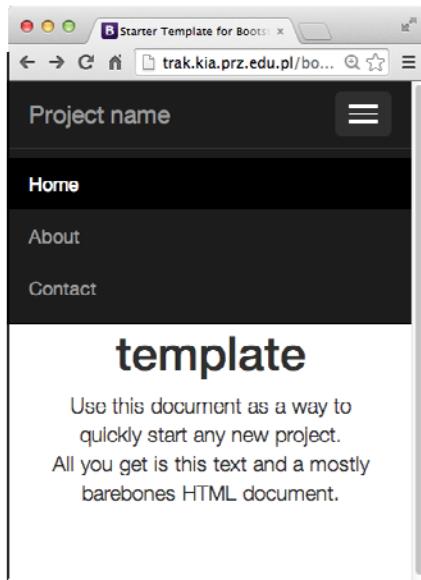
- ◆ :matches — pozwala dopasować kilka elementów do wzorca, np.:
li a:matches(:link, :hover, :visited, :focus),
- ◆ :not — oprócz prostych selektorów (tak jak w CSS 3) może przyjmować dodatkowo także ich listy.



Rysunek C.11. Okno o szerokości wystarczającej na pełne wyświetlenie nawigacji



Rysunek C.12. Widok responsywny zwinięty



Rysunek C.13. Widok responsywny rozwinięty

Pseudoklasy adresu:

- ◆ :any-link — pozwala określić każdy stan linku,
- ◆ :local-link — odpowiada za lokalny, aktualny adres na stronie, a parametrem może być liczba, która pozwoli odwołać się do określonego poziomu w strukturze, np. :local-link(0) — strona główna.

Pseudoklasy czasu:

- ◆ :past, :current, :future — pseudoklasy dotyczące elementów, które na przykład są odczytywane ze strony i prezentowane niewidomemu w formie ścieżki audio (tzw. *text-to-speech*).

Pseudoklasy formularza:

- ◆ :indeterminate — dotyczy elementów, które mogą być w stanie nieokreślonym — głównie elementów formularza.

Pseudoklasy struktury drzewa:

- ◆ :nth-match, :nth-last-match — pseudoklasy do przeszukiwania elementów DOM.

Pseudoklasy struktury siatki:

- ◆ :column, :nth-column, :nth-last-column — pseudoklasy pozwalające na manipulację strukturami tabelarycznymi.

Dodatek D

HTML DOM i JavaScript

Ważnym elementem jest poznanie struktury dokumentu (DOM) oraz języka pozwalającego na tworzenie aplikacji po stronie klienta.

HTML DOM

Obiektowy model dokumentu DOM (ang. *Document Object Model*) to niezależny od platformy i języka interfejs pozwalający na dostęp do struktury dokumentu (strony WWW). Dokument składa się z węzłów, które można usuwać, dodawać lub modyfikować za pomocą klas i metod dostępnych w DOM. Dostęp do elementów dokumentu jest możliwy poprzez wywołanie odpowiedniej metody. Każdy węzeł może posiadać kilka kolejnych węzłów. Dzięki drzewiastej budowie możliwe są operacje na węzłach — elementach strony WWW¹.

Struktura DOM została przedstawiona na rysunku D.1.

Metody dostępne w DOM

Dostęp do elementów:

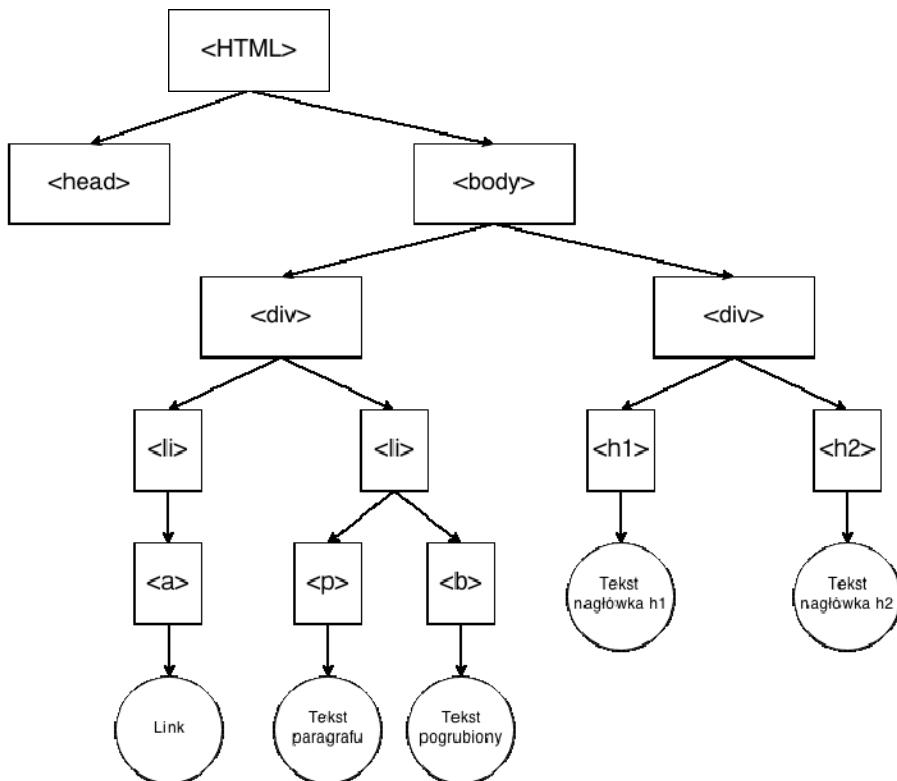
- ◆ `getElementById()` — wyszukiwanie elementów po `id` (zwraca element o danym ID, przy czym ID musi być unikalne — nie może się powtarzać w jednym dokumencie):

```
document.getElementById('bold');
```

- ◆ `getElementsByClassName()` — wyszukiwanie elementów po typie znacznika (zwraca tablice elementów):

```
document.getElementsByClassName('p');
```

¹ <http://www.w3.org/TR/REC-DOM-Level-1/level-one-core.html#ID-1590626201>,
<http://www.w3.org/TR/DOM-Level-2-Core/core.html#ID-1590626201>



Rysunek D.1. Schemat DOM

- ◆ `getElementsByClassName()` — wyszukiwanie elementów po klasie elementu (zwraca elementy danej klasy w formie tablicy, elementy jednej klasy mogą się powtarzać w jednym dokumencie):

```
document.getElementsByClassName('pogrubione');
```

Dodawanie i usuwanie węzłów:

- ◆ `appendChild()` — dodaje nowy podwózecel do danego węzła:

```
body.appendChild(element);
```

- ◆ `removeChild()` — usuwa węzeł:

```
body.removeChild(element);
```

- ◆ `replaceChild()` — podmienia węzeł:

```
element.replaceChild(nowy_el, stary_el);
```

- ◆ `insertBefore()` — wstawia nowy węzeł przed wybranym podwózeclem:

```
element.insertBefore(nowy_el, dany_el);
```

Atrybuty i elementy:

- ◆ `createAttribute()` — tworzy węzeł atrybutu:

```
document.createAttribute("class");
```

- ◆ `createElement()` — tworzy węzeł elementu:

```
document.createElement("div");
```

- ◆ `createTextNode()` — tworzy węzeł tekstowy:

```
document.createTextNode("napis");
```

- ◆ `getAttribute()` — zwraca wartość danego atrybutu:

```
element.getAttribute(nazwaAtrybutu);
```

- ◆ `setAttribute()` — ustawia lub zmienia wartość atrybutu:

```
document.getElementById("zdjeciel").setAttribute("src",  
    "zdjcie.jpg");
```

Właściwości dostępne w DOM

Właściwość `nodeName` określa nazwę węzła, która:

- ◆ jest tylko do odczytu,
- ◆ `nodeName` elementu jest taka sama jak jego `tagName`,
- ◆ `nodeName` atrybutu jest taka sama jak nazwa atrybutu,
- ◆ `nodeName` tekstu jest zawsze równa `#text`,
- ◆ `nodeName` dokumentu jest zawsze równa `#document`.

Właściwość `nodeValue` określa wartość danego węzła, gdzie:

- ◆ dla elementu jest niezdefiniowana (`undefined`),
- ◆ dla tekstu jest danym tekstem,
- ◆ dla atrybutu jest wartością atrybutu.

Właściwość `nodeType` jest tylko do odczytu i zwraca typ węzła. Dostępne typy węzłów: Document, DocumentFragment, DocumentType, Element, Attr, Text, CDATASection, Comment, Entity, Notation, EntityReference, ProcessingInstruction.

Poziomy DOM

Kolejne wersje DOM:

- ◆ DOM Level 0 — poziom zerowy (nieoficjalny) — pochodzi z przeglądarki Netscape Navigator 3.0 i jest zaimplementowany we wszystkich przeglądarkach internetowych.

- ◆ DOM Level 1 — poziom pierwszy (oficjalny) — poziom odpowiedzialny za tworzenie, edycję oraz dołączanie węzłów i atrybutów; składa się z dwóch części: Core i HTML.
- ◆ DOM Level 2 — poziom drugi (oficjalny) — składa się z sześciu specyfikacji:
 - ◆ DOM Level 2 Core,
 - ◆ DOM Level 2 Views,
 - ◆ DOM Level 2 Events,
 - ◆ DOM Level 2 Style,
 - ◆ DOM Level 2 Traversal and Range,
 - ◆ DOM Level 2 HTML.
- ◆ DOM Level 3 — poziom trzeci (oficjalny) — składa się z następujących specyfikacji:
 - ◆ DOM Level 3 Core,
 - ◆ DOM Level 3 Events,
 - ◆ DOM Level 3 Load and Save,
 - ◆ DOM Level 3 XPath,
 - ◆ DOM Level 3 Validation.
- ◆ DOM Level 4 — ciągle w przygotowaniu.

JavaScript

JavaScript to wieloplatformowy język skryptowy do tworzenia aplikacji po stronie klienta, gdzie kod jest interpretowany i wykonywany. Najczęściej wykorzystywany jest jako uzupełnienie strony internetowej o interaktywność poprzez reagowanie na zdarzenia, np. animacje czy zmianę treści podczas przeglądania. Pozwala również na dynamiczne dodawanie funkcjonalności, walidację formularzy, komunikację z serwerem i bazą danych oraz obsługę zdarzeń w dokumentach HTML. Operacje po stronie klienta mają na celu wygodę użytkownika, a przy okazji odciążają w pewnym stopniu serwer. Wszystkie dane pochodzące od użytkownika (również dane przekazane za pomocą formularzy) należy traktować jako potencjalnie niebezpieczne.

Składnia języka

Składnia języka jest podobna do składni C/C#, natomiast sam język jest zorientowany obiektywnie i dynamicznie typowany, podobnie jak PHP. Język jest bardzo prosty, dlatego zostaną w skrócie opisane tylko najważniejsze elementy.

Podstawowe informacje o JavaScriptie:

- ◆ kod JavaScript umieszczamy w tagach <script></script>;
- ◆ skrypt może zostać umieszczony w elemencie <body>² lub <head> (przeglądarka może wstrzymywać ładowanie i wyświetlanie strony, dopóki nie wykona skryptu);
- ◆ skrypt może zostać wczytany z pliku (kolejność osadzania dowolnej liczby skryptów ma znaczenie — będą wykonywane w kolejności umieszczenia na stronie):

```
<script type="text/javascript" src="skrypt.js"></script>
```

- ◆ istnieje możliwość skorzystania z mechanizmu zdarzeń, np. za pomocą zdarzenia onclick (przez kliknięcie przycisku myszą);
- ◆ komentarz pojedynczej linii:

```
//zakomentowana linia
```

- ◆ komentarz wieloliniowy:

```
/* zakomentowana linia1  
   zakomentowana linia2 */
```

- ◆ deklaracje zmiennych (dynamicznych) za pomocą słowa kluczowego var:

```
var zmienna = 1;  
zmienna = "Napis";
```

- ◆ deklaracje tablic:

```
var tablica = new Array();  
tablica[0] = "napis1";  
tablica[1] = "napis2";  
tablica[2] = "napis3";  
var tablica = new Array("napis1","napis2","napis3");  
var tablica = ["napis1","napis2","napis3"];
```

- ◆ deklaracje typów zmiennych za pomocą słowa kluczowego new:

```
var nazwa = new String;  
var numer = new Number;  
var bool = new Boolean;  
var tab = new Array;  
var ob = new Object;
```

² W przypadku stosowania skryptu bezpośrednio w kodzie strony należy umieścić skrypt na końcu strony (przed znacznikiem zamkającym <body>). Zaletą tego rozwiązania jest to, że wczytane zostanie najpierw to, co istotne, czyli treść i wygląd strony (kod HTML). Skrypt, ładowany na samym końcu, będzie w chwili wykonywania widział wszystkie elementy strony, co pozwala uniknąć błędu, którego powodem wystąpienia jest niemożność odwołania się do jeszcze nieistniejącego elementu.

deklaracje obiektów:

```
var auto = {marka:"Mercedes", model:"C Klasa",
    ↳pojemnosc:1998};
```

- ◆ dostęp do właściwości i metod po znaku kropki:

```
auto.marka = "Audi";
```

- ◆ deklaracja funkcji:

```
function nazwaFunkcji(arg1,arg2)
{
    // Ciało funkcji
}
```

- ◆ dostępne operatory arytmetyczne: +, -, *, /, %, ++, --;
- ◆ dostępne operatory przypisania: =, +=, -=, *=, /=, %=;
- ◆ dostępne operatory porównania: ==, ===, !=, !==, >, <, >=, <=;
- ◆ dostępne operatory logiczne: &&, ||, !;
- ◆ dostępne instrukcje warunkowe:

- ◆ if-else:

```
if (warunek){
    // Kod wykonany, gdy true
} else {
    // Kod wykonany, gdy false
}
```

- ◆ switch:

```
switch (zmienna){
    case 0:
        x="Gdy zmienna = 0";
        break;
    case 1:
        x="Gdy zmienna = 1";
        break;
    default:
        x="Gdy zmienna różna od 0 i 1";
}
```

- ◆ pętle:

- ◆ for:

```
for (var i=0;i<zmienna.length;i++){
    document.write(zmienna[i] + "<br>");
}
```

- ◆ while:

```
while (zmienna.length<5){
    document.write(zmienna[i] + "<br>");
}
```

◆ do while:

```
do{  
    document.write(zmienna[i] + "<br>");  
}  
while (zmienna.length<5)
```

◆ for in:

```
var auta = { marka: "Audi", model: "A3", pojemosc: 2500 };  
  
for (x in auta) {  
    txt = txt + auta[x];  
}
```

◆ obsługa błędów:

```
try{  
    //Jakiś kod  
}catch(err){  
    //Przechwytywanie błędów  
}
```

- ◆ nie ma klas w JavaScriptie, ponieważ język bazuje na prototypach, a nie na klasach;
- ◆ wszystkie liczby są zapisywane jako 64-bitowe;
- ◆ zmienne tekstowe string posiadają właściwość length.

Możliwości JavaScriptu

Możliwości JavaScriptu to m.in.:

◆ dopisywanie do dokumentów HTML:

```
document.write("<h1>To jest nagłówek</h1>");
```

◆ zmiana zawartości dokumentów HTML:

```
x = document.getElementById("IdElementu").innerHTML =  
    ↪"Nowa zawartość";
```

◆ obsługa zdarzeń:

```
<button type="button"  
    ↪onclick="alert('Kliknijcie')">Kliknij!</button>
```

◆ zmiana stylów dokumentu HTML:

```
x = document.getElementById("IdElementu")  
    ↪.style.color="#ffffff";
```

◆ walidacja formularzy;

◆ zmienna string:

- ◆ dostęp do określonego znaku w ciągu znaków za pomocą indeksu:

```
var znak = napis[3];
```

- ◆ znajdowanie określonego ciągu znaków w innym ciągu (zwraca indeks):

```
var napis = "Hello World!";  
var n = napis.indexOf("World");
```

- ◆ zamiana części tekstu na inny:

```
var napis = "Hello World!";  
var napisPolska = napis.replace("World", "Polska");
```

- ◆ zmiana wielkości liter:

```
var napis = "Hello World!";  
var napis1 = txt.toUpperCase();  
var napis2 = txt.toLowerCase();
```

- ◆ pozostałe metody w string:

```
charAt(), charCodeAt(), concat(), fromCharCode(), indexOf(),  
lastIndexOf(), match(), replace(), search(), slice(), split(), substr(),  
substring(), toLowerCase(), toUpperCase(), valueOf();
```

◆ wyrażenia regularne RegExp:

- ◆ exec() — zwraca poszukiwany ciąg znaków; jeśli nie znajdzie, zwraca null,

- ◆ test() — szuka ciągu — jeśli znajdzie, zwraca true; jeśli nie znajdzie, zwraca false;

◆ dostęp do okna przeglądarki: window.innerHeight, window.innerWidth,
window.open(), window.close(), window.moveTo(), window.resizeTo();

◆ dostęp do rozdzielczości ekranu użytkownika: screen.availWidth,
screen.availHeight;

◆ dostęp do informacji o lokalizacji: location.port (80, 443), location.hostname,
location.pathname, location.protocol (HTTP, HTTPS);

◆ dostęp do historii przeglądania: history.back(), history.forward();

◆ tworzenie i zapisywanie plików cookie — ciasteczka.

JQuery

JQuery to biblioteka JavaScript pozwalająca na dużo szybsze (działający, uporządkowany i zwięzły kod, bez dużej potrzeby skupiania się na kompatybilności kodu dla każdej przeglądarki) tworzenie aplikacji w języku JavaScript kosztem nieznacznie mniejszej wydajności. Ilość kodu napisanego w jQuery w stosunku do kodu o tej samej funkcjonalności napisanego w czystym JavaScriptie jest dużo mniejsza. JQuery pozwala

na prostą manipulację stylami CSS, elementami HTML DOM oraz obsługę zdarzeń, walidację formularzy, animacji i zapytań asynchronicznych AJAX. Biblioteka jQuery jest wspierana przez wszystkie najpopularniejsze przeglądarki.

Biblioteka występuje w dwóch wersjach: standardowej oraz *.min*. Wersja *.min* to biblioteka jQuery skompresowana tak, aby plik miał jak najmniejszą objętość. Cały kod jest pisany bez przerw i białych znaków, zmienne są zastąpione najkrótszymi możliwymi nazwami bądź znakami. W wersji *.min* nie można wprowadzać praktycznie żadnych zmian do biblioteki, ponieważ jest bardzo nieczytelna. Pełna wersja jQuery to biblioteka z odpowiednim formatowaniem kodu i białymi znakami, pozwalająca na modyfikacje.

Składnia jQuery wygląda następująco:

```
$ (selektor).action()
```

gdzie:

- ◆ \$ — znak dostępu do jQuery (można użyć także słowa *jQuery* w miejscu znaku \$),
- ◆ selektor — służy do przeszukiwania dokumentu i znajdowania określonych części kodu,
- ◆ . — łączenie poszczególnych instrukcji do wykonania,
- ◆ action() — metoda dostępna w jQuery, operująca na elementach odpowiadających selektorowi.

jQuery ma bardzo zwięzłą konstrukcję skryptów, do której wykorzystuje składnię łańcuchową. Każdy taki łańcuch rozpoczyna się znakiem \$, a polecenia łączone są w łańcuchy za pomocą kropki.

Instalacja jQuery

Aby użyć biblioteki jQuery, należy ją pobrać i dołączyć w sekcji <head> jak zwykły plik z kodem JavaScript lub użyć biblioteki bezpośrednio z CDN (ang. *Content Delivery Network*). Biblioteka jQuery jest hostowana zarówno przez Microsoft, jak i przez Google. Przykład dołączenia biblioteki z Google przedstawia listing D.1.

Listing D.1. Dołączenie biblioteki JQuery

```
<head>
<script src="http://ajax.googleapis.com/ajax/libs/jquery/1.5.2/jquery.min.js">
</script>
</head>
```

Selektory i filtry

Zasada działania polega na znajdowaniu elementów w dokumencie, a następnie odpowiednim działaniu na tych elementach. Aby odnaleźć interesujące elementy na stronie, używane są selektory bądź filtry. W zależności od tego, jaki rodzaj argumentów zostanie podany do metody \$(), możliwe ona realizować różnego rodzaju operacje.

Selektory to ciąg znakowy (znany z CSS), który pozwala odnieść się do wybranego elementu z drzewa DOM w dokumencie. Dostępne selektory:

- ◆ selektory ogólne:
 - ◆ # — wyszukuje elementy po id (element o podanym id powinien być tylko jeden w całym dokumencie, id powinno być unikalne),
 - ◆ . — wyszukuje elementy po klasie (elementów jednej klasy może być więcej niż jeden),
 - ◆ * — wszystkie elementy,
 - ◆ this — obecny element,
 - ◆ div — wszystkie *divy* (analogicznie inne znaczniki HTML);
- ◆ selektory formularzy:
 - ◆ :text — element input z atrybutem type ustawionym na text:
`<input type="text">,`
 - ◆ :submit — element input z atrybutem type ustawionym na submit:
`<input type="submit">,`
 - ◆ :checkbox — element input z atrybutem type ustawionym na checkbox:
`<input type="checkbox">,`
 - ◆ :radio — element input z atrybutem type ustawionym na radio:
`<input type="radio">,`
 - ◆ :input — wszystkie elementy typu input,
 - ◆ :checked — zaznaczone pola wyboru,
 - ◆ :selected — zaznaczone elementy listy wyboru.

Selektory można łączyć z filtrami, co rozszerza ich możliwości. Filtry umożliwiają stosowanie wyrażeń klasyfikujących pobrane elementy. Nazwę filtra poprzedza się znakiem dwukropka. Dostępne filtry:

- ◆ :first — pierwszy element,
- ◆ :last — ostatni element,
- ◆ :eq(index) — konkretny element (indeksowanie od 0),
- ◆ :odd — nieparzyste elementy,
- ◆ :even — parzyste elementy,
- ◆ :contains(text) — element zawierający podany tekst,
- ◆ :hidden — elementy ukryte oraz pola typu `<input type="hidden"/>`,
- ◆ :visible — elementy widoczne.

Przykłady użycia selektorów:

```
$(".klaśElementu:odd").toggle();
$(".klaśElementu:last").toggle();
```

Zdarzenia

JQuery obsługuje wiele zdarzeń (tabela D.1). Przykładowo zdarzenie \$(document).ready jest wywoływanie wcześniej niż \$(window).load i należy je stosować np. w sytuacji, w której nie można dopuścić do wyświetlenia danego elementu. Przy użyciu \$(window).load grafika, która jest usuwana ze strony, zostałaby w całości wczytana, na moment wyświetlona użytkownikowi i dopiero wtedy usunięta ze strony. Jednocześnie jeśli operujesz na elementach takich jak np. obrazki, wykorzystanie zdarzenia \$(window).load daje pewność, że obrazki zostały już załadowane na stronie.

Tabela D.1. Zdarzenia dla document i window

Nazwa	Opis
ready()	Wywoływanie w momencie wczytania dokumentu HTML i przygotowania wszystkich obiektów DOM
load()	Wywoływanie po przygotowaniu obiektów DOM oraz po ich pełnym wczytaniu
unload()	Wywoływanie, gdy okno przeglądarki jest zamknięte lub kiedy użytkownik przechodzi do nowej strony przez pasek adresu bądź kliknięty link
resize()	Wywoływanie, gdy rozmiar okna przeglądarki został zmieniony
scroll()	Wywoływanie podczas przewijania strony
error()	Wywoływanie, kiedy wystąpi błąd zwrócony przez żądanie HTTP

Delegacja zdarzeń przydaje się najczęściej, gdy strona bazuje na JavaScriptie, a w szczególności na AJAX, gdzie treści wczytywane są dynamicznie. Cała idea polega na zarejestrowaniu danego zdarzenia gdzieś wyżej w hierarchii dokumentu. Wiele zdarzeń w przeglądarce „bąbelkuje” ku górze. Do delegacji zdarzeń jQuery oferuje następujące metody:

- ◆ bind(),
- ◆ delegate(),
- ◆ on(),
- ◆ one().

Biblioteka jQuery oprócz łatwego korzystania ze standardowych zdarzeń pozwala także na tworzenie własnych. Nowe zdarzenie można zadeklarować, korzystając z metody bind(). Aby wywołać takie zdarzenie, należy użyć metod trigger() lub triggerHandle(). Obie metody przyjmują za argument nazwę zdefiniowanego zdarzenia oraz dodatkowe parametry, które są potrzebne do obsługi zdarzenia, i można je wykorzystać do obsługi standardowych zdarzeń. Różnica polega na tym, że metoda triggerHandle() wykonuje zdarzenie tylko dla pierwszego dopasowania i nie zapewnia jego propagacji do nadrzędnych elementów DOM.

Efekty w postaci animacji

Biblioteka jQuery udostępnia także różne metody dające efekty specjalne. Lista dostępnych efektów:

- ◆ `show()` — pokazuje element,
- ◆ `hide()` — ukrywa element,
- ◆ `toggle()` — pokazuje lub ukrywa element w zależności od obecnego stanu,
- ◆ `slide()` — pokazuje lub ukrywa element poprzez rozsuwanie (`slideDown()`, `slideUp()`, `slideToggle()`),
- ◆ `fade()` — pokazuje lub ukrywa element poprzez zmianę widoczności (`fadeIn()`, `fadeOut()`, `fadeToggle()`, `fadeTo()`),
- ◆ `animate()` — pozwala na tworzenie skomplikowanych animacji,
- ◆ `stop()` — zatrzymuje animacje,
- ◆ `callback` — drugi parametr w funkcjach, który zostaje wywołany po zakończeniu wcześniejszej funkcji.

Metody

Metody związane z wymiarami elementów:

- ◆ `width()` — ustawia lub zwraca szerokość elementu (bez padding, border i margin),
- ◆ `height()` — ustawia lub zwraca wysokość elementu (bez padding, border i margin),
- ◆ `innerWidth()` — ustawia lub zwraca szerokość elementu (z padding, bez border i margin),
- ◆ `innerHeight()` — ustawia lub zwraca wysokość elementu (z padding, bez border i margin),
- ◆ `outerWidth()` — ustawia lub zwraca szerokość elementu (z padding i border, bez margin),
- ◆ `outerHeight()` — ustawia lub zwraca wysokość elementu (z padding i border, bez margin),
- ◆ `outerWidth(true)` — ustawia lub zwraca szerokość elementu (z padding, border i margin),
- ◆ `outerHeight(true)` — ustawia lub zwraca wysokość elementu (z padding, border i margin).

Pozostałe metody:

- ◆ `text()` — zwraca lub ustawia treść z bądź do określonego elementu,
- ◆ `html()` — zwraca lub ustawia treść razem z kodem HTML z bądź do określonego elementu,

- ◆ `val()` — zwraca lub ustawia wartość z bądź do pola wejściowego `input`,
- ◆ `attr()` — ustawia lub zmienia wartość atrybutu,
- ◆ `append()` — wstawia zawartość na końcu wybranego elementu,
- ◆ `prepend()` — wstawia zawartość na początku wybranego elementu,
- ◆ `before()` — wstawia zawartość przed wybranym elementem,
- ◆ `after()` — wstawia zawartość po wybranym elemencie,
- ◆ `remove()` — usuwa element razem z podelementami (ang. *child elements*),
- ◆ `empty()` — usuwa elementy dzieci z wybranego elementu,
- ◆ `addClass()` — dodaje klasy do wybranych elementów,
- ◆ `removeClass()` — usuwa klasy z wybranych elementów,
- ◆ `toggleClass()` — usuwa lub dodaje elementy do wybranych elementów,
- ◆ `css()` — ustawia lub zwraca style wybranych elementów.

Przechodzenie po elementach HTML

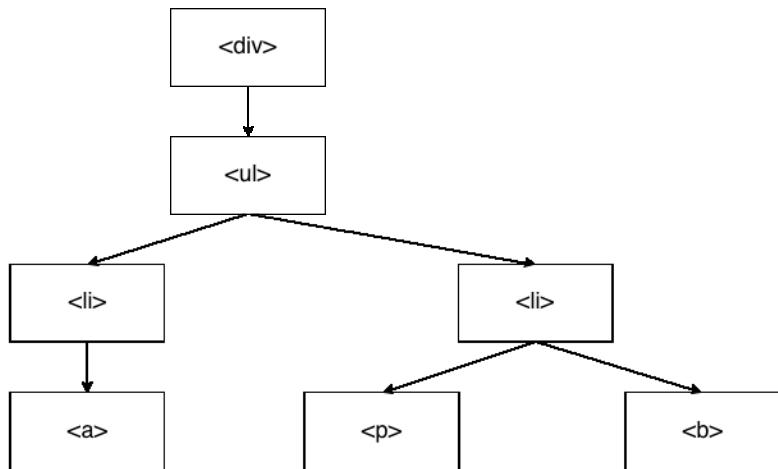
Zależności i przechodzenie po elementach (rysunek D.2):

- ◆ <div> jest rodzicem (ang. *Parent*) dla i przodkiem (ang. *Ancestor*) dla wszystkich elementów w liście,
- ◆ jest rodzicem dla elementów i dzieckiem (ang. *Children*) dla elementu <div>,
- ◆ lewy element jest rodzicem dla <a>, dzieckiem oraz potomkiem (ang. *Descendant*) dla <div> i ,
- ◆ elementy są rodzeństwem (ang. *Siblings*), mają wspólnego rodzica.

Metody do przechodzenia po elementach:

- ◆ przodkowie (ang. *Ancestors*):
 - ◆ `parent()` — zwraca element rodzica,
 - ◆ `parents()` — zwraca wszystkich rodziców aż do znacznika <html>,
 - ◆ `parentsUntil()` — zwraca wszystkich przodków aż do wybranego elementu,
- ◆ potomkowie (ang. *Descendants*):
 - ◆ `children()` — zwraca tylko dzieci (tylko jeden poziom),
 - ◆ `find()` — zwraca wszystkich potomków,
- ◆ rodzeństwo (ang. *Siblings*):
 - ◆ `siblings()` — zwraca całe rodzeństwo,
 - ◆ `next()` — zwraca kolejny element z rodzeństwa,

Rysunek D.2.
Struktura, zależności
i przechodzenie
po elementach



- ◆ `nextAll()` — zwraca wszystkie kolejne elementy z rodzeństwa,
- ◆ `nextUntil()` — zwraca kolejnych członków rodzeństwa aż do określonego elementu,
- ◆ `prev()` — zwraca poprzedni element z rodzeństwa,
- ◆ `prevAll()` — zwraca wszystkie poprzednie elementy z rodzeństwa,
- ◆ `prevUntil()` — zwraca poprzednich członków rodzeństwa aż do określonego elementu.

JQuery UI

JQuery UI to rozszerzenie biblioteki jQuery o elementy graficzne interfejsu użytkownika (gotowe kontrolki), takie jak:

- ◆ suwaki,
- ◆ menu rozwijane,
- ◆ przeciaganie elementów (ang. *Drag and Drop*),
- ◆ rozszerzalne pola tekstowe,
- ◆ autouzupełnianie,
- ◆ kalendarz (ang. *Datepicker*) z wyborem daty,
- ◆ podpowiedzi w dymkach (ang. *Tooltip*).

JQuery Mobile

JQuery Mobile to biblioteka podobna do JQuery UI, jednak przeznaczona na urządzenia mobilne. JQuery Mobile opiera się na HTML 5 i CSS 3. W skład biblioteki wchodzą gotowe elementy interfejsu użytkownika dostosowane do urządzeń mobilnych. Ponie-

waż nie wszystkie przeglądarki wspierają w całości HTML 5, niektóre elementy biblioteki działają tylko w wybranych przeglądarkach³.

AJAX

AJAX (ang. *Asynchronous JavaScript and XML*) pozwala na asynchroniczną komunikację z serwerem (optymalny sposób transportu danych) i obsługa zdarzeń bez konieczności przeładowywania całego dokumentu (strony WWW). AJAX jest niezależny od platformy i przeglądarki. AJAX nie jest technologią, lecz zbiorem technik programistycznych, pewnym podejściem do programowania webowego. Bazuje na języku JavaScript oraz żądaniach HTTP. Typowe zastosowania AJAX to:

- ◆ dynamiczne menu — przydatne, gdy drzewo nawigacji ma dużą liczbę liści;
- ◆ autouzupełnianie — przewidywanie fraz tekstu w polach tekstowych mające na celu przyspieszenie procesu wpisywania tekstu (zaimplementowane m.in. w wyszukiwarkach Google i Wikipedia);
- ◆ autozapisywanie — automatyczne zapisywanie zawartości pola tekstowego bez pytania o to użytkownika;
- ◆ podział na strony lub organizowanie dużej liczby wyników — gdy przez zapytanie, takie jak w wyszukiwarce, zwracane są duże ilości danych, AJAX może pomóc w ich sortowaniu, organizowaniu oraz wyświetlaniu;
- ◆ natychmiastowa komunikacja z innymi użytkownikami w aplikacji webowej.

Rodzaje żądań AJAX:

- ◆ asynchroniczne — wykonywane na akcję użytkownika, a żądanie zwracane jest z wywołaniem zwrotnym, nie blokując interfejsu;
- ◆ synchroniczne — wykonywane automatycznie co określony interwał czasu, a żądanie wykonywane jest bezpośrednio ze zwracaniem wartości, co może powodować blokowanie interfejsu.

Odpitywanie cykliczne AJAX (ang. *AJAX Polling*) (rysunek D.3):

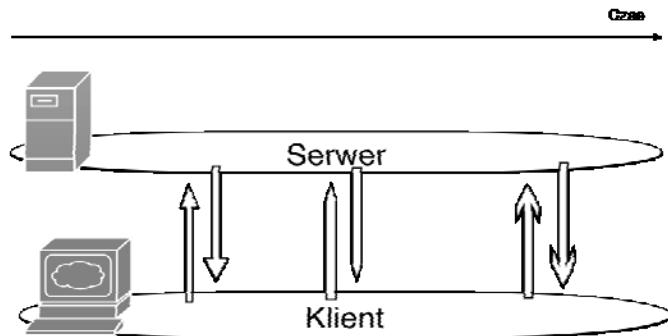
- ◆ klient wysyła zapytanie o stronę WWW do serwera, tak jak przy użyciu HTTP;
- ◆ żądana strona wykonuje kod JavaScript, który wysyła zapytania o dane z serwera w regularnych odstępach czasu;
- ◆ serwer wysyła odpowiedź do klienta, tak jak przy użyciu HTTP.

„Długie” odpitywanie cykliczne AJAX (ang. *AJAX Long-Polling*) (rysunek D.4) polega na tym, że:

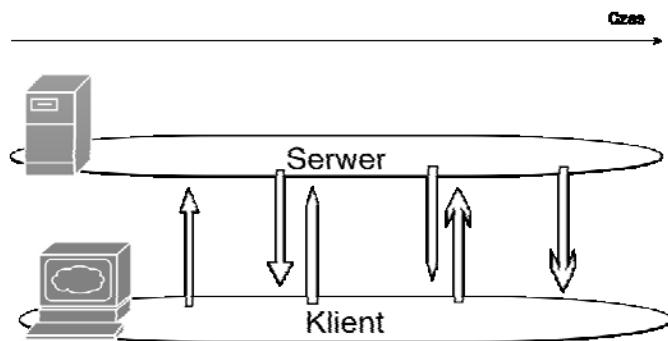
- ◆ klient wysyła zapytanie o stronę WWW do serwera, tak jak przy użyciu HTTP;
- ◆ żądana strona wykonuje kod JavaScript, który wysyła zapytania o dane z serwera w regularnych odstępach czasu;

³ <http://learn.jquery.com/>

Rysunek D.3.
AJAX Polling



Rysunek D.4.
AJAX Long-Polling



- ◆ serwer nie wysyła od razu odpowiedzi do klienta, ale czeka na pojawienie się nowych danych;
- ◆ gdy pojawiają się nowe dane, serwer odpowiada, tak jak przy użyciu HTTP;
- ◆ klient otrzymuje nowe dane i natychmiast wysyła kolejne żądanie do serwera.

Dane przekazywane za pomocą AJAX nie muszą być przesyłane w formacie XML. Alternatywą są czysty tekst bądź format JSON.

JSON

JSON (ang. *JavaScript Object Notation*) to niezależny od języka, prosty, tekstowy format wymiany danych komputerowych (typ MIME: application/json). Jego obsługę zapewniają dodatkowe pakiety bądź biblioteki. JSON to format tekstowy („lżejszy”, bardziej wydajny, czytelniejszy, łatwiejszy do generowania i parsowania w porównaniu do XML), który stanowi podzbiór języka JavaScript. JSON używa składni języka JS do opisu obiektów. W obiekcie żądania XMLHttpRequest pobierany jest jako tekst, który może zostać przekształcony w obiekt przy użyciu funkcji `JSON.parse()`. W przypadku JSON do pobranych danych można się odwoływać jak do obiektu. Składnia danych JSON opiera się na parach: nazwa – wartość ("Imie" : "Adam"). Obiekty w JSON to nieuporządkowany zbiór par nazwa – wartość oddzielonych przecinkami, umieszczonych w nawiasach klamrowych ({ "Imie" : "Adam" , "Nazwisko" : "Kowalski" }). Tablice w JSON to zbiór obiektów oddzielonych przecinkami, umieszczonych w nawiasach kwadratowych (listing D.2).

Listing D.2. Przykładowa tablica w JSON

```
{  
    "Tablica" : [  
        { "Imie" : "Adam" , "Nazwisko" : "Kowalski" },  
        { "Imie" : "Jerzy" , "Nazwisko" : "Nowak" },  
        { "Imie" : "Kamil" , "Nazwisko" : "Mazur" }  
    ]  
}
```

Typy wartości w JSON:

- ◆ łańcuch znakowy (string) — umieszczony w cudzysłowie,
- ◆ liczba,
- ◆ wartość true,
- ◆ wartość false,
- ◆ wartość null,
- ◆ obiekt,
- ◆ tablica.

„Opakowany” JSON (ang. *JSON with Padding*), czyli JSONP, to format, który pozwala na zniesienie ograniczenia co do możliwości tworzenia zapytań do innych domen. Dodatkowo umożliwia komunikację dwukierunkową oraz pobieranie obiektów JSON na żądanie.

XMLHttpRequest

Dane wymieniane są za pośrednictwem XMLHttpRequest, a żądania mogą być przesyłane zarówno metodą POST, jak i GET. Tworzenie obiektu XMLHttpRequest (w skrócie jqXHR) jest proste: ObiektXMLHttp = new XMLHttpRequest();. Obiekt ten, tak jak standardowy obiekt XMLHttpRequest, zawiera takie właściwości jak: responseText, responseXML, a także metody: getResponseHeader() i overrideMimeType().

Metody XMLHttpRequest to:

- ◆ open(method,url,async) — określa typ i parametry połączenia, gdzie:
 - ◆ method — GET lub POST,
 - ◆ url — lokalizacja pliku na serwerze,
 - ◆ async — gdy true, to żądanie asynchroniczne; gdy false, to synchroniczne,
- ◆ send() — wysyła żądanie do serwera,
- ◆ abort() — zatrzymuje żądanie,
- ◆ setRequestHeader() — wysyła wybrany nagłówek HTTP,
- ◆ getResponseHeader("nazwa_nagłówka") — pobiera wybrany nagłówek HTTP,

- ◆ getAllResponseHeaders() — pobiera jako string wszystkie wysłane nagłówki HTTP.

Właściwości XMLHttpRequest to:

- ◆ onreadystatechange — przetrzymuje funkcję lub nazwę funkcji, która jest uruchamiana w chwili zmiany stanu połączenia,
- ◆ readyState — zawiera aktualny status połączenia, gdzie wartość:
 - ◆ 0 — połączenie nienawiązane,
 - ◆ 1 — połączenie nawiązane,
 - ◆ 2 — żądanie odebrane,
 - ◆ 3 — przetwarzanie żądania,
 - ◆ 4 — dane zwrócone i gotowe do użycia,
- ◆ responseText — zawiera zwrócone dane jako tekst,
- ◆ responseXML — zawiera zwrócone dane w formacie XML,
- ◆ status — zwraca status połączenia, gdzie: 404 — strona nie istnieje, a 200 — OK.

Przykładowy kod nawiązujący połączenie z serwerem i podmieniający treść w *divie* prezentuje listing D.3.

Listing D.3. Nawiązywanie połączenia z serwerem

```
<script>
    function loadXMLDoc() {
        var xmlhttp;
        if (window.XMLHttpRequest) {
            xmlhttp = new XMLHttpRequest();
        }
        else {//Dla IE6, IES
            xmlhttp = new ActiveXObject("Microsoft.XMLHTTP");
        }
        xmlhttp.onreadystatechange = function () {
            if (xmlhttp.readyState == 4 && xmlhttp.status == 200) {
                document.getElementById("nazwaDiv").innerHTML =
                    xmlhttp.responseText;
            }
        }
        xmlhttp.open("GET", "tekst.txt", true);
        xmlhttp.send();
    }
</script>
```

AJAX w jQuery

Biblioteka jQuery posiada funkcje upraszczające korzystanie z AJAX. Udostępnia zdarzenia globalne i lokalne. Zdarzenia globalne można podpiąć do dowolnego elementu drzewa DOM, natomiast lokalne są definiowane dla danego obiektu żądania.

Najważniejsze funkcje w jQuery związane z AJAX:

- ◆ `$.ajax()` — najbardziej rozbudowana metoda do wywoływania żądań AJAX;
- ◆ `$.ajaxPrefilter()` — przetrzymuje lub edytuje ustawienia, zanim zostaje wysłane żądanie oraz przed wywołaniem metody `$.ajax()`;
- ◆ `$.ajaxSetup()` — ustawia domyślne wartości dla żądania AJAX;
- ◆ `$.ajaxTransport()` — tworzy obiekt przechowujący dane aktualnie transmitowane przez AJAX;
- ◆ `$.get()` — ładuje dane z serwera za pomocą żądania HTTP GET;
- ◆ `$.getJSON()` — ładuje dane w formacie JSON z serwera za pomocą żądania HTTP GET;
- ◆ `$.getScript()` — ładuje kod JavaScript z serwera za pomocą żądania HTTP GET;
- ◆ `$.param()` — serializuje⁴ tablice lub obiekt, aby można było z nich skorzystać w żądaniu lub parametrze;
- ◆ `$.post()` — ładuje dane z serwera za pomocą żądania HTTP POST;
- ◆ `ajaxComplete()` — określa funkcje do uruchomienia po wykonaniu żądania AJAX;
- ◆ `ajaxError()` — określa funkcje do uruchomienia, gdy wykonanie żądania AJAX kończy się błędem;
- ◆ `ajaxSend()` — określa funkcje do uruchomienia przed wysłaniem żądania AJAX;
- ◆ `ajaxStart()` — określa funkcje do uruchomienia, kiedy zostaje wywołane pierwsze żądanie AJAX;
- ◆ `ajaxStop()` — określa funkcje do uruchomienia po wykonaniu wszystkich żądań AJAX;
- ◆ `ajaxSuccess()` — określa funkcje do uruchomienia po pomyślnym wykonaniu żądania AJAX;
- ◆ `load()` — ładuje dane z serwera i umieszcza w wybranym elemencie;
- ◆ `serialize()` — koduje elementy `form` jako `string`;
- ◆ `serializeArray()` — koduje elementy `form` jako tablice nazw i wartości.

⁴ Serializacja to proces przekształcania obiektów, np. instancji określonych klas, do postaci szeregowej, czyli w strumień bajtów, z zachowaniem aktualnego stanu obiektu. Serializowany obiekt może zostać utrwalony w pliku dyskowym, przesłany do innego procesu lub innego komputera poprzez sieć. W uproszczeniu: jest to zapis obiektu z pamięci operacyjnej do pliku lub ciągu znaków, które można przesyłać.

Dodatek E

Bazy nierelacyjne

Bazy nierelacyjne (NoSQL) nie wymagają sztywnego schematu bazy danych, przez co nie korzystają z operacji *join*. Bazy nierelacyjne posiadają lepszą skalowalność, wysoką wydajność oraz większą elastyczność. Dane, które w standardowej bazie relacyjnej byłyby rozrzucone po kilku tabelach, w bazie nierelacyjnej znajdują się w jednym dokumencie, który nie ma stałej struktury. Aby dodać nowe pole, dopisuje się je do dokumentu. W bazie relacyjnej konieczna byłaby zmiana struktury bazy danych. Dokumenty zapisywane są zazwyczaj w formacie binarnym (ang. *Binary*) JSON. Bazy nierelacyjne większość danych przechowują w pamięci RAM, dzięki czemu są bardzo szybkie. Bazy NoSQL rozwiążają pewne problemy baz relacyjnych i z tego względu są najczęściej używane razem z bazami relacyjnymi, jednak nie zastępują ich całkowicie.

Podział baz danych ze względu na zastosowanie:

- ◆ bazy przechowujące dane głównie w pamięci (Memcache, Redis),
- ◆ bazy nastawione na strukturę dokumentów (MongoDB),
- ◆ bazy nastawione na zapis danych (Cassandra),
- ◆ bazy nastawione na replikację danych, czyli współdzielenie danych pomiędzy serwerami (CouchDB),
- ◆ bazy grafowe (Neo4j),
- ◆ zwykłe bazy relacyjne (MS SQL, MySQL).

Podział baz danych ze względu na strukturę:

- ◆ bazy klucz – wartość — struktura słownika (Riak),
- ◆ bazy kolumnowe — dane są zapisywane w kolumnach, a nie w wierszach (Cassandra),
- ◆ bazy dokumentowe — dane są zapisywane jako dokumenty, których struktura opiera się na parach klucz – wartość (MongoDB),
- ◆ bazy obiektowe — dane są zapisywane jako obiekty i nie wymagają ORM.

Do zalet baz NoSQL należą: wysoka wydajność dla prostych operacji i zapisu, skalowalność, replikacja danych, elastyczność, łatwość użycia, brak konieczności tworzenia schematu (struktura tworzy się sama w czasie rozbudowy aplikacji). Poza tym w większości są one darmowe.

Do wad baz NoSQL należą: brak wsparcia dla transakcji, młoda technologia, niewielka liczba programistów znających bazy NoSQL, brak powiązań (operacji *join*), brak wyszukiwania pełnotekstowego, słaba wydajność przy generowaniu złożonych raportów, brak standardowego języka zapytań, a duplikacje powodują niespójność danych.

MongoDB

MongoDB to darmowa baza dokumentowa (NoSQL). Ma wysoką wydajność, wspiera replikację oraz skalowalność. Nie wspiera wyszukiwania pełnotekstowego. Dane są przechowywane w formacie JSON. Dostępna jest na wszystkie platformy i wspiera większość języków.

Oficjalna strona MongoDB:

<http://www.mongodb.org/>

MongoDB w ASP.NET MVC:

<http://www.joe-stevens.com/2011/10/02/a-mongodb-tutorial-using-c-and-asp-net-mvc/>

RavenDB

RavenDB to płatna baza dokumentowa NoSQL na platformę .NET. Działa tylko na platformie Windows i wspiera jedynie języki dostępne w .NET. W przeciwieństwie do MongoDB wspiera wyszukiwanie pełnotekstowe oraz posiada wiele innych dodatkowych udogodnień.

Oficjalna strona RavenDB:

<http://ravendb.net/>

Dodatek F

Podstawy pozycjonowania w Google

Pozycjonowanie jest procesem prowadzącym do uzyskania jak najwyższej pozycji danej witryny internetowej w wyszukiwarkach internetowych dla podanych w zapytaniu fraz. Fraza to nic innego jak słowo lub kilka słów kluczowych, które użytkownicy wpisują w oknie wyszukiwarki. Użytkownik korzystający z wyszukiwarki otrzymuje tysiące wyników dla konkretnego zapytania. Średnio około 95% ruchu (kliknięć) pochodzi z pierwszej strony w wyszukiwarce, na której znajduje się 10 wyników. Pozycja, na której znajduje się konkretna strona, zależy głównie od liczby linków prowadzących do danej domeny. Im więcej linków prowadzi do strony, tym „głośniej” jest o niej w internecie, czyli tym bardziej jest ona popularna. W ostatnim czasie znaczenie liczby odnośników stale maleje, ale wciąż jest to główny element pozwalający na ustalenie pozycji strony lub podstrony w wynikach wyszukiwania. Obecnie najpopularniejszą wyszukiwarką jest Google, która posiada ponad 90% udziałów w światowym rynku oraz 97% na rynku polskim. Taka dominacja jednej wyszukiwarki powoduje, że jest ona praktycznie bezkonkurencyjna, a pozycjonowanie opiera się na „odkrywaniu” algorytmu tylko tej jednej wyszukiwarki. Obecnie SEO (ang. *Search Engine Optimization*) jest jedną z najskuteczniejszych i najbardziej opłacalnych technik reklamy w internecie. W wyszukiwarce Google oprócz wyników organicznych (bezpłatnych, obliczanych przez algorytm) można znaleźć linki sponsorowane (płatne) wyświetlające się zwykle na samym początku listy wyników. Poza wynikami organicznymi oraz linkami sponsorowanymi można trafić na wyniki powiązane z mapą Google. Są to zwykłe wyniki regionalne, pokazujące się po wpisaniu nazwy miasta w zapytaniu, dostępne jedynie dla najpopularniejszych regionalnych fraz.

Metatagi

Metatagi są zbiorem znaczników (w części nagłówkowej dokumentu) używanych do opisu zawartości danej strony lub podstrony, powszechnie stosowanym przy tworzeniu stron internetowych w formacie HTML. Jeśli korzysta się z wyszukiwarki w drzewie

wyników wyszukiwania, otrzymuje się tytuł oraz krótki opis dla każdej ze stron. Są to opisy ustawione w metatagach, które umieszcza się w sekcji `<head>` stron w języku HTML.

Znacznik `<title>`

Tytuł znajduje się w znaczniku `<title>` i powinien zawierać do 60 znaków, ponieważ jest to maksymalna liczba znaków, które można zobaczyć w tytule na liście wyników wyszukiwania. Tytuł jest jednocześnie odnośnikiem do strony WWW:

```
<title>Tytuł strony</title>
```

Opis strony

Opis widoczny pod tytułem jest zazwyczaj opisem ustawionym w tagu `description` i powinien zawierać do 160 znaków. Jeśli nie ma ustawionego tagu `description` lub gdy opis nie pasuje do wyszukiwanej frazy, wyszukiwarka wyświetla najbardziej odpowiedni w stosunku do zapytania tekst ze strony WWW:

```
<meta name="Description" content="Opis strony">
```

Słowa kluczowe

Poza tymi dwoma tagami występuje jeszcze tag `keywords`, czyli słowa kluczowe, które dawniej były podstawą dla wyszukiwarek, jednak aktualnie straciły swoją moc i nie mają praktycznie żadnego wpływu na miejsce zajmowane przez stronę w wyszukiwarce:

```
<meta name="Keywords" content="Słowa kluczowe">
```

Wartości `noindex` i `nofollow`

Poza wymienionymi wyżej tagami bardzo istotny jest również tag `robots`, który pozwala na sterowanie zachowaniem wyszukiwarek w zakresie pobierania i indeksowania stron. Aby strona była widoczna w wyszukiwarce (zaindeksowana), należy nie ustawiać tego tagu (domyślnie ma wartość pozwalającą na indeksowanie) lub ustawić go na wartości `index` i `follow`, które pozwalają na zaindeksowanie zawartości strony. W przypadku gdy nie chce się, aby zawartość była widoczna w wyszukiwarce, należy ustawić wartości na `noindex` i `nofollow`:

```
<meta name="robots" content="noindex, nofollow">
```

Znaczniki HTML

Najważniejszą rolę w kodzie strony WWW pełni znacznik `<h1>`, dlatego powinien zawierać treść ściśle powiązaną z pozycjonowanymi frazami. Poza znacznikiem `<h1>` duże znaczenie mają także znaczniki od `<h2>` do `<h6>` oraz `<i>`, ``, `` i `<u>`.

Wszystkie wyżej wymienione znaczniki mają za zadanie zwrócić uwagę odwiedzających stronę, a co za tym idzie — mają większe znaczenie przy pozycjonowaniu niż zwykły tekst.

Linkowanie

Linkowanie jest najważniejszą częścią procesu pozycjonowania. Linki ze względu na moc pozycjonerską można podzielić na dwa typy:

- ◆ linki z atrybutem `rel="follow"` (pozycjonujące) — przekazujące *Page Rank* dla linkowanej domeny,
- ◆ linki z atrybutem `rel="nofollow"` — nieprzekazujące *Page Rank* dla linkowanej domeny.

Linki z atrybutem `follow` są bardzo cenne i pozycjonowanie opiera się głównie na pozyskiwaniu tego typu linków. Linki z atrybutem `nofollow` jeszcze do niedawna miały bardzo małe znaczenie, jednak ostatnimi czasy są bardzo cenne, ponieważ pozwalają zabezpieczyć domenę przed wpadnięciem w filtr.

Kolejnym bardzo ważnym elementem linków są *anchory*. *Anchor* to tekst, który po kliknięciu przenosi nas na linkowaną stronę. *Anchory* powinny odpowiadać pozycjonowanym przez nas frazom, ponieważ po dodaniu linku strona pozycjonowana jest na frazy związane właśnie z tym *anchorem* lub słowami w nim zawartymi. Aby zabezpieczyć się przed filtrem w wyszukiwarce, należy różnicować *anchory*.

Oprócz wyżej wymienionych czynników bardzo ważne są również źródła pochodzenia linków. Im wyższy *Page Rank* strony linkującej oraz im mniejsza liczba linków wychodzących z tej strony, tym większą wartość ma link do strony. Wskaźnik *Page Rank* jest jedynym oficjalnym wskaźnikiem podawanym przez Google, jednak nie zawsze jest on zgodny z rzeczywistą wartością strony, przez co w ostatnim czasie stracił w dużej mierze znaczenie w pozycjonowaniu. Aktualnie ważniejszym wskaźnikiem jest wskaźnik *Trust Flow* (zaufania) — nie jest on oficjalnie podawany przez Google, jednak istnieje wiele serwisów przeliczających ten wskaźnik w zbliżony sposób do wyszukiwarki Google.

Najważniejszym czynnikiem w pozycjonowaniu jest zachowanie naturalności linkowania. Linki powinny być dodawane stopniowo i w miarę regularnie. Serwis powinien się ciągle rozwijać, a co za tym idzie — ciągle zdobywać nowe linki.

Kolejnym ważnym aspektem są rozszerzenia domen. Naturalnie domeny najdroższe w utrzymaniu posiadają zazwyczaj najbogatszą treść i linki z takich stron mają większą wartość w pozycjonowaniu. Należy zapewnić różnorodność rozszerzeń domen linkujących. W przypadku niespełnienia tych warunków strona bardzo szybko może dostać karę i zniknąć z wyników wyszukiwarki.

Zaplecze, katalogi stron i precle

Zdobywanie linków z cudzych stron jest dość trudnym zadaniem, dlatego powstały katalogi stron internetowych. Są to moderowane serwisy oferujące umieszczenie linku oraz krótkiego opisu na temat dodawanej do spisu strony za pewną opłatą.

Oprócz katalogów tworzone są strony zapleczowe, tzw. precle, działające na zasadzie blogów. Pozwalają one na zamieszczenie dłuższego opisu wraz ze zdjęciami i kilkoma linkami w treści. Ponieważ linki te są oblane tekstem, mają przez to większą moc w pozycjonowaniu oraz pozwalają na stosowanie dowolnych *anchorów* w linkach. Aby zapewnić jak najniższe koszty utrzymania takich stron, zamieszczane są zazwyczaj na najtańszych domenach regionalnych. Najlepsze precle powinny być umieszczone na domenach z rozszerzeniami *.pl* lub *.net*.

Poza rozszerzeniami nazw domen ważną rolę pełnią adresy IP serwerów, na których utrzymywane są linkujące strony. Ważne jest, aby strony były zamieszczone na różnych hostingach z różnymi adresami IP. Linki prowadzące z różnych domen, ale z jednego IP nie mają tak dużej mocy jak te same linki pochodzące z różnych domen i różnych adresów. Po raz kolejny w grę wchodzą kwestie finansowe. Utrzymując strony na jednym hostingu (pojedynczym adresie IP), udałoby się zaoszczędzić pieniądze, jednak nie jest to naturalne i bardzo łatwe do wykrycia przez wyszukiwarkę Google.

Bardzo ważną rolę pełnią skrypty, na których bazują katalogi oraz platformy blogowe (precle). W zależności od skryptu można używać innych znaczników HTML oraz wstawiać różną liczbę linków.

Skrypty katalogów

Najpopularniejsze skrypty, na których oparte są katalogi stron, to:

- ◆ WebMini,
- ◆ Sinooke,
- ◆ SeoKatalog,
- ◆ OtwartyMini,
- ◆ Freeglobes,
- ◆ Qlweb,
- ◆ EasyDirectory,
- ◆ DonMini,
- ◆ PKSI.

Skrypty blogowe

Najpopularniejsze skrypty, na których oparte są strony typu *pressel page*, to:

- ◆ WordPress,
- ◆ OmniPress,
- ◆ OmniPressv2,
- ◆ BuddyPress,
- ◆ Articems.

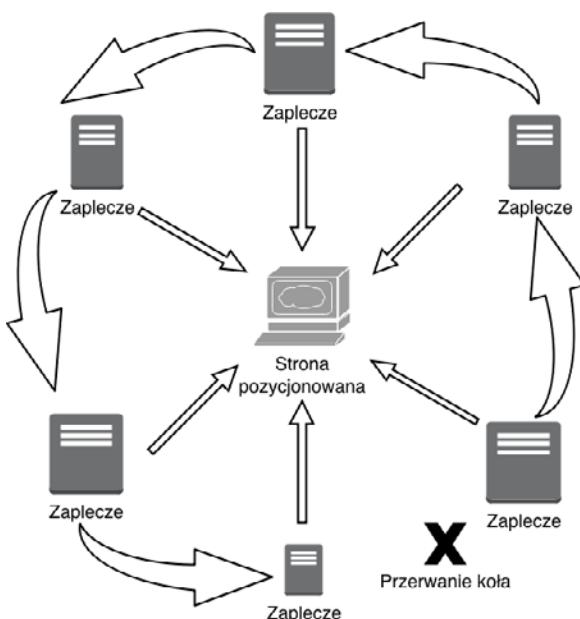
Schematy linkowania

Aby wykorzystać maksymalnie wartość stron zapleczowych i prowadzących z nich linków, zostały wymyślone schematy linkowania.

Schemat koła

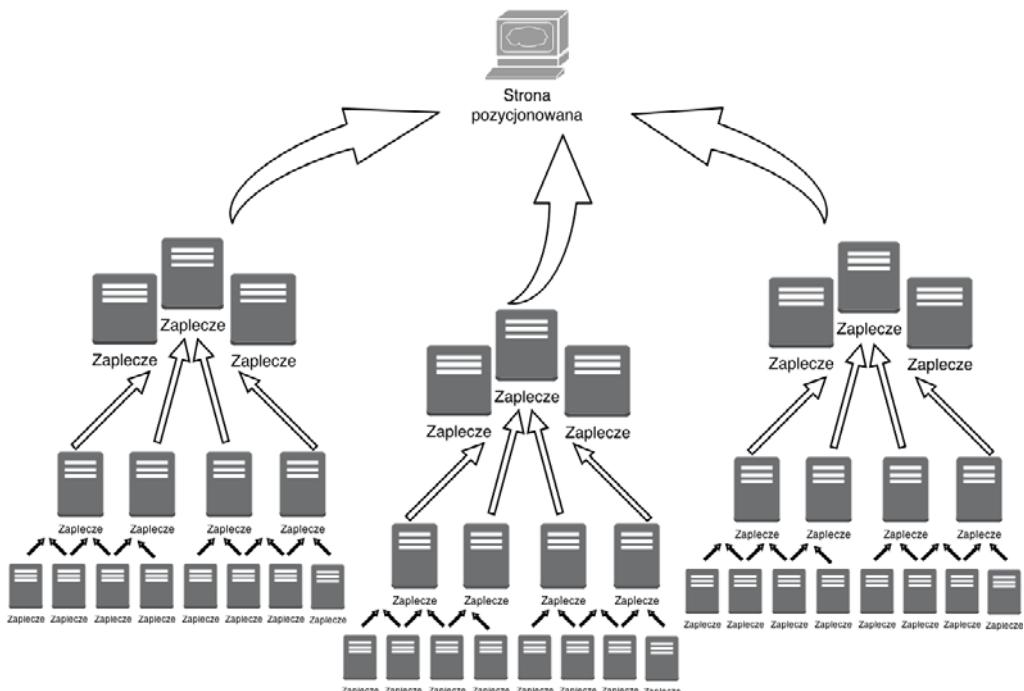
Linkowanie w kole (ang. *Link Wheel*) jest jednym z najpopularniejszych schematów (rysunek F.1). Opiera się na jednostronnym linkowaniu domen zaplecza pomiędzy sobą. Koło musi być w pewnym momencie przerwane, ponieważ w przypadku zamkniętego koła istnieje duże prawdopodobieństwo wykrycia nienaturalnego linkowania. Jedyną wadą tego rozwiązania jest to, że podążając za kolejnymi linkami, można odkryć całe zaplecze.

Rysunek F.1.
Schemat koła



Schemat piramidy

Schemat piramidy (rysunek F.2) opiera się na kilku poziomach domen. Do strony pozycjonowanej linkują zazwyczaj domeny z dwóch górnych poziomów lub linkuje jedyne ostatni poziom. Schemat ten ma kilka zalet. Tego typu linkowanie jest najbardziej naturalne, pozwala również przekazywać moc z domen niższych (gorszej jakości) bez narażenia witryny pozycjonowanej. Co charakterystyczne dla linkowania w piramidzie, żadna z domen wyższego poziomu nie linkuje do domen niższego poziomu. Jest to również element naturalny — bardzo rzadko się zdarza, aby dobre domeny linkowały do słabszych, powszechnie jest natomiast linkowanie słabszych stron do popularnych, dobrych witryn.



Rysunek F.2. Schemat piramidy

Gwiazda

Ten schemat jest bardzo podobny do kołowego, tyle że domeny nie linkują między sobą. Jest to najprostsze linkowanie, niewymagające większej kontroli — w przypadku utraty linków nie zaburza schematu (co ma znaczenie w przypadku koła i piramidy). Strony nie są ze sobą również powiązane, co zwiększa bezpieczeństwo, jednak zmniejsza moc linków prowadzących do domeny.

Schematy mieszane

Poza wyżej wymienionymi najbardziej znanymi schematami istnieją też mieszane, czyli połączenie schematu kołowego z piramidami, rozbudowany schemat kołowy lub zupełnie dowolne linkowanie bez wyraźnej struktury. Brak wyraźnej struktury sprawia, że linkowanie wygląda na naturalne, jednak w praktyce może być problematyczne podczas kontroli oraz późniejszej rozbudowy.

Linkowanie wewnętrzne

Poza linkami pochodzącyimi z innych stron bardzo ważną rolę pełnią linki wewnętrzne na stronie, czyli linki kierujące do podstron serwisu. Przez Google każda podstrona jest traktowana jak osobna strona, dlatego bardzo często podstrony bez żadnych linków pochodzących z innych domen są bardzo wysoko w wyszukiwarce. Zakładając, że posiada się stronę, która ma tysiąc podstron, a na każdej podstronie zamieszczony jest link do konkretnej podstrony, to otrzymuje się 1000 linków dla tej podstrony. Z linkowaniem wewnętrznym bardzo ściśle związana jest technika pozycjonowania na „długi ogon”. Podobnie jak w linkowaniu z różnych domen, tak samo w linkowaniu wewnętrznym można wyróżnić różnego typu schematy linkowania. Należy tutaj zauważyć, że nie powinno się przekraczać 100 linków na jednej podstronie, ponieważ niektóre roboty sieciowe nie radzą sobie z tak dużą liczbą odnośników.

„Długi ogon”

Strategia „długiego ogona” (ang. *Long Tail*) opiera się na pozycjonowaniu strony na bardzo dużą liczbę słów kluczowych, lecz mniej popularnych i z mniejszym ruchem. Skoro wypozycjonowanie kilku ogólnych wyrażeń kluczowych może być bardzo trudne lub, ze względu na bardzo dużą konkurencję, niemożliwe, to lepiej pozycjonować słowa mniej popularne, za to w ilości hurtowej. Bardziej szczegółowych słów kluczowych zwykle używają internauci zainteresowani konkretną informacją lub ofertą, dzięki czemu stronę odwiedzają osoby, które z dużym prawdopodobieństwem skorzystają ze znajdującej się na stronie oferty, co powoduje, że strategia „długiego ogona” jest najczęściej stosowana w sklepach internetowych i przynosi lepsze efekty w postaci wyższej konwersji sprzedażowej oraz niższego współczynnika odrzuceń.

Bardzo ważnym aspektem jest również stopień trudności wypozycjonowania takich fraz — bardziej niszowe słowa kluczowe są łatwiejsze i tańsze do wypozycjonowania i w większości wypadków wystarcza dobrze podlinkowana strona główna z dobrym linkowaniem wewnętrznym, aby uzyskać bardzo wysokie pozycje podstron na frazy z „długiego ogona”. Najważniejszą korzyścią z pozycjonowania „długiego ogona” jest fakt, że serwis staje się odporny na wahania pozycji ważniejszych słów kluczowych. Nawet jeśli kilka słów nie wejdzie do czołówki wyników, to i tak serwis pozostanie widoczny dla internautów ze względu na wysokie pozycje pozostałych słów. Bardzo

często zdarzają się kary i filtry za zbyt mocne lub nienaturalne linkowanie, przez co strona główna znika z wyszukiwarki. W tym przypadku pozostałe podstrony pozycjonowane na „długi ogon” nadal pozostają na swoich pozycjach.

Przyjazne adresy URL — Friendly URL

Przyjazne adresy URL (ang. *Friendly URL*) to adresy stron bez widocznych parametrów. Adres składa się jedynie ze słów lub liczb. Przyjazne adresy URL powstały, aby zwiększyć estetykę, użyteczność oraz optymalizację stron pod wyszukiwarki. Pozwalają się domyślić tematu danej podstrony na podstawie jej adresu.

Zwykłe adresy URL:

`http://adres.pl/index.php?page=ubania`
`http://adres.pl/spodnie/index.php?page=damskie`

Przyjazne adresy URL:

`http://adres.pl/ubrania`
`http://adres.pl/spodnie/damskie`

Pliki związane z pozycjonowaniem

robots.txt

Plik *robots.txt* służy do ograniczenia dostępu do witryny robotom indeksującym. Przed pobraniem strony robot sprawdza, czy plik *robots.txt* nie blokuje dostępu do strony. W pliku *robots.txt* podaje się również adres do pliku z mapą strony, a więc *sitemap.xml*. Reguły stosowane w pliku *robots.txt*:

- ◆ User-agent — określa robota, którego dotyczy dana reguła,
- ◆ Allow/Disallow — określa treści lub adres URL, który chcemy odblokować bądź zablokować,
- ◆ Sitemap — adres do pliku z mapą witryny,
- ◆ Request-rate — maksymalna prędkość, z jaką robot może pobierać witrynę,
- ◆ Visit-time — godziny, w których robot może odwiedzać witrynę.

Przykładowy plik *robots.txt*:

```
User-agent: *
Allow: /
User-agent: GoogleBot
Disallow: /podstrona/
# Mapa strony
Sitemap: http://adres.pl/sitemap.xml
```

gdzie:

* — oznacza, że wszystkie roboty mają dostęp do witryny,

Allow: / — określa, że cała witryna jest dostępna dla robota, nie blokuje żadnej treści,

User-agent: GoogleBot — odnosi się do narzędzia Google indeksującego treść stron,

Disallow /podstrona/ — blokuje jedną podstronę dla robota Google.

sitemap.xml

Mapa strony (*sitemap.xml*) to plik XML zawierający informacje na temat adresów URL witryny. Ułatwia indeksację podstron w wyszukiwarce, zawiera informacje o dacie ostatniej aktualizacji, ważności oraz częstości aktualizacji. W przypadku bardzo dużych map witryn stosuje się kompresję pliku do formatu *.gzip*. Mapa strony nie może przekraczać 10 MB lub 50 000 adresów URL. Przykładowa mapa strony znajduje się na rysunku F.3¹.

Rysunek F.3.

Mapa strony
— sitemap.xml

```
<?xml version="1.0" encoding="UTF-8" ?>
<urlset xmlns="http://www.sitemaps.org/schemas/sitemap/0.9">
    <url>
        <loc>http://www.adres.pl/</loc>
        <lastmod>2013-01-01</lastmod>
        <changefreq>monthly</changefreq>
        <priority>0.8</priority>
    </url>
</urlset>
```

Parametry mapy strony to:

- ◆ <loc> — zawiera adres podstrony,
- ◆ <lastmod> — data ostatniej aktualizacji,
- ◆ <changefreq> — częstotliwość zmian podstrony, dostępne wartości:
 - ◆ always — zawsze,
 - ◆ hourly — co godzinę,
 - ◆ daily — codziennie,
 - ◆ weekly — co tydzień,
 - ◆ monthly — co miesiąc,
 - ◆ yearly — corocznie,
 - ◆ never — nigdy,
- ◆ <priority> — określa ważność strony; obowiązuje skala ważności od 0 do 1.

¹ <http://www.sitemaps.org/faq.html>

.htaccess

.htaccess (*hypertext access*) to plik konfiguracyjny działający dla katalogu i podkatalogów, w których się znajduje. Pozwala na zdecentralizowane zarządzanie konfiguracją serwera. Plik ma bardzo dużą liczbę funkcjonalności:

- ◆ kontrola dostępu do zasobów — określa zasady dostępu do katalogu, w którym się znajduje (odnosi się również do podkatalogów);
- ◆ ograniczanie dostępu do zasobów — udostępnia możliwość zabezpieczania dostępu do katalogu lub strony WWW za pomocą hasła; do zabezpieczenia hasłem wymagany jest plik *.htpasswd*;
- ◆ definiowanie strony startowej — pozwala wybrać, który plik będzie startowym; domyślnie są to pliki *index.xxx*, gdzie *xxx* to rozszerzenie pliku (np. *.html*, *.php*, *.cshtml*);
- ◆ definiowanie stron błędów — obsługa błędów 404, 400, 403, 500; wyświetla odpowiednią stronę błędu;
- ◆ definiowanie *MimeType* — *MimeType* to rozszerzenia, które są zwracane przeglądarkce WWW po wywołaniu znajdującego się na serwerze typu pliku;
- ◆ definiowanie ustawień listowania — zmiana opcji listowania zawartości katalogu;
- ◆ blokowanie użytkowników po IP bądź hoście — prosty sposób na zablokowanie niechcianych użytkowników;
- ◆ przekierowania — przekierowanie użytkownika z jednego adresu URL na inny; dostępne rodzaje przekierowań:
 - ◆ permanent — zwraca status 301 — przekierowuje na inny adres,
 - ◆ temp — zwraca status 302 (*Moved Temporarily*) oraz wykonuje przekierowanie,
 - ◆ seeother — zwraca status 303 (*See Other*),
 - ◆ forbidden — przekierowanie z błędem 403 (*Forbidden*),
 - ◆ notfound — przekierowanie z błędem 404 (*Not Found*),
 - ◆ gone — zwraca status 410 (*Gone*),
 - ◆ error — zwraca status 500 (*Internal Error*);
- ◆ przepisywanie adresów (*mod_rewrite*) — pozwala na zamianę zwykłych adresów na przyjazne adresy URL;
- ◆ Cache — pozwala na kontrolę cachowania przez przeglądarki internetowe w celu zredukowania obciążenia serwera, transferu i opóźnień.

Filtры и кары

Aby zapobiec oszustwom i masowemu zdobywaniu sztucznych linków, wyszukiwarka posiada zaimplementowane algorytmy pozwalające na nakładanie kar i obniżanie pozycji stron w wynikach wyszukiwania. Głównym powodem, dla którego implementowane są algorytmy służące do karania stron, są pozycjonerzy. Gdyby nie było pozycjonerów, kary nie byłyby potrzebne. Kary można podzielić na te nakładane ręcznie oraz nakładane automatycznie.

Ban

Pierwszym i zarazem najbardziej popularnym stwierdzeniem używanym w odniesieniu do kar Google jest tzw. *ban*. Najczęściej określenie to jest zdecydowanie nadużywane i stosowane bez względu na rodzaj kary, jaką faktycznie otrzymała witryna. Dla wielu użytkowników *ban* jest także synonimem wszystkich kar Google i najgorszą rzeczą, jaka może się przytrafić pozycjonowanej domenie. W praktyce *ban* jest faktycznie najsuwoszą formą kary, jednak jest przyznawany bardzo rzadko, bo jedynie w sytuacjach bardzo poważnego naruszenia regulaminu wyszukiwarki, czyli wskazówek dla webmasterów. *Ban* powoduje całkowite usunięcie witryny z wyników wyszukiwania — zarówno strony głównej, jak i podstron. Przyczyny nakładania *banów*:

- ◆ ukryty tekst,
- ◆ ukryte linki,
- ◆ dziesiątki tysięcy subdomen,
- ◆ *Doorway Page* — tworzenie osobnych stron dla każdej frazy i przekierowywanie użytkownika na stronę docelową,
- ◆ przekierowania,
- ◆ *Cloaking* polegający na prezentowaniu innej treści użytkownikom, a innej robotowi indeksującemu witrynę,
- ◆ strony utworzone tylko dla wyłudzenia pieniędzy.

Ban jest nakładany tylko i wyłącznie za zawartość domeny i nie ma związku z linkami przychodząymi ani innymi czynnikami niezależnymi od webmastera. Aby ściągnąć *bana* ze strony, należy usunąć przyczynę i zgłosić stronę do ponownego rozpatrzenia przez Google. Trwa to zazwyczaj miesiąc lub dwa.

Sandbox

Sandbox (piaskownica) to drugi w kolejności bardzo często nakładany filtr, który dla większości pozycjonerów stanowi największe zagrożenie. *Sandbox* to filtr bezpośrednio związany z czynnikiem o największym wpływie na pozycję strony, czyli oczywiście z linkami prowadzącymi do witryny, a dokładniej ich *anchorami*.

Filtr polega na czasowym filtrowaniu i niewyświetlaniu strony internetowej pod konkretną frazą bądź frazami czy słowem kluczowym w wyszukiwarce. Kara jest najczęściej przyznawana „młodym” stronom oraz tym, które są silnie pozycjonowane, zwykle pod jedno słowo kluczowe ze słabej jakości stron. Aby uniknąć kary, należy zadbać o zróżnicowanie *anchorów* oraz dywersyfikację pochodzenia linków. Bardzo ważna jest również liczba dodawanych linków oraz przyrost w czasie. Granica do otrzymania filtra nie jest stała i zazwyczaj trudno ją określić.

Kara trwa od 3 do nawet 12 miesięcy. W tym czasie Google nie bierze pod uwagę linków z określonym *anchorem* (tekstem) i wrzuca je do poczekalni. Bardzo ważne jest wówczas zdobywanie nowych mocnych linków generujących ruch, a co za tym idzie — bardzo wiarygodnych.

Wychodzenie z filtra nie jest łatwym zadaniem. Należy konsekwentnie linkować, stopniowo zmniejszając udział filtrowanych fraz w *anchorach* linków — i tak aż do skutku. Co najważniejsze, po wyjściu z filtra strona zazwyczaj wraca na te same lub jeszcze wyższe pozycje!

Zmiany algorytmu Google

Co pewien czas przeprowadzane są aktualizacje algorytmu wprowadzające do niego zmiany i zazwyczaj nakładające kary na złe pozycjonowane strony.

Panda

Panda jest aktualizacją wyszukiwarki, która analizuje jakość strony bez zagłębiania się w profil linków do niej prowadzących. Ocena jest przeprowadzana na podstawie treści zamieszczonej na stronie ze szczególnym uwzględnieniem kopii treści (wewnątrz strony). Ponadto wykrywa podstrony automatycznie generowane przez portal, a więc puste podstrony kategorii, podstrony tagów oraz indeksowane wyniki wyszukiwarki wewnętrznej.

Strona dotknięta aktualizacją Google Panda nagle traci znaczną część ruchu z wyszukiwarki, co jest spowodowane spadkiem pozycji słów kluczowych w wyszukiwarce. Analiza pozwala na odnalezienie przyczyny spadku i niezwłoczne naprawienie błędów. W zależności od stopnia oraz liczby stron objętych filtrem przywrócenie ruchu może być bardzo czasochłonne i kosztowne.

Pingwin

Pingwin jest aktualizacją wyszukiwarki Google, która analizuje profil linków każdej strony w celu wyeliminowania serwisów pozyskujących nienaturalne linki. W przypadku wykrycia przez algorytm nienaturalnych linków nakładany jest algorytmiczny filtr na domenę, obniżający jej pozycję w wyszukiwarce.

Nienaturalne linki to linki z systemu wymiany linków, linki nienaturalnie pozyskane przez właściciela serwisu lub linki kupione w celach manipulacji pozycjami w wyszukiwarce. Należy tutaj zauważyć, że prowadząc katalog, należy pobierać płatność za rozpatrzenie wpisu, a nie za dodany link. Zapobiega to nałożeniu *bana* na katalog i zmniejszeniu wiarygodności domen, do których dany katalog linkował.

EDM

Jedna z najnowszych aktualizacji Google dotycząca nazwy domeny EDM (ang. *Exact-Match Domain*). Aktualizacja dotyczy stron, które mają nazwę domeny dobieraną pod pozycjonowaną frazę. Domeny dotknęte tą aktualizacją mają zazwyczaj identyczną nazwę jak wyszukiwana fraza — składającą się z więcej niż dwóch słów, zwykle rozdzielonych znakiem -. Przykładowa nazwa domeny dotknięta tą aktualizacją: *zabawki-dla-dzieci.pl*. Jak łatwo się domyślić, strona po przeprowadzonej aktualizacji znika z wyników wyszukiwania frazy: *zabawki dla dzieci*.

Narzędzia związane z pozycjonowaniem

Google Analytics i Google Webmasters Tools

Są to dwa podstawowe narzędzia, które należy uruchomić po stworzeniu nowej strony. Google Analytics to narzędzie online do analizy pochodzenia i zachowania użytkowników odwiedzających naszą stronę. Pozwala na sprawdzenie liczby odwiedzin, pochodzenia odwiedzających, współczynnika odrzuceń oraz wielu innych bardzo przydatnych, a właściwie podstawowych i niezbędnych rzeczy. Google Webmasters Tools pozwala na sprawdzenie witryny pod kątem bardziej technicznym. W przypadku filtra lub *bana* zazwyczaj wysyłane są powiadomienia właśnie na konto w Google Webmasters Tools. Narzędzia dla webmasterów pozwalają również przesyłać mapę witryny, plik *robots.txt* oraz zarządzać parametrami w adresie URL.

Narzędzia do pracy z tekstem

Wyszukiwarka kładzie bardzo duży nacisk na wartościowe oraz unikalne treści, dlatego bardzo ważne są teksty zamieszczone na stronie. Korzystając z programów do automatycznego dodawania wpisów w katalogach, bardzo trudno byłoby utworzyć unikalny tekst dla każdego wpisu, dlatego powstały teksty synonimizowane. Teksty synonimizowane pozwalają na wygenerowanie bardzo dużej liczby unikalnych tekstów bez konieczności pisania każdego tekstu od początku.

Najciekawszym narzędziem do pracy z tekstem jest program SeoLeniwiec mający bardzo długą listę funkcjonalności, pozwalający na automatyczne generowanie synonimizowanych tekstów oraz sprawdzanie unikalności tekstu w wyszukiwarce.

Przykład tekstu synonimizowanego prezentuje listing F.1. Tekst zawiera jedynie dwa zdania, a pozwala na wygenerowanie 2304 unikalnych kombinacji tekstu różniących się przynajmniej jednym słowem.

Listing F.1. Tekst synonimiczny

{Zarówno|Tak} w {domach|budynkach} mieszkalnych, jak {i|również} biurach {i|oraz|tudzież} obiektach {handlowo-usługowych|handlowych|usługowych} {stosowana|wykorzystywana} jest drewniana stolarka drzwiowa. Drzwi wewnętrzne Podkarpackie drewniane są {praktyczne wszędzie|powszechnie} {stosowane|wykorzystywane} {ze względu|z uwagi} na swoją {użyteczność|praktyczność}.

Systemy wymiany linków

System wymiany linków (określany także jako program wymiany linków) to platforma umożliwiająca publikację linków na stronach dodanych przez innych użytkowników. Celem tego działania jest chęć wpłynięcia na rankingi w wyszukiwarkach poprzez zdobycie linków z różnych stron utworzonych wyłącznie dla zamieszczania tego typu linków. Systemy opierają się na punktach przyznawanych za dołączanie nowych witryn do systemu. W zależności od jakości strony naliczana jest proporcjonalna liczba punktów do wykorzystania na umieszczenie linków na stronach u innych użytkowników. Systemy wymiany linków są bardzo niebezpieczną i potępianą przez Google formą pozycjonowania. Korzystanie z tego sposobu linkowania może spowodować nałożenie filtra za nienaturalny profil linków.

Systemy wymiany linków dzielą się na dwie kategorie:

- ◆ rotacyjne — emitujące losowe linki na losowych stronach,
- ◆ stałe — emitujące te same linki na tych samych stronach.

Najpopularniejsze systemy wymiany linków to:

- ◆ Statlink,
- ◆ Gotlink,
- ◆ Linkme,
- ◆ Linkorn.

Półautomaty, „dodawarki” i automaty do postowania

Aby usprawnić zakładanie kont i stawianie opisów oraz linków w formularzach, powstały programy pozwalające na ręczne dodawanie, jednak w przypadku automatycznego wypełniania pól są to tzw. półautomaty. Wymagają one pracy człowieka, która jest bardzo kosztowna.

Aby jeszcze bardziej zautomatyzować i usprawnić dodawanie wpisów w katalogach oraz postów na blogach, powstały programy automatycznie zakładające konta na różnych stronach opartych na wyżej wymienionych skryptach, a następnie dodające wpisy. Najpopularniejszy jest skrypt WordPress i to na nim opiera się większość stron. Automat potrafi zarejestrować konto, odczytać z e-maila podanego przy rejestracji login i hasło, aktywować konto, a następnie dodać wpis zawierający link do strony.

Kolejnym typem automatów są automaty do zakładania profili na forach internetowych oraz innych serwisach pozwalających na zamieszczanie linku do własnej strony w profilu lub stopce pod każdym napisanym postem. Działają one na podobnej zasadzie jak „dodawarki” — rejestrują konto w serwisie i uzupełniają profil, tak aby uzyskać link do strony.

Oprócz dwóch wcześniej wymienionych automatów można wyróżnić różne typy automatów dodających komentarze pod wpisami i artykułami. Aby dodać komentarz, zazwyczaj wystarczy złamać kod obrazkowy *CAPTCHA*, z czym automaty bardzo dobrze sobie radzą. Istnieją również serwisy internetowe łamiące kody za drobną opłatą, które można podpiąć do automatu postępującego. Najpopularniejsze automaty dodające:

- ◆ OmniFLASH,
- ◆ OmniADD,
- ◆ SeoCat,
- ◆ XRummer,
- ◆ DonLinkage,
- ◆ Publiker,
- ◆ Adder,
- ◆ Licorne AIO.

Inne narzędzia

Harvestery to programy służące do przeszukiwania sieci i znajdowania stron zawierających potrzebny kawałek treści lub kodu HTML. Działają na podobnej zasadzie do wyszukiwarki Google, gdyż przechodzą po linkach pomiędzy stronami linkującymi między sobą. Interesujące treści lub części kodu HTML to tzw. *footprinty*. Dla skryptu WordPress jest to np. napis „Powered by WordPress” znajdujący się zazwyczaj w stopce strony. Aby przykładowo uzyskać adresy stron związanych z ubraniami i opartych na skrypcie WordPress, podaje się słowa kluczowe, takie jak „odzież”, „ubrania”, „ciuchy”, oraz *footprinty* odpowiednie dla tego typu skryptu. Po zakończonym wyszukiwaniu listę zwróconych stron dodaje się do automatu, który automatycznie dodaje wpisy z linkami na wcześniej wyszukanych stronach. Najpopularniejszym harvesterem jest program ScrapeBox.

Oprócz wcześniej wymienionych programów służących bezpośrednio do zdobywania linków niezbędne są narzędzia do analizy pozycji i pochodzenia linków. Narzędzia tego typu to zazwyczaj serwisy internetowe, płatne lub bezpłatne, w zależności od

wymagań. Bez sprawdzania pozycji strony na wybrane frazy nie można dobrze pozyjonować witryny. Narzędzia i serwisy służące do sprawdzania pozycji w wyszukiwarce to:

- ◆ *webpozycja.pl*,
- ◆ *stat4seo.pl*,
- ◆ *cmonitor.pl*.

Poza sprawdzaniem pozycji na wybrane frazy niezbędna jest również analiza pochodzenia linków i rozkład *anchorów* prowadzących do witryny. Listę linków do witryny można również pobrać z narzędzi Google dla webmasterów, jednak jest to tylko i wyłącznie lista domen posiadających link do witryny, bez żadnych dodatkowych informacji na temat *anchorów* czy typu linków. Narzędzia służące do analizy linków i popularności domeny to:

- ◆ *suite.searchmetrics.com*,
- ◆ *alexa.com*,
- ◆ *Ahrefs.com*,
- ◆ *serpstat.pl*.

Skorowidz

.NET Framework Data Provider, 147

A

abstrakcja, 53

adnotacje do walidacji, 222

ADO.NET, 146

adresaty atrybutów, 94

Agile, 437

AJAX, 491, 495

AJAX Long-Polling, 492

AJAX Polling, 492

akcja

Create, 353, 379

Delete, 345, 350, 379

Details, 342, 376

Edit, 360, 363, 377

Index, 311, 317, 327, 385

aktualizacja

bibliotek, 281

Google

EDM, 511

Panda, 510

Pingwin, 510

kontrolera, 381, 387

pakietów, 279

szablonu, 365

widoku, 317, 345, 354, 360, 389

aliasy, 82

aliasy zewnętrzne, 83

animacje, 488

API

Drag and Drop, 463

Web Storage, 461

architektura

aplikacji, 334

dwuwarstwowa, 126

jednowarstwowa, 126

n-warstwowa, 126

trójwarstwowa, 126

argumenty

metod, 49

nazwane, 110

arkusz stylów CSS, 467

ASP.NET Identity, 244

ASP.NET MVC, 252

ASP.NET MVC 1, 185

ASP.NET MVC 2, 185

ASP.NET MVC 3, 186

ASP.NET MVC 4, 186

ASP.NET MVC 5, 185, 186

ASP.NET MVC 6, 187

ASP.NET Web API 2, 251

ASP.NET Web Forms, 145

ASP.NET Web Pages, 146

atak

Cross-Site Request Forgery, 236

Cross-Site Scripting, 237

CSRF, 357

SQL Injection, 236

atrybut

Bind, 237

OutputCache, 232

atrybuty, 93

atrybuty Caller Info, 119

dla formularza, 459

globalne, 456

Attribute Routing, 216, 407

autocomplete, 408

automaty do postowania, 512

automatyczna inicjalizacja właściwości, 120

autoryzacja, 242, 372

dostępu, 369

RSA, 448

Azure Service Bus, 144

B

ban, 509

baza danych, 305, 419

niespójność, 310

strategie dziedziczenia, 178

baza dokumentowa

MongoDB, 498

RavenDB, 498

bazy nierelacyjne, 497

bezpieczeństwo, 235, 369

biblioteka

Bootstrap, 329

JQuery, 484

jQuery Unobtrusive, 221

PagedList, 382

binding, 357

BLOB Storage, 143

blok try-catch, 238

błąd, 310, 348

404, 394

dla podstrony, 397

podczas edycji, 378

w dostępie do danych, 395

Browser Link, 270

C

C#, 19

C# 2.0, 20

C# 3.0, 20

C# 4.0, 21

C# 5.0, 21

C# 6.0, 21

cache, 231, 394

cachowanie

częściowe, 233

po stronie klienta, 234

po stronie serwera, 231

rozproszone, 233

w HTML 5, 234

CAPTCHA, 513

ceny

licencji SQL Server, 265

Windows Server 2012, 268

certyfikat

CSR, 450

SSL, 449, 450

aktywacja, 450

instalacja, 450

walidacja, 450

ciasteczka, 243

CIL, Common Intermediate Language, 19, 135

claimy, 245

CLI, Common Language Infrastructure, 135

CLR, Common Language Runtime, 135, 137

Code First, 157, 158

Code First Data annotations, 235

Code First Fluent API, 181

controller, 128

CORS, 255, 256

CRUD, 223

CSS, 330

CSS 3, 466

nowe selektory, 471

nowe właściwości, 472

CSS 4, 473

cykl życia obiektu, 341

czas, 90

D

DAL, Data Access Layer, 157

data, 90

Data Annotations, 181, 235

Data Contract, 140

Data Member, 140

Database First, 157

DDD, 132

debugowanie, 324

aplikacji, 325, 327

metody Seed, 308

obiektu db, 327

definiowanie własnego atrybutu, 256

deklaracja zmiennej, 27

deklaracje, 24

deklaracje inline, 122

delegaty, 72

destruktor, 64

diagram ERD, 159, 413

DIP, 431

DLR, Dynamic Language Runtime, 137

długi ogon, Long Tail, 505

dobre praktyki, 26

dodawanie

domeny, 415

dyrektywy using, 285

encji, 411

kolumny, 413

kontrolera, 224, 312

kontrolerów, 311

modelu, 410

nowego projektu, 295

nowej klasy, 283

referencji, 296

węzłów, 478

widoków, 311

dokument SOAP, 253

DOM, 463, 469

dostępne metody, 477

dostępne właściwości, 479

poziomy, 480

schemat, 478

domena, 415

domena aplikacji, 189
domyślny
 layout, 313
 widok, 402
Drag and Drop, 463
DRY, Don't Repeat Yourself, 435
drzewa wyrażeń, 101
drzewo, 469
DV, Domain Validation, 449
dyrektywa using, 81, 121, 285
dyrektywy preprocesora, 76
dziedziczenie, 54
 TPC, 179
 TPH, 178
 TPT, 178

E

EDA, 133
edytacja ogłoszenia, 378
edytowane pliki, 425
EF 5, 159
EF 6, 160
elementy HTML 5, 456
encje, 411
Entity Framework, 153
Entity Framework 6, 154, 157
Entity SQL, 172
EV, Extended Validation, 449
eXtreme Programming, 439

F

Federated Authentication, 247
filtry, 485, 509
 akcji, 197
 wyjątków, 124
folder
 Models, 294
 Shared, 201
 z widokami, 316
format
 JSON, 406, 493
 XML, 406
formularz, 456
framework Twitter Bootstrap, 473

G

Garbage Collector
 podział na generacje, 66
 zasada działania, 66

generowanie
 kontrolerów, 223, 225
 widoków, 226, 227
geolokalizacja, 465
Google Analytics, 511
Google Webmasters Tools, 511
GRASP, 432

H

HandleError, 239
Harvestery, 513
hasło, 372
hermetyzacja, 54
hierarchia drzewa, 469
hostowanie aplikacji, 143
HTML 5, 453
 atrybuty globalne, 456
 nowe elementy, 456
 sekcje, 457
 SSE, 461
 szablon, 454
 typy pól formularza, 458
 WebSockets, 462
 znaczniki, 454, 459
HTML 5 Application Cache, 234
HTML 5 WebStorage, 234
HTML DOM, 477
HTML helper, 208, 401
HTTP, 443

I

Identity Provider, 246
identyfikacja, 241
IEnumerable, 96
IHttpActionResult, 260
IIS, Internet Information Services, 263
 moduły, 263
 przetwarzanie żądań, 264
 pule aplikacji, 264
implementacja
 .NET, 138
 interfejsu, 69, 395
 klasy repozytorium, 395
 kontrolera, 396
 metody POST, 362
indeksatory, 61
index, 373
inicjalizacja
 pol, 64
 zmiennej, 28

inicjalizatory
 obiektów i kolekcji, 100
 słownikowe, 121
 instalacja
 bibliotek, 298
 biblioteki PagedList, 382
 certyfikatu SSL, 450
 jQuery, 485
 kontenera IoC Unity, 338
 migracji, 300
 pakietów, 298
 instrukcja
 checked, 78
 if, 40, 41
 switch, 41
 unchecked, 78
 instrukcje
 iteracyjne, 43
 skoku, 45
 sterujące, 40
 interfejs, 68, 86
 ICollection<>, 87
 IDictionary< TKey, TValue >, 86
 IEnumerable< T >, 87
 IHttpActionResult, 260
 IList< >, 87
 IOgloszenieRepo, 338
 IQueryable< >, 88
 IoC, 397
 ISAPI, 189
 ISP, 429
 iteratory, 97

J

JavaScript, 271, 352, 480
 JQuery, 484
 możliwości, 483
 składnia, 481
 jawna implementacja interfejsu, 69
 jawne ładowanie, 170
 język C#, 19
 JQuery, 352, 484, 495
 animacje, 488
 filtry, 485
 metody, 488
 selektory, 485
 zdarzenia, 487
 JQuery Mobile, 491
 JQuery UI, 490
 JSON, 406, 493
 tablica, 493
 typy wartości, 493
 JSON Editor, 271
 JSONP, 255

K
 kary, 509
 kaskadowe usuwanie, 348
 katalogi stron, 502
 kategorie, 395
 KISS, 436
 klasa, 46
 ApplicationUser, 288
 FilterConfig, 239
 Kategoria, 286
 Ogloszenie_Kategoria, 287
 OgloszenieRepo, 335
 System.Array, 35
 System.Object, 63
 Uzytkownik, 287
 z kontekstem, 293
 klasy
 częściowe, 103
 generyczne, 84
 klucz
 FK, 166
 obcy, 166
 szyfrujący, 447
 kolekcje, 37
 generyczne, 86
 IEnumerable, 123
 komentarze, 23
 komunikat o błędzie, 222, 351, 380
 konfiguracja
 migracji, 301
 witryny, 417
 konstruktor, 64
 konstruktory pierwotne, 120
 kontekst, 410
 kontener
 IoC, 336, 341
 IoC Unity, 338
 konto FTP, 421
 kontrawarianca, 111, 112
 kontroler, 194, 223
 Web API, 257
 z widokami, 311
 konwencja ponad konfiguracją, 188
 konwencje, 21
 nazewnicze, 25
 w MVC, 187
 konwersja
 jawna, 58
 niejawna, 57
 konwersje typów, 31

L

layout, 367
leniwa inicjalizacja, 114
licencjonowanie
 SQL Server, 265
 Windows Server, 267
linki, 502
linkowanie, 501
linkowanie wewnętrzne, 505
LINQ, 148, 168, 397, 400
LINQ to DataSet, 149
LINQ to Entities, 149
LINQ to Objects, 149
LINQ to SQL, 149
LINQ to XML, 148
lista ogłoszeń, 321, 395, 405
literały, 30
literały binarne, 124
logowanie, 193, 369, 370, 373
logowanie globalne, 240
LSP, 428

Ł

łańcuchy, 36
łączenie stylów CSS, 210

M

manifest Agile, 437
mapowanie żądań na akcje, 257
mechanizm
 refleksji, 93
 Scaffoldingu, 224
metatagi, 499
metoda, 49
 Aktualizuj, 365
 AsNoTracking, 171, 324, 327
 Attach, 165
 Create, 358
 Delete, 348
 Detach, 165
 Details, 342
 Dispose, 313
 Down, 304
 Edit, 359
 Equals, 57
 GET, 446
 GetHashCode, 57
 Index, 388
 OnException, 238
 POST, 362, 446
 POST dla akcji Create, 355

POST dla akcji Delete, 347
RegisterBundles, 209
SaveChanges, 349
Seed, 183, 305
ToList, 325, 328
Up, 302
metody
 anoniemowe, 74
 asynchroniczne, 118, 228, 230
 częściowe, 104
 generyczne, 84
 rozszerzające, 102
 synchroiczne, 228, 230
 zwrotne, 118
metodyka Scrum, 439
Microsoft Azure, 143
Microsoft SQL Server 2014, 264
Microsoft Visual Studio, 423
Microsoft Visual Studio 2013 Ultimate, 268
migracja startowa, 302
migracje, 183, 300, 309
minimalizacja, 209
model, 128, 130, 218
 aktywny, 129
 danych, 157
 First, 157, 158
 pasywny, 129
modele licencjonowania SQL Server, 265
moduły
 Elmah, 241
 IIS, 263
modyfikator private protected, 123
MongoDB, 498
MoSCoW, 437
MVC, Model View Controller, 127, 131, 185
 autoryzacja, 242
 filtry, 198
 identyfikacja, 241
 konwencje, 188
 logowanie błędów, 241
 metody asynchroniczne, 228
 metody synchroniczne, 228
 role, 242
 struktura projektu, 187
 uwierzytelnianie, 241
 wyjątki, 238
MVC Pipeline, 189
 moduły HTTP, 193
 ścieżka wywołań, 189
 uchwyty, 193
MVC Scaffolding, 222
MVP, Model View Presenter, 129, 131
MVVM, Model View ViewModel, 131

N

narzędzia
do pracy z tekstem, 511
ORM, 153
narzędzie NLOG, 240
nawiasy klamrowe, 25
nazwa domeny, 417
nazwy
 kwalifikowane, 80
 niekwalifikowane, 80
NHibernate, 153
NHibernate 3, 154
nowy projekt, 279

O

obiekt, 48
 DataRelation, 147
 DataRow, 147
 DataSet, 147
 DataTable, 147
 DataView, 147
 DbContext, 165
 DbSet, 165
 HttpApplication, 190
obsługa
 błędów, 350, 363
 żądań, 191
obszary, 217
OCP, 428
OData, 254
odkomentowanie kodu, 370
odnajdywanie widoków, 200
odpytywanie
 bazy danych, 168
 cykliczne AJAX, 491
odroczone wykonanie, 325
 ogłoszeniowania użytkownika, 393
okno
 dodawania domeny, 416
 dodawania widoku, 207, 400
 NLOG, 240
 Output, 326
 potwierdzania wyboru, 352
Properties, 414
tworzenia bazy danych, 420
tworzenia użytkownika, 421
Watch, 326
wyboru
 szablonu widoku, 228
 trybu działania witryny, 419

typu domeny, 417
typu widoku, 228
z błędami, 241
zarządzania
 bazą danych, 420
 kontami FTP, 421
 witryną, 418
opakowywanie, 32
opcja
 Reference script libraries, 227
 Strongly Typed, 207
OpenAuth, 250
OpenId, 249
operacja join, 400
operacje
 CRUD, 223
 współbieżne, 176
Operation Contract, 140
operator, 38
 ??, 40
 trójargumentowy, 38
opis strony, 500
optymalizacja
 listy ogłoszeń, 322
 pod kątem pozycjonowania, 331
 pod kątem SEO, 329
 SEO, 343
OV, Organization Verification, 449
OWIN, 244

P

Page Inspector, 269
pakiet PagedList.Mvc, 382
pakiety, 282, 299
panel użytkownika, 415
parametr binding, 237
parametry
 akcji, 196
 opcjonalne, 111
pętla
 do while, 43
 for, 44
 foreach, 45
 while, 43
piaskownica, 509
plik
 .htaccess, 508
 _Layout.cshtml, 211
 BundlesConfig.cs, 209
 robots.txt, 506
 sitemap.xml, 507
PLINQ, Parallel LINQ, 151

- pobieranie danych, 220, 321, 326
po stronie klienta, 261
po stronie serwera, 261
tokena, 246
podejście Model First, 409
podpowiedzi składni, 319
podświetlanie wierszy, 330
podział wymagań klienta, 437
pole, 48
polimorfizm, 55
połączenie z bazą danych, 422
z serwerem, 494
poprawa architektury aplikacji, 334
wyglądu, 329
postęp publikacji, 424
potwierdzanie tożsamości, 447
powiązania pomiędzy tabelami, 412, 413
pozycjonowanie, 331, 499
.htaccess, 508
Friendly URL, 506
linkowanie, 501
linkowanie wewnętrzne, 505
metatagi, 499
narzędzia, 511
nofollow, 500
noindex, 500
robots.txt, 506
schematy linkowania, 503
sitemap.xml, 507
słowa kluczowe, 500
znaczniki HTML, 500
prefiksy, 215
presenter, 130
problem N+1, 170
procedury składowane w EF, 173
programowanie
 ekstremalne, 439
 metodologie, 437
 obiektywne, 53
 zasady, 427
projekt
 Mono, 139
 Repozytorium, 296
protokół
 SOAP, 253
 SSL, 447
przechodzenie po elementach HTML, 489
przeciążanie
 metod, 61
operatorów, 56
 arytmetycznych, 60
konwersji, 57
logicznych, 58
relacji, 57
przeniesienie
 metody, 334
pliku, 299
 zapytania LINQ, 334
przestrzeń nazw, 79
przestrzeń System.IO, 92
przeszukiwanie sieci, 513
przetwarzanie żądań, 264
przyciski Bootstrap, 329
przyjazne adresy URL, 506
pseudoklasy
 adresu, 475
 czasu, 475
formularza, 475
logiczne, 473
struktury drzewa, 475
struktury siatki, 475
pseudouwierzytelnianie, 250
publikacja, 424
publikacja systemu, 414
pule aplikacji, 264
punkt przerwania, 326
puste linie, 24

Q

Queue Storage, 143

R

- RavenDB, 498
razor, 202
refleksja, 93, 95
regiony, 290
reguła CSS, 468
rejestracja
 użytkownika, 369
 z użyciem Google, 371
relacja
 FK, 166
 jeden do jednego, 161
 jeden do wielu, 161
 wiele do wielu, 162, 408
relacje
 niezależne, 166, 167
 opcjonalne, 165
 poprzez klucz obcy, 166
relacyjne bazy danych, 160
repozytorium, 277, 335, 340, 410
repozytorium generyczne, 278
REST, 253

rodzaje

arkuszy stylów:, 468

certyfikatów, 449

role, 242, 372

routing, 214, 316

ignorowanie ścieżek, 214

kolejność, 214

na podstawie atrybutów, 215, 252

obszary, 217

ograniczenia, 215

w Web API, 257

rozpakowywanie, 32

RP, Relying Party, 246

rzutowanie, 31

S

sandbox, 509

Scaffolding, 224

schemat

dla Federated Authentication, 248

DOM, 478

formularza, 455

gwiazdy, 504

koła, 503

kontrolera, 225

piramidy, 504

wywołań, 192

schematy

linkowania, 503

mieszane, 505

Scrum, 439

sekcja featured, 213

sekcje, 211, 457

selektory, 471, 485

SEO, 329, 331, 343, 499

separacja zagadnień, 436

separatory cyfr, 124

serializacja, 495

Server Side Events, 461

Service Bus Queue, 145

Service Bus Relay, 144

Service Bus Topic, 145

Service Contract, 140

serwer

hostingowy, 414

IIS, 263

serwis z ogłoszeniami, 278

sesje, 243

silnik Razor, 203

Silverlight, 142

składnia

metod, 150

zapytań, 151

skróty klawiszowe, 272

skrypty, 209

blogowe, 503

katalogów, 502

słowo kluczowe, 29

as, 114

async, 118, 230

await, 118, 124, 230

base, 106

in, 111

is, 114

out, 111

task, 230

this, 106

typeof, 114

using, 71

yield return, 97

snippety, 269

SOA, 132

SOAP, 253

SoC, Separation of Concern, 436

SOLID, 427

sortowanie, 387

cyfr, 464

ogłoszeń, 392

sprawdzanie

danych, 308

wartości null, 123

SQL Logging, 179

SQL Server 2014, 266

SQL Server Management Studio, 422

SRP, 427

SSL/TLS, 447

stan aplikacji, 242

strona

edycji, 377

ufająca, 246

stronicowanie, 381

struktura, 67

.NET, 135

bazy danych, 305

dokumentu, 470

katalogów, 463

projektów, 280

projektu Repozytorium, 299

STS, 246

systemy wymiany linków, 512

szablon, 225

_Layout.cshtml, 365

dokumentu HTML 5, 454

S

śledzenie zmian, 175
dynamiczne, 176
migawkowe, 175

T

tablica
ograniczeń, 217
znaków szczególnych, 30
Table Storage, 143
tablica, 34
tablica typów, 28
TDD, Test Driven Development, 440
TempData, 204
testy, 440
TLS, Transport Layer Security, 448
token, 246
TPC, Table Per concrete Class, 179
TPH, Table Per Hierarchy, 178
TPT, Table Per Type, 178
transakcje w EF, 174
tryb Inspect Mode, 271
Twitter Bootstrap, 329, 473
tworzenie
 baz danych, 419, 420
 interfejsu dla repozytorium, 337
 klasy kontekstu, 290
 konta FTP, 421
 migracji początkowej, 302
 modelu danych, 283
 nowego projektu, 279
 nowego projektu Repozytorium, 296
 obiektu XMLHttpRequest, 493
 paczek skryptów, 209
 profilu, 232
 referencji, 297
typ, 26
 ActionResult, 260
 HttpResponseMessage, 259
 void, 259
 wyliczeniowy, 31
 z aplikacji, 260
typy
 anonimowe, 105
 danych MIME, 444
 dopuszczające wartości zerowe, 33
 dynamiczne, 107
 pół formularza, 458
 rezultatu, 194
 walidacji, 449
 wyjątków, 240
tytuł strony, 333

U

uchwyty HTTP, 193
UoW, 410
uruchamianie
 aplikacji, 316
 CORS, 256
 pierwszej migracji, 304
 Web Deploy, 419
uruchomiona aplikacja, 295
ustawienie
 atrybutu, 463
 projektu startowego, 297
usuwanie
 kaskadowe, 177
 węzłów, 478
uwierzytelnianie, 241, 369, 370
 przez Windows ACS, 249
 za pomocą claimów, 245

V

view, 128, 130
ViewBag, 204
ViewData, 204
ViewModel, 218, 402
Virtual Machine, 144
Visual Studio 2013, 269
 Browser Link, 270
 GIT, 272
 JavaScript, 271
 JSON Editor, 271
 Microsoft Azure, 272
 pasik przewijania, 270
 podgląd definicji, 270
 skróty klawiszowe, 272
 tryb Inspect Mode, 271

W

walidacja, 221, 449
 strony, 466
 żądania POST, 357
walidator W3C, 468
warstwa modelu, 258, 294, 311
wartości
 zerowe, 33
 zmiennych, 325
WCF, Windows Communication Foundation, 140
WCF Endpoint, 142
wcięcia, 22
wczytywanie
 leniwe, 169
 zachłanne, 168

wdrażanie aplikacji, 423

Web API, 251

ASP.NET MVC, 252

CORS, 256

Entity Framework, 258

pobieranie danych, 261

routing, 257

typy, 258

wersjonowanie, 262

Web API 2, 251

Web Deploy, 420

Web Role, 144

Web serwis, 251, 253

Web Site, 144

Web Storage, 461

WebSockets, 462

wersje

języka C#, 20

Windows Server 2012, 267

wersjonowanie w Web API, 262

weryfikacja położenia, 467

widok, 200, 390, 400, 404

Details, 344

dla konta Admin, 375

dla konta Pracownik, 375

dla niezalogowanych użytkowników, 376

dla zalogowanych użytkowników, 375

Index, 398

Partial, 368, 402

Partial View, 229, 386

z layoutem, 229

widoki

częściowe, 201, 366

typowane, 205

WIF, 245

Windows ACS, 248

Windows Azure Storage, 143

Windows Server 2012, 267

właściwości, 52

właściwości XMLHttpRequest, 494

włączenie NuGet, 281

Worker Role, 144

WPF, Windows Presentation Foundation, 139

wstrzykiwanie

implementacji, 397

kontekstu, 340

repozytorium, 336

wybieranie

klasy z modelem, 313

miejscza publikacji, 423

szablonu, 296

treści modelu, 411

typu aplikacji, 280

wygląd

aplikacji, 330–332

aplikacji z paginacją, 385

strony, 399

wyjątki, 77, 238

wykonanie

natychmiastowe, 171

odroczone, 171

wyrażenia

dla metod, 122

dla właściwości, 122

lambda, 74

regularne, 89

wyświetlanie ogłoszeń, 397

wywoływanie metod, 51

wzorce

architektoniczne, 125, 277

projektowe, 277

wzorzec

IoC, 277

MVC, 185

UnitOfWork, 278

X

XMLHttpRequest, 493

Z

zabezpieczanie akcji, 373

zagnieździanie przestrzeni nazw, 79

zainstalowane pakiety, 299

zakup certyfikatu SSL, 450

zapytania

bezpośrednie, 173

LINQ, 150

zarządzanie

kontami FTP, 421

wyjątkami

globalne, 239

lokalne, 238

zasada

Controller, 433

Creator, 433

DRY, 435

High Cohesion, 434

Indirection, 435

Information Expert, 433

KISS, 436

Low Coupling, 433

odwrócenia zależności, 431

otwarte-zamknięte, 428

podstawienia Liskov, 428

- pojedynczej odpowiedzialności, 427
Polymorphism, 434
Protected Variations, 435
Pure Fabrication, 434
separacji interfejsów, 429
YAGNI, 437
zasoby niezarządzane, 70
zastosowanie
 atrybutu Serializable, 94
 ViewModel, 402
zbior zasad
 GRASP, 432
 SOLID, 427
zdarzenia, 75, 464, 487
zdobywanie
 linków, 502
 sztucznych linków, 509
zgłaszczenie wyjątków, 77
zmienne
 domniemane, 105
 tylko do odczytu, 29
znaczniki HTML, 454, 459, 500
zwalnianie zasobów niezarządzanych, 70

ż

- żądania
 AJAX, 491
 HTTP, 445
żądanie
 do aplikacji ASP.NET, 189
 GET, 196, 445
 POST, 196, 446

PROGRAM PARTNERSKI

GRUPY WYDAWNICZEJ HELION



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

Zmień swoją stronę WWW
w działający bankomat!

Dowiedz się więcej i dołącz już dzisiaj!

<http://program-partnerski.helion.pl>

GRUPA WYDAWNICZA
Helion SA