Paweł Dróżdż

PROGRAMUJ Z .NET

Praktyka ponad teorią

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 Helion SA 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 Helion SA nie ponoszą również żadnej odpowiedzialności za ewentualne szkody wynikłe z wykorzystania informacji zawartych w książce.

Redaktor prowadzący: Małgorzata Kulik

Projekt okładki: Studio Gravite / Olsztyn
Obarek, Pokoński, Pazdrijowski, Zaprucki
Grafika na okładce została wykorzystana za zgodą Shutterstock.com

Helion SA
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/netpra
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Kody źródłowe wybranych przykładów dostępne są pod adresem:
ftp://ftp.helion.pl/przyklady/netpra.zip

ISBN: 978-83-283-5644-3

Copyright © Helion 2020

Printed in Poland.

• Kup książkę
• Poleć książkę
• Oceń książkę

• Księgarnia internetowa
• Lubię to! » Nasza społeczność

Spis treści

Wstęp 7

Rozdział 1. Serwis usprawniający proces produkcyjny 9

Załóżmy, że… 9
Dokumentacja 9
Mapa możliwości tego projektu 10
Trochę teorii dotyczącej projektu 11
Konfiguracja środowiska do pracy 12
Plan bazy danych 12
Implementacja bazy danych 13
Koncepcja oraz implementacja interfejsów dla serwisów 17
Implementacja serwisów 19
Implementacja kontrolerów 30
Wstępne testy przy użyciu narzędzi dedykowanych 36
Podsumowanie projektu 38

Rozdział 2. Manager rachunków 39

Załóżmy, że… 39
Dokumentacja 39
Mapa możliwości tego projektu 42
Część backendowa 42
- Trochę teorii dotyczącej projektu 42
- Konfiguracja środowiska do pracy 44
- Plan bazy danych 44
- Implementacja bazy danych 45
- Koncepcja oraz implementacja interfejsów 51
- Implementacja serwisów 55
- Implementacja kontrolerów 61
- Wstępne testy przy użyciu narzędzi dedykowanych 68
- Podsumowanie projektu 69

Kup ksi k   Pole ksi k

Część frontendowa 69
Wstęp 69
Omówienie koncepcji projektu 70
Konfiguracja 70
Logowanie 70
Podsumowanie 99

Rozdział 3. Manager rachunków w wersji desktopowej 101

Załóżmy, że… 101
Dokumentacja 101
Mapa możliwości tego projektu 102
Trochę teorii dotyczącej projektu 102
Plan bazy danych 103
Implementacja bazy danych 103
Serwis do bazy danych 105
Podpięcie serwisów do widoków 110
Podsumowanie projektu 129

Rozdział 4. Serwis usprawniający managera rachunków 131

Załóżmy, że… 131
Dokumentacja 131
Mapa możliwości tego projektu 132
Burza mózgów 132
Trochę teorii dotyczącej projektu 133
Nasz wizja WCF 133
Tworzenie nowego projektu WCF 134
Podłączenie do istniejącego serwisu 135
Podsumowanie projektu 137

Rozdział 5. Asystent managera sklepu 139

Załóżmy, że… 139
Dokumentacja 139
Mapa możliwości tego projektu 140
Trochę teorii dotyczącej projektu 141
SCRUM 141
Planowanie backlogu 142
Tworzenie backlogu 142
Sprint 1. 145
Sprint 2. 152
Sprint 3. 160

Kup ksi k Pole ksi k

Sprint 4. 165
Sprint 5. 170
Testowanie 174
Podsumowanie projektu 175

Rozdział 6. Asystent managera sklepu w wersji desktopowej 177

Załóżmy, że… 177
Dokumentacja 177
Mapa możliwości tego projektu 178
Trochę teorii dotyczącej projektu 179
Tworzenie backlogu 179
Sprint 1. 182
Sprint 2. 190
Sprint 3. 197
Sprint 4. 199
Sprint 5. 204
Testowanie 208
Podsumowanie projektu 211

Rozdział 7. Serwis usprawniający asystenta managera sklepu 213

Załóżmy, że… 213
DOKUMENTACJA 213
Mapa możliwości tego projektu 214
Trochę teorii dotyczącej projektu 214
Tworzenie backlogu 215
Podsumowanie projektu 223

Rozdział 8. Elektroniczny podpis przy użyciu technologii blockchain 225

Załóżmy, że… 225
Dokumentacja 226
Mapa możliwości tego projektu 226
Trochę teorii dotyczącej projektu 227
Tworzenie tablicy canbanowej 228
Rozpoczynamy pracę 228
1. Projekt bazy danych 228
2. Skrypt do bazy danych 229
3. Konfiguracja projektu 229
4. Migracja bazy danych 229
5. Przygotowanie oraz skompilowanie projektu w języku Solidity 230

Kup ksi k Pole ksi k

6. Serwis do obsługi użytkowników ... 231
7. Kontroler do obsługi użytkowników ... 233
8. Serwis do obsługi BC ... 233
9. Kontroler do obsługi BC ... 240
10. Swagger ... 240
Testy manualne ... 242
Podsumowanie projektu ... 242

Rozdział 9. Chmurowy serwis magazynujący i agregujący dane ... 243

Załóżmy, że… ... 243
Dokumentacja ... 244
Mapa możliwości tego projektu ... 244
Trochę teorii dotyczącej projektu ... 245
Tworzenie tablicy canbanowej ... 246
Rozpoczynamy pracę ... 246
1. Stworzenie modeli bazy danych ... 246
2. Konfiguracja serwera SQL w chmurze i migracja modeli do bazy chmurowej ... 248
3. Stworzenie API do obsługiwania użytkowników ... 252
4. Konfiguracja bazy BLOB oraz dopisanie serwisu umieszczającego pliki w chmurze ... 256
5. Konfiguracja NRD w chmurze Azure i dopisanie serwisu umieszczającego w bazie informacje o dodaniu plików ... 261
6. Funkcja do generowania raportu ... 265
7. WebJob do sprawdzania, czy pracownik umieścił jakiś plik na serwerze ... 268
8. Testowanie ... 270
Podsumowanie projektu ... 271

Rozdział 10. Serwis do zarządzania plikami w wersji AWS ... 273

Załóżmy, że… ... 273
Dokumentacja ... 273
Mapa możliwości tego projektu ... 274
Trochę teorii dotyczącej projektu ... 274
Tworzenie tablicy canbanowej ... 275
Rozpoczynamy pracę ... 275
1. Przyjrzenie się pracy z bazami danych w AWS ... 275
2. Dostosowanie API do AWS-owych standardów ... 276
3. Wersja beta kodu zastępującego azurową funkcję przez odpowiedniki z AWS ... 277
4. Wersja beta kodu zastępującego zadanie WebJob przez odpowiedniki z AWS ... 278
Podsumowanie projektu ... 279

Kup ksi k Pole ksi k

Rozdział 1.

Serwis usprawniający proces produkcyjny

Załóżmy, że…

Jesteśmy stosunkowo małą, zatrudniającą około 50 pracowników firmą, specjalizującą się tylko w obszarze .NET oraz wszystkim tym, co z nim związane. Jesteśmy od niedawna na rynku i nasze stawki nie są wygórowane, gdyż dopiero zaczynamy budować markę oraz renomę. W piątkowy jesienny poranek przychodzi do nas 2 przedstawicieli potencjalnego klienta z konkretnym zleceniem. Reprezentują oni firmę produkcyjną, mającą kilka swoich fabryk na całym świecie. Specjalizują się w wytwarzaniu urządzeń elektronicznych, mają swój dział IT, ale obecne nie wyrabiają się z pracami programistycznymi i chcieliby nam zlecić napisanie serwisu dla jednego ze swoich priorytetowych projektów. Posiadają już szkielet interfejsu, brakujące jego elementy mają powstać ASAP — tak szybko, jak to tylko możliwe — a naszym zadaniem ma być implementacja bazy danych, zrobienie serwisu oraz wystawienie API. Po wyrażeniu wstępnego zainteresowania projektem i wypiciu kawy, wchodzimy w większe szczegóły. Serwis ma służyć do wyliczania kosztów produkcji urządzeń elektrycznych według ściśle określonych kryteriów. Osoba obsługująca aplikację podaje kilka parametrów urządzenia, a serwis na podstawie odpowiednich wzorów oraz stałych wylicza koszt produkcji i zwraca go interesantowi. Naszemu klientowi zależy na czasie i niezawodności, pieniądze grają rolę drugorzędną. Przy takim podejściu klienta szybko dochodzimy do porozumienia i zabieramy się do pracy, zaczynając ją od analizy dokumentów, które otrzymaliśmy. Dokumenty te zawierają szczegółowy opis obliczania kosztów produkcji urządzeń.

Dokumentacja

Urządzenia składają się z 1, 2, 3 lub 4 modułów, w zależności od typu urządzenia. Każdy moduł ma z góry określoną cenę, dodatkowo w przypadku każdego z góry określono, ile godzin trzeba poświęcić na jego montaż. Składanie urządzeń odbywa się w 5 miastach w Polsce, a stawki za godzinę pracy pracownika różnią się w zależności od lokalizacji.

Kup ksi k

Pole ksi k

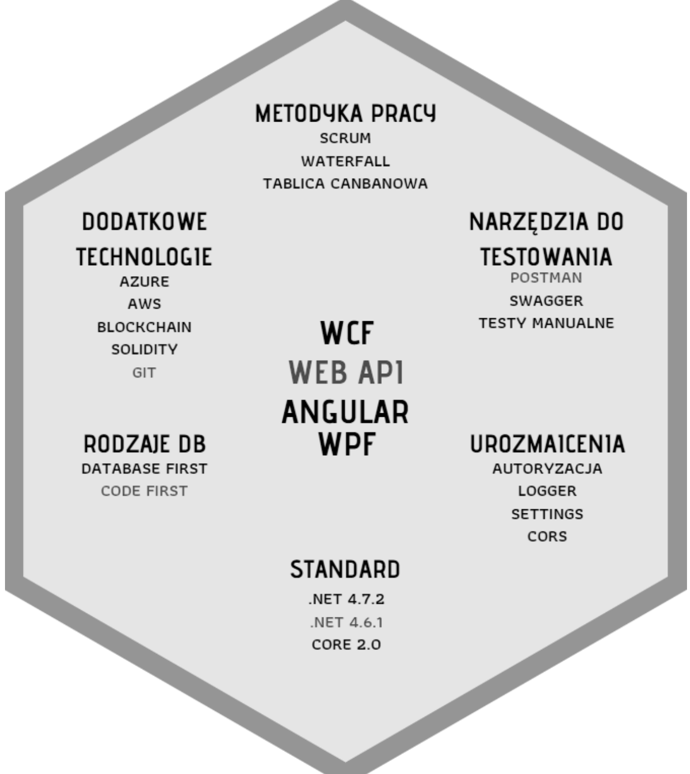
Koszt produkcji urządzenia jest ponadto podnoszony przez koszt transportu modułów z fabryki w Chinach, różny w zależności od miasta docelowego. Ostatnim czynnikiem wpływającym na cenę są koszty dodatkowe, które stanowią 30% wszystkich wcześniej wymienionych kosztów. Przy modułach powinny być przechowywane informacje o wadze elementu oraz krótki opis. Jednostki dotyczące cen oraz wagi zawsze są takie same i są wyrażane — odpowiednio — w dolarach oraz kilogramach. Podsumowując, możemy pokusić się o taki wzór:

> **koszt** = (cena transportu + zsumowana cena modułów + zsumowana liczba godzin na montaż modułów * koszt pracy za 1 godzinę dla miasta) * 130%

Jako użytkownicy API chcemy podawać niezbędne parametry i otrzymywać koszt produkcji. Ponadto chcemy mieć możliwość pobierania, dodawania, edytowania, usuwania miasta, a także — dodatkowo — edycji kosztów transportu oraz stawki za godzinę pracy dla poszczególnego miasta. Chcemy również pobierać, dodawać, usuwać i edytować informacje o modułach. Życzylibyśmy sobie też, żeby po każdym zapytaniu o koszt było ono gdzieś przechowywane, dzięki czemu można będzie przy kolejnym wyszukaniu dostać informację, że urządzenie o takiej specyfikacji było już wyszukiwane.

Mapa możliwości tego projektu

RYSUNEK 1.1.
Mapa rozdziału



Trochę teorii dotyczącej projektu

Istnieje wiele rodzajów baz danych oraz wiele sposobów ich tworzenia w .NET. We wszystkich programach opisywanych w tej książce będziemy używać SQL, jednak, żeby tworzenie bazy danych przy ósmym z kolei projekcie nie było zbyt nudne i proste, będziemy używać różnych sposobów tworzenia bazy. Wykorzystując EntityFramework, mamy do dyspozycji 3 sposoby:

- Code First,
- Model First,
- Database First.

Zdania co do tego, który jest najlepszy, są podzielone. To, z którego warto skorzystać, zależy w bardzo dużym stopniu od wielkości bazy danych, jaką będziemy tworzyć. W pierwszym projekcie użyjemy sposobu Code First. Więcej o nim powiemy sobie, gdy przejdziemy do projektowania bazy danych.

Istnieje co najmniej kilka wzorców projektowych dotyczących warstwy dostępu do danych. Na początek rozpoczniemy od standardowej warstwy dostępu do danych, korzystającej z klasy `DbContext`, nie będziemy też używać żadnego wzorca projektowego. Cała struktura projektu będzie miała dwie warstwy: prezentacji oraz logiki biznesowej. Czasami stosuje się trzecią warstwę, zwaną warstwą dostępu do danych, i w niej implementuje się mechanizm łączenia aplikacji z bazą danych. Zważywszy na fakt, iż projekt będzie stosunkowo mały, zrezygnujemy z tej warstwy i będziemy się łączyć z bazą danych na poziomie serwisu.

Kiedy już uda nam się zaimplementować wszystkie niezbędne funkcjonalności, przejdziemy do testowania naszego API. W większych projektach testowanie odbywa się równocześnie z implementacją kolejnych funkcjonalności, jednak dla uproszenia w naszym małym projekcie zrobimy to na końcu.

Do tematu testowania możemy podejść na co najmniej dwa sposoby. Jednym jest napisanie testów, a drugim użycie jednego z wielu narzędzi do szybkiego testowania API. Wszystkie te narzędzia działają na podobnej zasadzie. Użytkownik wywołuje konkretny endpoint i po chwili dostaje odpowiedź, którą może zweryfikować pod kątem tego, czy jest zgodna z oczekiwanym rezultatem.

Najbardziej popularne narzędzia to:

- Postman,
- Swagger,
- Fiddle.

Kup ksi k Pole ksi k

W tym projekcie użyjemy Postmana, a więcej szczegółów dotyczących jego użycia podamy podczas testowania naszego projektu.

W każdym projekcie będziemy starali się omówić jakiś dodatkowy element związany z pracą dewelopera w zespole. Przy tworzeniu tego serwisu postaram się ponadto przybliżyć pojęcie systemu kontroli wersji, a w szczególności zapoznać Czytelnika z Gitem, czyli obecnie najpopularniejszym narzędziem do wersjonowania plików. Git służy do porządkowania kolejnych wersji programu oraz organizowania pracy w zespole nad jednym projektem, w tym samym lub różnym czasie.

Konfiguracja środowiska do pracy

Przed rozpoczęciem pracy musimy zawsze skonfigurować środowisko. Głównie będą to czynności związane z konfiguracją Gita. We wszystkich projektach będziemy korzystali z tego darmowego narzędzia Microsoftu, służącego do zarządzania kodem projektu, jak i samym projektem. Zakładając, że każdy Czytelnik zna już podstawy Gita, nie będziemy opisywać tutaj procesu jego pobierania z oficjalnej strony i instalowania. Po założeniu konta na stronie internetowej (*https://identity.getpostman.com/signup?continue=https%3 ↪A%2F%2Fapp.getpostman.com%2F%3F*) tworzymy projekt i kopiujemy jego adres URL, po czym udajemy się do miejsca na komputerze, gdzie chcemy przechowywać nasz projekt, i w konsoli Gita wpisujemy te polecenia:

LISTING 1.1. Komendy inicjalizujące i kopiujące repozytorium

```
git init
git clone <skopiowany URL>
```

Zakończyliśmy konfigurację, w tym miejscu na dysku w dalszych krokach utworzymy projekt.

Plan bazy danych

Każdą aplikację będziemy rozpoczynać od projektowania bazy danych. Ważne jest, żeby dokładnie przemyśleć każdy punkt wymagań klienta, gdyż im lepiej zaprojektujemy teraz bazę, tym więcej czasu zaoszczędzimy później, ograniczając liczbę ewentualnych poprawek. Jednak trzeba pamiętać, że normą jest, iż w nawet najlepszych zespołach deweloperskich zdarzają się poprawki, gdyż projektanci w fazie projektowania nie przewidzieli pewnych konfliktów modeli czy też optymalizacji. Nie jest sztuką zaprojektować bazę dla projektu, ale prawdziwym wyzwaniem jest sprawić, by ta baza była optymalna, dlatego coraz częściej można spotkać specjalistów zajmujących się tylko projektowaniem baz danych.

Kup ksi k Pole ksi k

Nasza struktura bazy będzie prosta; zaczynamy od tabeli `City`, która będzie przetrzymywała informacje o nazwie miejscowości, koszcie transportu do niej i cenie za godzinę pracy. Kolejnym oczywistym wyborem jest tabela `Module`, w której dla każdego modułu będziemy przechowywać jego kod, nazwę, wagę oraz krótki opis. Pozostaje nam kwestia przechowywania danych historycznych, które umieścimy w 3. tabelce, o nazwie `SearchHistory`; będziemy tam trzymać informacje o mieście i modułach, których dotyczyło zapytanie. Nie będziemy jednak duplikować informacji o miastach i modułach, które zapisane są już wszakże w poprzednio wspomnianych tabelach. W tej tabeli zapiszemy jedynie datę wyszukiwania, natomiast do miast i modułów będziemy się odwoływać poprzez pole `Id` z odpowiadającej im tabeli. Pole `Id` będzie posiadać każda tabela w naszej bazie.

Implementacja bazy danych

Pora przejść do tego, co tygryski lubią najbardziej, czyli kodowania, a dokładnie do tworzenia modeli obiektów. Następnie dzięki frameworkowi EntityFramework dokonamy migracji, która spowoduje wygenerowanie bazy danych z odpowiednimi tabelami.

Na początku tworzymy nowy projekt ASP.NET WEB Application, oparty na frameworku .NET 4.6.1. Jako typ projektu wybieramy MVC, pozostałe parametry pozostawiamy domyślne. Po utworzeniu projektu, wykonujemy nasz pierwszy commit z dziewiczą wersją projektu, używając poleceń:

LISTING 1.2. Komendy tworzące commit i wypychające go do repozytorium

```
git add .
git commit -m  "Initialize commit"
git push master
```

Następnie w folderze *Models* tworzymy trzy klasy: `City`, `Module` oraz `SearchHistory`. Implementujemy je zgodnie z wcześniejszymi ustaleniami. Dodatkowo musimy zaimplementować klasę kontekstu, czyli klasę, która w ogólnym rozumieniu będzie naszym łącznikiem z bazą danych i modelami, które będą odzwierciedlać tabele. Przed omówieniem szczegółów tego pliku, musimy w NutGetPackage pobrać paczkę **EntityFramework**. Dobrą praktyką jest użycie w nazwie słowa *context*. Klasa ta powinna dziedziczyć po klasie `DbContext` z EntityFramework oraz posiadać konstruktor wywołujący konstruktor bazowy. Dodatkowo powinniśmy użyć DbSet-ów, które są bardzo ważne w kontekście całej aplikacji i mają za zadanie mapować stworzone modele na obiekty, do których będziemy się odwoływać używając obiektów kontekstu. Klasa kontekstu powinna wyglądać następująco:

Kup ksi k Pole ksi k

Listing 1.3. Kod konfigurujący modele bazodanowe

```
public class CalculatorContext : DbContext
    {
        public CalculatorContext()
            : base("name=ConnectionString") { }

        public DbSet<SearchHistory> SearchHistory { get; set; }
        public DbSet<Module> Module { get; set; }
        public DbSet<City> City { get; set; }
    }
```

Stworzone modele powinny wyglądać tak jak poniżej. Jeśli są zgodne z poniższym kodem, to możemy wykonać commit.

Listing 1.4. Kod modeli bazodanowych

```
public class City
{
    public int Id { get; set; }
    public string Name { get; set; }
    public double TransportCost { get; set; }
    public double CostOfWorkingHour { get; set; }
    public virtual SearchHistory SearchHistory { get; set; }
}

public class Module
{
    public int Id { get; set; }
    public string Code { get; set; }
    public string Name { get; set; }
    public double Price { get; set; }
    public double AssemblyTime { get; set; }
    public double Weight { get; set; }
    public string Description { get; set; }
    public virtual SearchHistory SearchHistory { get; set; }
}

public class SearchHistory
{
    public int Id { get; set; }
    public int CityId { get; set; }
    public string ModuleName1 { get; set; }
    public string ModuleName2 { get; set; }
    public string ModuleName3 { get; set; }
    public string ModuleName4 { get; set; }
    public double ProductionCost { get; set; }
}
```

Kup ksi k Pole ksi k

LISTING 1.5. Komendy tworzące commit

```
git add .
git commit -m "Initialize commit"
git push master
```

Kiedy już stworzyliśmy modele, pora przejść do pierwszej migracji bazy danych. W tym celu powinniśmy dostarczyć do EF (EntityFramework) informacje o bazie danych; robimy to w pliku *Web.config*, dodając sekcję <connectionStrings></connectionStrings> w tagach <configuration></configuration> i ustawiając w niej między innymi adres serwera lokalnego na adres SQL Server oraz nazwę bazy danych, np. w taki sposób: <add name="ConnectionString" providerName="System.Data.SqlClient" connectionStr ↪ing="Server=.;Database=nazwa_bazy_danych;Trusted_Connection=True;MultipleAct ↪iveResultSets=true" />.

Następnie przechodzimy do narzędzia *package manager console*. Musimy zacząć od komendy: enable-migration, która powoduje odblokowanie migracji. W kolejnym kroku dodajemy migrację komendą: add-migration „nazwa-migracji", dzięki której wygeneruje się plik migracji, a w nim wszystkie komendy niezbędne do zbudowania 3 pustych tabel. Pozostaje uruchomienie powstałego w kroku poprzednim pliku migracji komendą update-database i sprawdzenie, czy wszystko wygenerowało się poprawnie.

LISTING 1.6. Komendy tworzące commit

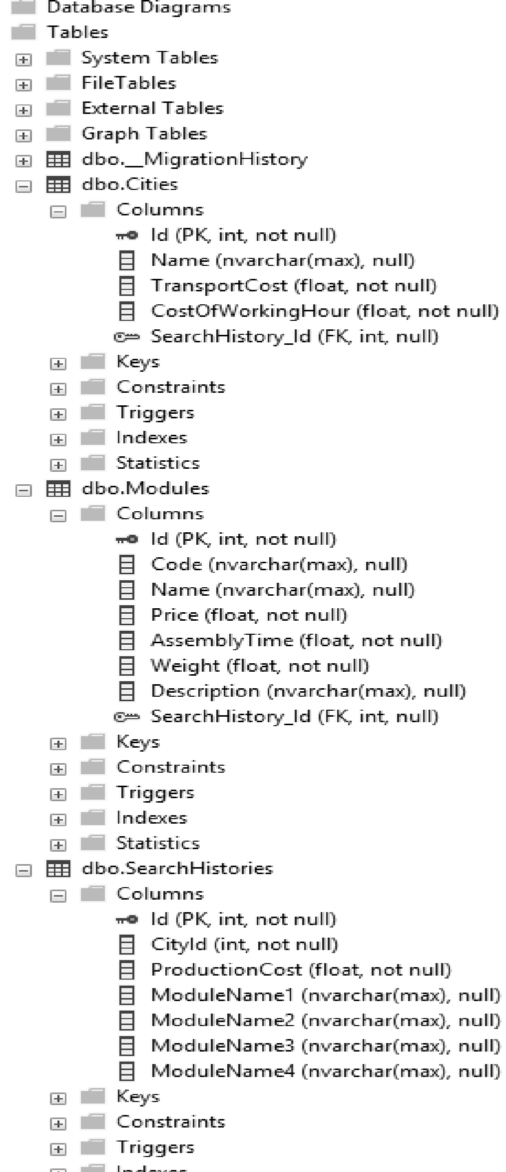
```
enable-migrations
add-migration "Nazwa migracji"
update-database
```

Uruchamiamy program SSMS lub podglądamy naszą świeżo powstałą bazę w oknie *SQL Server Object Explorer* w Visual Studio, które służy właśnie do pracy z bazami danych. Jeśli wszystko w tabelach się zgadza i wygląda tak jak na rysunku na następnej stronie, to możemy przejść dalej i zrobić commit z naszymi zmianami.

LISTING 1.7. Komendy tworzące commit

```
git add .
git commit -m "Added first migration. Created models."
```

Kup ksi k Pole ksi k



RYSUNEK 1.2. *Tabele utworzone w SQL-u po wykonaniu migracji*

Koncepcja oraz implementacja interfejsów dla serwisów

Kiedy już mamy bazę danych naszej aplikacji, czas przejść do zaplanowania architektury API. To jest ten moment podczas pracy nad projektem, kiedy palce same rwą się do pisania programu, bo przecież wszystko jest jasne i proste, i wiadomo co trzeba zrobić; musimy jednak zwolnić i dokładnie przemyśleć, czy na pewno to, co chcemy pisać, będzie w 100% spełniało założenia projektowe. Jest kilka sposobów planowania architektury; my zastosujemy jeden z najprostszych i po prostu wypiszemy sobie, jakie interfejsy chcemy implementować, dzięki czemu dalsza praca będzie dużo przyjemniejsza, a podczas listowania interfejsów sprawdzimy, czy wszystko się zgadza z wymaganiami klienta. Ruszmy więc głową i zacznijmy planowanie.

Zanim rozpoczniemy prace nad interfejsami, najpierw stworzymy modele DTO, pomocne w pracy z kodem. Musimy je opracować w tym miejscu, ponieważ niektóre z metod omawianych w interfejsie będą zwracać właśnie te modele. Aby posiadać informacje, że operacja zakończyła się powodzeniem lub niepowodzeniem, będziemy zwracali obiekt informujący o rezultacie metody. W tym celu zaimplementujemy 3 dodatkowe klasy, które umieścimy w nowo stworzonym katalogu *ModelsDTO*. Klasy `OperationSuccessDTO` oraz `OperationErrorDTO` będą dziedziczyły po klasie `OperationResultDTO` i w zależności od potrzeby będą zwracać odpowiedni obiekt. Klasa `OperationErrorDTO` będzie zawierała tylko `message` z informacją o błędzie, natomiast klasa `OperationSuccessDTO` będzie generyczna, tzn. będzie zawierać taki obiekt, jakiego będziemy potrzebowali i który będziemy zwracali. Klasa, po której będziemy dziedziczyć, będzie zawierała tylko pole `Code`, informujące o statusie wykonanej operacji. Wszystko to po to, aby użytkownik API otrzymał jak na tacy rezultat operacji, którą wykonał. Jest to dość prosty mechanizm, który w dużym stopniu ułatwia użytkownikowi obsługę aplikacji, podobnie jak programiście jej testowanie. W parametrach metody na pewno będziemy otrzymywali listę modułów, którą również opakujemy w obiekt `DTO`. Oto implementacja omawianych powyżej klas:

LISTING 1.8. Kod modeli pomocniczych

```
public class OperationErrorDTO : OperationResultDTO
{
    public int Code { get; set; }
}
public class OperationResultDTO
{
    public string Message { get; set; }
}
public class OperationSuccesDTO<T> : OperationResultDTO
    where T : class
{
    public T Result { get; set; }
}
```

Kup ksi k Pole ksi k

```
public class ModuleListDTO
{
    public List<String> ModuleList { get; set; }
}
```

Zaczynamy od rzeczy najprostszych; będziemy implementować metody wpisujące się w pełni lub częściowo we wzorzec CRUD (ang. *create*, *read*, *update*, *delete*). W ten wzorzec możemy wpisać nasz pierwszy interfejs `IModuleService`. Podobnie będzie wyglądał interfejs dla miasta, tylko wzbogacimy go o dwie dodatkowe metody do edycji kosztów transportu i ceny za godzinę pracy w danym mieście. Interfejs dla serwisu odpowiedzialnego za operacje na tabeli `SearchHistory` będzie zawierał 3 metody: pobierającą całą tabelę, pobierającą rekordy po nazwie miasta i modułach oraz dodającą rekord do bazy danych. Pozostaje nam do zaimplementowania serwis do obliczania kosztów. Podzielimy go na dwa mniejsze, ale dzięki temu w pełni niezależne serwisy. Pierwszy będzie implementował tylko jedną metodę obliczającą koszty, a drugi będzie miał za zadanie zwrócić obliczony koszt w zależności od podanych modułów i miasta, czyli albo z tabeli `SearchHistoty` (jeśli zapytanie się powtarza), albo korzystając z serwisu do obliczania kosztów. Zatem będziemy potrzebowali modelu pomocniczego również do zwrócenia obliczonych kosztów. Będzie on zawierał obliczony koszt oraz informację, czy w przeszłości było już zapytanie o taką specyfikację.

LISTING 1.9. Model pomocniczy

```
public class ResultCostDTO
{
    public double Cost { get; set; }
    public bool InSearchHistory { get; set; }
}
```

Powinniśmy stworzyć katalog *services*, a w nim katalog *interfaces*, i w nim umieszczać omówione pliki warstwy abstrakcyjnej. Interfejsy powinny wyglądać tak jak na listingu poniżej; standardowo, jeśli masz taki sam kod, to możesz wykonać kolejny commit.

LISTING 1.10. Kod interfejsów dla serwisów

```
public interface ICalculatorCostService
{
    OperationResultDTO CalculateCost(string cityName, ModuleListDTO
    ↪moduleListDTO);
}

public interface ICityService
{
    City GetCityByName(string cityName);
    OperationSuccesDTO<IList<City>> GetCities();
    OperationResultDTO UpdateCostOfWorkingHour(string cityName, double
    ↪workingHourCost);
```

Kup ksi k Pole ksi k

```
        OperationResultDTO UpdateTransportCost(string cityName, double transportCost);
        OperationResultDTO AddCity(City city);
        OperationResultDTO DeleteCity(string cityName);
    }

    public interface IModuleService
    {
        Module GetModuleByName(string moduleName);
        OperationSuccesDTO<List<Module>> GetModules();
        OperationSuccesDTO<Module> AddModule(Module module);
        OperationSuccesDTO<Module> UpdateModule(Module module);
        OperationSuccesDTO<Module> DeleteModule(string name);
    }

    public interface ISearchHistoryService
    {
        ResultCostDTO GetSearchHistory(string cityName, ModuleListDTO moduleListDTO);
        OperationSuccesDTO<IList<SearchHistory>> GetSearchHistories();
        OperationResultDTO AddSearchHistory(SearchHistory searchHistory);
    }

    public interface IShowResultService
    {
        ResultCostDTO PresentResult(string cityName, ModuleListDTO moduleListDTO);
    }
```

LISTING 1.11. Komendy tworzące commit

```
git add .
git commit -m  "Added services interfaces"
```

Implementacja serwisów

Opierając się na zdefiniowanych powyżej interfejsach, przechodzimy do zaprogramowania logiki biznesowej. W serwisach będziemy odwoływali się do bazy danych przy użyciu kontekstu; jest to możliwe dzięki EntityFrameworkowi. Kontekst, a dokładnie interfejs `ICalclculatorContext`, będzie wstrzykiwany do każdego konstruktora w serwisie za pomocą mechanizmu *dependency injectora*. Za pomocą kontekstu możemy wykonywać operacje na bazie danych, między innymi dodawać, edytować, pobierać i usuwać dane z tabeli. W naszym programie te metody wystarczą do zaimplementowania około 80% metod zawartych w serwisach.

Skupmy się teraz na implementacji serwisów, które będziemy umieszczać w utworzonym przez nas folderze *Implemenatations*, mieszczącym się w folderze *Services*. Zdecydowana większość metod będzie używała gotowych metod kontekstu, a nasze działania będą sprowadzały się tylko do sprawdzenia, czy przekazywany obiekt nie jest nullem, wywołania odpowiedniej metody z kontekstu (`Add`, `Update`, `Delete`) i ewentualne dodatkowego

wywołania metody SaveChanges do zapisania wprowadzonych przez nas zmian w bazie danych. Idealnym przykładem jest serwis ModuleService, w którym są użyte podręcznikowo metody CRUD; prawidłowy kod powinien wyglądać jak poniżej:

LISTING 1.12. Kod implementacji serwisów ModuleService oraz CityService

```
public class ModuleService : IModuleService
  {
      private readonly CalculatorContext context;

      public ModuleService(CalculatorContext context)
      {
          this.context = context;
      }

      public OperationSuccesDTO<Module> AddModule(Module module)
      {
          context.Module.Add(module);
          context.SaveChanges();
          return new OperationSuccesDTO<Module> { Message = "Success" };
      }

      public Module GetModuleByName(string moduleName)
      {
          return context.Module.Where(module => module.Name ==
          ↪moduleName).FirstOrDefault();
      }

      public OperationSuccesDTO<List<Module>> GetModules()
      {
          List<Module> modules = context.Module.ToList();
          return new OperationSuccesDTO<List<Module>> { Message = "Success",
          ↪Result = modules };
      }

      public OperationSuccesDTO<Module> DeleteModule(string name)
      {
          var module = context.Module.Where(m => m.Name == name).FirstOrDefault();
          context.Module.Remove(module);
          context.SaveChanges();
          return new OperationSuccesDTO<Module> { Message = "Success" };
      }

      public OperationSuccesDTO<Module> UpdateModule(Module module)
      {
          var mod = context.Module.Where(m => m.Name ==
          ↪module.Name).FirstOrDefault();
          mod.Name = module.Name;
          mod.Price = module.Price;
          mod.Weight = module.Weight;
          mod.AssemblyTime = module.AssemblyTime;
          mod.Code = module.Code;
          mod.Description = module.Description;
          context.SaveChanges();
```

Kup książkę Poleć książkę

```
            return new OperationSuccesDTO<Module> { Message = "Success" };
        }
    }

public class CityService : ICityService
    {
        private readonly CalculatorContext context;

        public CityService(CalculatorContext context)
        {
            this.context = context;
        }

        public City GetCityByName(string cityName)
        {
            return context.City.Where(city => city.Name == cityName).FirstOrDefault();
        }

        public OperationResultDTO UpdateTransportCost(string cityName, double
        ➥transportCost)
        {
            var updateCity = context.City.Where(city => city.Name ==
            ➥cityName).FirstOrDefault();

            if (updateCity == null)
            {
                return new OperationErrorDTO { Code = 404, Message = $"City with name:
                ➥{cityName} doesn't exist" };
            }

            updateCity.TransportCost = transportCost;
            context.SaveChanges();
            return new OperationSuccesDTO<Module> { Message = "Success" };
        }

        public OperationResultDTO UpdateCostOfWorkingHour(string cityName, double
        ➥workingHourCost)
        {
            var updateCity = context.City.Where(city => city.Name ==
            ➥cityName).FirstOrDefault();

            if (updateCity == null)
            {
                return new OperationErrorDTO { Code = 404, Message = $"City with name:
                ➥{cityName} doesn't exist" };
            }

            updateCity.CostOfWorkingHour = workingHourCost;
            context.SaveChanges();
            return new OperationSuccesDTO<Module> { Message = "Success" };
        }

        public OperationSuccesDTO<IList<City>> GetCities()
        {
            List<City> cities = context.City.ToList();
```

```
            return new OperationSuccesDTO<IList<City>> { Message = "Success", Result =
            ↪cities };
        }

        public OperationResultDTO AddCity(City city)
        {
            context.City.Add(city);
            context.SaveChanges();

            return new OperationSuccesDTO<Module> { Message = "Success" };
        }

        public OperationResultDTO DeleteCity(string cityName)
        {
            var city = GetCityByName(cityName);

            if (city == null)
            {
                return new OperationErrorDTO { Code = 404, Message = $"City with name:
                ↪{cityName} doesn't exist" };
            }
            context.City.Remove(city);
            context.SaveChanges();

            return new OperationSuccesDTO<Module> { Message = "Success" };
        }
    }
```

Na szczęście nie wszystko jest takie proste i będziemy mogli choć na chwilę zmusić nasz mózg do troszkę większego wysiłku, ponieważ implementacja liczenia kosztów oraz sprawdzania, czy urządzenie o zadanych parametrach istnieje już w bazie, wymaga czegoś więcej niż użycia gotowych metod. Zważywszy na fakt, że jest to pierwszy program w tej książce, postaramy się zakodować rozwiązanie w sposób łatwy i czytelny.

Potrzebujemy serwisu, który zwróci nam koszt urządzenia. Za jego pomocą musimy sprawdzić, czy w bazie danych było już zapytanie o takie urządzenie; jeśli nie — wtedy będziemy musieli obliczyć koszt urządzenia i zapisać rezultat do bazy danych. Będziemy tego dokonywać w serwisie `ShowResultService`, jednak nie będziemy całego kodu pisali właśnie w nim; wykorzystamy natomiast serwisy `SearchHistoryService` oraz `CalculatorService`. Zacznijmy od omówienia `SearchHistoryService`, a dokładnie sprawdzenia za jego pomocą, czy urządzenie istnieje w bazie. W parametrach metody podajemy miasto oraz listę modułów, które musimy sprawdzić w bazie danych, aby ustalić, czy kiedykolwiek już było zapytanie o te parametry. W ramach optymalizacji posłużymy się obiektem `DTO`, a dokładnie stworzymy obiekt `ModuleListDTO`, który będzie zawierał listę modułów i będzie przekazywany w parametrze omawianej metody. Zaczynamy od miasta i sprawdzamy, czy odpowiedni wpis istnieje w bazie; jeśli tak, to pobieramy z tabeli `SearchHistory` wszystkie rekordy zawierające nazwę tego miasta w kolumnie `CityName`, następnie iterujemy po wszystkich otrzymanych w ostatnim kroku wynikach. Przy każdej uzyskanej w ten sposób pozycji będziemy iterować szukając przekazanych w parametrze nazw modułów;

dzięki temu kolejno sprawdzimy, czy wszystkie moduły są zgodne z pozycjami zawartymi w bazie danych. Niestety, liczba modułów może być różna i musimy dodać jeszcze jeden warunek sprawdzający, czy liczba modułów przekazanych w parametrze jest taka sama jak liczba modułów dla danego urządzenia w bazie i taka sama jak liczba modułów zwróconych w wyniku porównywania. Jeśli będzie zachodziła pełna zgodność, to możemy zwrócić informację, że urządzenie o takiej specyfikacji jest już w bazie, po czym możemy podać koszt tego urządzenia.

LISTING 1.13. Kod implementacji serwisu SearchHistoryService

```
public class SearchHistoryService : ISearchHistoryService
    {
        private readonly CalculatorContext context;
        private readonly IModuleService moduleService;
        private readonly ICityService cityService;

        public SearchHistoryService(
                CalculatorContext context,
                IModuleService moduleService,
                ICityService cityService)
        {
            this.context = context;
            this.moduleService = moduleService;
            this.cityService = cityService;
        }

        public OperationResultDTO AddSearchHistory(SearchHistory searchHistory)
        {
            context.SearchHistory.Add(searchHistory);
            context.SaveChanges();

            return new OperationSuccesDTO<Module> { Message = "Success" };
        }

        public ResultCostDTO GetSearchHistory(string cityName, ModuleListDTO
        ➥moduleListDTO)
        {
            var city = cityService.GetCityByName(cityName);

            if (city == null)
            {
                return new ResultCostDTO { InSearchHistory = false };
            }

            var searchHistoryList = context.SearchHistory.Where(sh => sh.CityId ==
            ➥city.Id);

            if (searchHistoryList == null)
            {
                return new ResultCostDTO { InSearchHistory = false };
            }
            int counterModule = 0;

            foreach (SearchHistory searchHistory in searchHistoryList)
```

```
            {
                counterModule = 0;

                foreach (string searchHistoryPar in moduleListDTO.ModuleList)
                {
                    if (searchHistory.ModuleName1 == searchHistoryPar ||
                         searchHistory.ModuleName2 == searchHistoryPar ||
                         searchHistory.ModuleName3 == searchHistoryPar ||
                         searchHistory.ModuleName4 == searchHistoryPar)
                    {
                        counterModule++;
                    }
                    else
                    {
                        break;
                    }
                }

                if (moduleListDTO.ModuleList.Count() == ModuleHasValue(search
                ➥History) && moduleListDTO.ModuleList.Count() == counterModule)
                {
                    return new ResultCostDTO { InSearchHistory = true, Cost =
                    ➥searchHistory.ProductionCost };
                }
            }

            return new ResultCostDTO { InSearchHistory = false };
        }

        OperationSuccesDTO<IList<SearchHistory>> ISearchHistoryService.
        ➥GetSearchHistories()
        {
            List<SearchHistory> searchHistories = context.SearchHistory.ToList();

            return new OperationSuccesDTO<IList<SearchHistory>> { Message =
            ➥"Success", Result = searchHistories };
        }

        private int ModuleHasValue(SearchHistory SearchHistory)
        {
            int counter = 0;

            if (!(SearchHistory.ModuleName1 == string.Empty))
                counter++;
            if (!(SearchHistory.ModuleName2 == string.Empty))
                counter++;
            if (!(SearchHistory.ModuleName3 == string.Empty))
                counter++;
            if (!(SearchHistory.ModuleName4 == string.Empty))
                counter++;

            return counter;
        }
    }
```

Kup ksi k Pole ksi k

Pozostała jeszcze jedna metoda do implementacji; jest ona analogiczna do metod zawartych w dwóch pierwszych serwisach. Cały serwis wygląda tak:

LISTING 1.14. Kod implementacji serwisu SearchHistoryService, część 2

```
public class SearchHistoryService : ISearchHistoryService
    {
        private readonly CalculatorContext context;
        private readonly IModuleService moduleService;
        private readonly ICityService cityService;

        public SearchHistoryService(
                CalculatorContext context,
                IModuleService moduleService,
                ICityService cityService)
        {
            this.context = context;
            this.moduleService = moduleService;
            this.cityService = cityService;
        }

        public OperationResultDTO AddSearchHistory(SearchHistory searchHistory)
        {
            context.SearchHistory.Add(searchHistory);
            context.SaveChanges();

            return new OperationSuccesDTO<Module> { Message = "Success" };
        }

        public ResultCostDTO GetSearchHistory(string cityName, ModuleListDTO
        ↪moduleListDTO)
        {
            var city = cityService.GetCityByName(cityName);

            if (city == null)
            {
                return new ResultCostDTO { InSearchHistory = false };
            }

            var searchHistoryList = context.SearchHistory.Where(sh => sh.CityId ==
            ↪city.Id);

            if (searchHistoryList == null)
            {
                return new ResultCostDTO { InSearchHistory = false };
            }
            int counterModule = 0;

            foreach (SearchHistory searchHistory in searchHistoryList)
            {
                counterModule = 0;

                foreach (string searchHistoryPar in moduleListDTO.ModuleList)
                {
                    if (searchHistory.ModuleName1 == searchHistoryPar ||
                         searchHistory.ModuleName2 == searchHistoryPar ||
```

```
                    searchHistory.ModuleName3 == searchHistoryPar ||
                    searchHistory.ModuleName4 == searchHistoryPar)
                {
                    counterModule++;
                }
                else
                {
                    break;
                }
            }
            if (moduleListDTO.ModuleList.Count() == ModuleHasValue(searchHistory)
            ↪&& moduleListDTO.ModuleList.Count() == counterModule)
            {
                return new ResultCostDTO { InSearchHistory = true, Cost =
                ↪searchHistory.ProductionCost };
            }
        }
        return new ResultCostDTO { InSearchHistory = false };
    }
    OperationSuccesDTO<IList<SearchHistory>> ISearchHistoryService.
    ↪GetSearchHistories()
    {
        List<SearchHistory> searchHistories = context.SearchHistory.ToList();

        return new OperationSuccesDTO<IList<SearchHistory>> { Message = "Success",
        ↪Result = searchHistories };
    }

    private int ModuleHasValue(SearchHistory SearchHistory)
    {
        int counter = 0;

        if (!(SearchHistory.ModuleName1 == string.Empty))
            counter++;
        if (!(SearchHistory.ModuleName2 == string.Empty))
            counter++;
        if (!(SearchHistory.ModuleName3 == string.Empty))
            counter++;
        if (!(SearchHistory.ModuleName4 == string.Empty))
            counter++;

        return counter;
    }
}
```

Serwis `CalculatorCostService` na szczęście tylko z nazwy brzmi groźnie, ponieważ zawiera on tylko jedną metodę, liczącą koszt urządzenia. W parametrach metody otrzymujemy nazwę miasta oraz znany już nam obiekt `ModuleListDTO`, zawierający listę nazw modułów. Do wyliczenia kosztów będziemy potrzebowali informacji z bazy danych odnośnie wszystkich obiektów, których nazwy zostały przekazane w parametrach. Zaczynamy od miasta, które pobieramy, używając wstrzykniętego do konstruktora serwisu `ICityService`, po czym sprawdzamy, czy krotka z nazwą miasta istnieje w bazie danych. Następnie iterujemy `ModuleListDTO` i przy każdej iteracji pobieramy obiekt `module` za

Kup książkę Poleć książkę

pomocą `ModuleService` z bazy danych, sprawdzając po każdym pobraniu, czy taki moduł istnieje. Mając moduł, stosujemy wzór, który mogliśmy wyprowadzić z przekazanej nam dokumentacji technicznej:

```
modulesCost = modulesCost + module.Price + (module.AssemblyTime *
city.CostOfWorkingHour);
```

Po zakończeniu iteracji, mnożymy wynik przez 1,3, w ten sposób spełniając wymaganie, że każdy koszt należy zwiększyć o 30%. Ostatnią czynnością pozostaje zwrócenie obiektu `ResultCostDTO`. Oto zaimplementowany serwis:

LISTING 1.15. Kod implementacji serwisu CalculatorCostService

```
public class CalculatorCostService : ICalculatorCostService
    {
        private readonly CalculatorContext context;
        private readonly ICityService cityService;
        private readonly IModuleService moduleService;
        private readonly ISearchHistoryService searchHistoryService;

        public CalculatorCostService(
                CalculatorContext context,
                ICityService cityService,
                IModuleService moduleService,
                ISearchHistoryService searchHistoryService)
        {
            this.context = context;
            this.cityService = cityService;
            this.moduleService = moduleService;
            this.searchHistoryService = searchHistoryService;
        }

        OperationResultDTO ICalculatorCostService.CalculateCost(string cityName,
        ↪ModuleListDTO moduleListDTO)
        {
            var city = cityService.GetCityByName(cityName);

            if (city == null)
            {
                return new OperationErrorDTO { Code = 404, Message = $"City with
                ↪name: {cityName} doesn't exist" };
            }

            var modulesCost = city.TransportCost;

            foreach (String moduleName in moduleListDTO.ModuleList)
            {
                var module = moduleService.GetModuleByName(moduleName);

                if (module == null)
                {
                    return new OperationErrorDTO { Code = 404, Message = $"Module
                    ↪with name: {moduleName} doesn't exist" };
                }
```

```
                modulesCost = modulesCost + module.Price + (module.AssemblyTime *
                ↪city.CostOfWorkingHour);
            }

            modulesCost = modulesCost * 1.3;

            return new OperationSuccesDTO<ResultCostDTO> { Message = "Success", Result
            ↪= new ResultCostDTO { Cost = modulesCost, InSearchHistory = false } };
        }
    }
```

Ostatnim serwisem do zaprogramowania jest ShowResultService. W głównej mierze będzie on wykorzystywał inne serwisy. W parametrach jedynej jego metody otrzymujemy to samo, co w dwóch poprzednio omawianych metodach, którym poświęciliśmy bardziej szczegółowy opis — nazwę miasta i listę nazw modułów. Sprawdzamy, czy w historii jest rekord z przekazanymi parametrami; jeśli tak, to pobieramy potrzebne wartości i zwracamy wynik w postaci ResultCostDTO. Jeśli nie, to wywołujemy metodę CalculateCost z serwisu liczącego koszt, następnie przed zwróceniem wyniku tworzymy obiekt SearchHistory na bazie przekazanych parametrów i obliczonego kosztu, po czym zapisujemy go do bazy danych za pomocą odpowiedniej metody z serwisu SearchHistoryService. Po tej operacji zwracamy wynik. Poprawna implementacja wygląda następująco:

LISTING 1.16. Kod implementacji serwisu ShowResultService

```
public class ShowResultService : IShowResultService
{
    private readonly ISearchHistoryService searchHistoryService;
    private readonly ICalculatorCostService calculatorCostService;
    private readonly ICityService cityService;

    public ShowResultService(
        ISearchHistoryService searchHistoryService,
        ICalculatorCostService calculatorCostService,
        ICityService cityService)
    {
        this.searchHistoryService = searchHistoryService;
        this.calculatorCostService = calculatorCostService;
        this.cityService = cityService;
    }

    public ResultCostDTO PresentResult(string cityName, ModuleListDTO moduleListDTO)
    {
        var checkInHistory = searchHistoryService.GetSearchHistory(cityName,
        ↪moduleListDTO);
        OperationSuccesDTO<ResultCostDTO> calculateCost = null;

        if (checkInHistory.InSearchHistory == true)
        {
            return new ResultCostDTO { Cost = checkInHistory.Cost, InSearchHistory =
            ↪checkInHistory.InSearchHistory };
        }
```

Kup ksi k Pole ksi k

```
        try
        {
            calculateCost = (OperationSuccesDTO<ResultCostDTO>)calculatorCost
            ↪Service.CalculateCost(cityName, moduleListDTO);
        }
        catch
        {
            return new ResultCostDTO { Cost = -1, InSearchHistory = false };
        }
        var city = cityService.GetCityByName(cityName);

        SearchHistory searchHistory = new SearchHistory
        {
            CityId = city.Id,
            ProductionCost = calculateCost.Result.Cost,
            ModuleName1 = moduleListDTO.ModuleList.Count > 0 ?
            ↪moduleListDTO.ModuleList[0] : string.Empty,
            ModuleName2 = moduleListDTO.ModuleList.Count > 1 ?
            ↪moduleListDTO.ModuleList[1] : string.Empty,
            ModuleName3 = moduleListDTO.ModuleList.Count > 2 ?
            ↪moduleListDTO.ModuleList[2] : string.Empty,
            ModuleName4 = moduleListDTO.ModuleList.Count > 3 ?
            ↪moduleListDTO.ModuleList[3] : string.Empty,
        };

        searchHistoryService.AddSearchHistory(searchHistory);
        return new ResultCostDTO { Cost = calculateCost.Result.Cost,
        ↪InSearchHistory = false };
    }
}
```

Ostatnią rzeczą jest zarejestrowanie użytych klas i interfejsów w mechanizmie *dependency injectora*. Za pomocą *NutgetPackage* instalujemy paczkę *SimpleInjector* oraz *SimpleInjector.Integration.WebApi*. W pliku *Global.asax*, w głównej metodzie programu rejestrujemy wszystkie użyte (wstrzykiwane) serwisy w programie. Wygląda to następująco:

LISTING 1.17. Kod rejestrujący serwisy

```
protected void Application_Start()
      {
          var container = new Container();
          container.Options.DefaultScopedLifestyle = new AsyncScopedLifestyle();

          container.Register<CalculatorContext>(Lifestyle.Scoped);
          container.Register<ICityService, CityService>(Lifestyle.Scoped);
          container.Register<IModuleService, ModuleService>(Lifestyle.Scoped);
          container.Register<ICalculatorCostService, CalculatorCostService>
          ↪(Lifestyle.Scoped);
          container.Register<ISearchHistoryService, SearchHistoryService>
          ↪(Lifestyle.Scoped);
          container.Register<IShowResultService, ShowResultService>(Lifestyle.Scoped);

          container.Verify();
```

```
        GlobalConfiguration.Configuration.DependencyResolver =
            new SimpleInjectorWebApiDependencyResolver(container);
        AreaRegistration.RegisterAllAreas();
    }
```

Implementacja kontrolerów

Kolejnym etapem jest zaprogramowanie kontrolerów. W programie pełnią one rolę wystawianego na świat interfejsu, a mówiąc mniej metaforycznie, zawierają punkty dostępu do poszczególnych funkcjonalności projektu. Uzyskujemy do nich dostęp poprzez wywołanie odpowiedniego żądania `http/https`. Klasyczna składnia żądania ma format `nazwa_kontrolera/nazwa_metody/ewentualne_parametry`. W rzeczywistości punkty te nazywają się endpointami i są wywołaniem metody o danej nazwie z parametrami przekazanymi w żądaniu. Jest kilka typów endpointów; podstawowe cztery to: `Get`, `Post`, `Put` i `Delete`. `Get` wykorzystujemy do pobierania danych i przekazujemy parametry w żądaniu http, `Post` jest wykorzystywany do tworzenia nowych obiektów bądź „wkładania" danych do serwisów. Dane w tym przypadku mogą być przesyłane w ciele żądania. `Put` ma taka samą zasadę działania jak `Post`, z tą różnicą, że znajduje zastosowanie przy modyfikacji danych już istniejących. `Delete` jest stosowany przy operacjach związanych z usuwaniem obiektów. Typ metody zazwyczaj jest określany przez atrybut ustawiany przed metodą.

Kontrolery będą w swych konstruktorach miały wstrzykiwane odpowiednie serwisy, czyli po raz kolejny użyjemy bardzo praktycznego *dependency injectora*. Kontrolery będą bazować na metodach z serwisów, gdzie znajduje się cała logika biznesowa. Kontroler powinien sprowadzać się do prostej implementacji powierzonego mu zadania. Na pewno w każdym kontrolerze należy sprawdzić, czy przekazywany do niego parametr (o ile taki istnieje) nie jest nullem, gdyż to może w skrajnych przypadkach powodować późniejsze przerwanie programu. Zarówno w serwisach, jak i w kontrolerach, często implementowany jest wzorzec CRUD. Podobnie będzie i w omawianym programie. Jako pierwszy na warsztat weźmiemy kontroler `CitiesController`, który będzie miał wstrzykiwany w konstruktorze interfejs `ICityService`. Klikamy prawym przyciskiem myszy na folder *Controllers* i wybieramy *Add/Controller*. W pojawiającym się oknie wybieramy *WebAPI 2 — empty*. Dodajemy kontroler, nadając mu nazwę `CitiesController`.

Kontroler, zgodnie z wymaganiami klienta, będzie zawierał następujące metody:

- `GetCities`
- `GetCityByName`
- `UpdateTransportCost`
- `UpdateCostOfWorkingHour`
- `AddCity`
- `DeleteCity`

Pełna ich implementacja wygląda następująco:

LISTING 1.18. Kod kontrolera CitiesController

```
public class CitiesController : ApiController
  {
      private readonly ICityService cityService;

      public CitiesController(ICityService cityService)
      {
          this.cityService = cityService;
      }

      [HttpGet]
      [Route("Cities/GetCities")]
      public IHttpActionResult GetCities()
      {
          return Content(HttpStatusCode.OK, cityService.GetCities().Result);
      }

      [HttpGet]
      [Route("Cities/GetCityByName/{name}")]
      [ResponseType(typeof(City))]
      public IHttpActionResult GetCityByName(string name)
      {
          City city = cityService.GetCityByName(name);
          if (city == null)
          {
              return NotFound();
          }
          return Content<City>(HttpStatusCode.OK, city);
      }

      [HttpPut]
      [Route("Cities/UpdateTransportCost")]
      public IHttpActionResult UpdateTransportCost(City city)
      {
          var response = cityService.UpdateTransportCost(city.Name,
          ↪city.TransportCost);

          if (response.Message.Equals("Success"))
          {
              return Content(HttpStatusCode.OK, response.Message);
          }
          return Content(HttpStatusCode.BadRequest, response.Message);
      }

      [HttpPut]
      [Route("Cities/UpdateCostOfWorkingHour")]
      public IHttpActionResult UpdateCostOfWorkingHour(City city)
      {
          var response = cityService.UpdateCostOfWorkingHour(city.Name,
          ↪city.CostOfWorkingHour);

          if (response.Message.Equals("Success"))
```

Kup ksi k Pole ksi k

```
            {
                return Content(HttpStatusCode.OK, response.Message);
            }
            return Content(HttpStatusCode.BadRequest, response.Message);
        }

        [HttpPost]
        [Route("Cities/AddCity")]
        [ResponseType(typeof(void))]
        public IHttpActionResult AddCity(City city)
        {
            if (city == null)
            {
                return Content(HttpStatusCode.BadRequest, "Object city is null");
            }

            var response = cityService.AddCity(city);

            if (response.Message.Equals("Success"))
            {
                return Content(HttpStatusCode.OK, response.Message);
            }

            return Content(HttpStatusCode.BadRequest, "Error");
        }

        [HttpDelete]
        [Route("Cities/DeleteCity/{name}")]
        [ResponseType(typeof(void))]
        public IHttpActionResult DeleteCity(string name)
        {
            if (name == null)
            {
                return Content(HttpStatusCode.BadRequest, "City name is null");
            }

            var response = cityService.DeleteCity(name);

            if (response.Message.Equals("Success"))
            {
                return Content(HttpStatusCode.OK, response.Message);
            }

            return Content(HttpStatusCode.BadRequest, "Error");
        }
    }
```

Utworzyliśmy kontroler, jednak to nie wystarczy; aby działał poprawnie, musimy poinformować aplikację o ścieżce routingu. Będzie ona obowiązywała dla wszystkich tworzonych kontrolerów. W tym celu w pliku *Global.asax* dodajemy taki wpis:

Kup książkę Poleć książkę

LISTING 1.19. Kod konfigurujący routing w aplikacji

```
…
            AreaRegistration.RegisterAllAreas();
            GlobalConfiguration.Configure(WebApiConfig.Register);
            FilterConfig.RegisterGlobalFilters(GlobalFilters.Filters);
…
```

Analogicznie postępujemy w przypadku pozostałych 3 kontrolerów: `ModulesController`, `SearchHistoriesControler` oraz `ShowResultController`. Będą one zawierały następujące metody:

ModulesController	**SearchHistoriesController**	**ShowResultController**
GetModules	GetSearchHistory	GetCost
GetModuleByName	AddSearchHistory	
UpdateModule		
AddModule		
DeleteModule		

Spróbuj teraz sam zaimplementować wymienione wyżej metody, analogicznie jak w przypadku pierwszego kontrolera. Z jedną małą różnicą — w kontrolerze `ShowResult` parametry przekazywane w metodzie `GestCost` opakujemy w obiekt `ShowResultDTO`. Tworzymy go w folderze *ModelsDTO*. `GetCost` to metoda typu `POST`, zatem rekomendowane jest opakowanie parametrów w jeden obiekt.

LISTING 1.20. Kod modelu pomocniczego — ShowResultDTO

```
public class ShowResultDTO
{
    public string CityName { get; set; }

    public ModuleListDTO ModuleListDTO { get; set; }
}
```

Jestem pewien, że dasz sobie radę! Efekt prawidłowo wykonanej pracy powinien wyglądać tak jak poniższy kod. Oczywiście, jeśli wszystko się zgadza, możesz wykonać commit.

LISTING 1.21. Kod pozostałych kontrolerów

```
public class ModulesController : ApiController
    {
        private readonly IModuleService moduleService;

        public ModulesController(IModuleService moduleService)
        {
            this.moduleService = moduleService;
```

Kup ksi k Pole ksi k

```
        }

        [HttpGet]
        [Route("Module/GetModules")]
        public IHttpActionResult GetModules()
        {
            return Content<IList<Module>>(HttpStatusCode.OK,
            ➥moduleService.GetModules().Result);
        }

        [HttpGet]
        [Route("Module/GetModule/{name}")]
        public IHttpActionResult GetModuleByName(string name)
        {
            if (name == null)
            {
                return NotFound();
            }

            return Content<Module>(HttpStatusCode.OK,
            ➥moduleService.GetModuleByName(name));
        }

        [HttpPut]
        [Route("Module/UpdateModule")]
        public IHttpActionResult UpdateModule(Module module)
        {
            return Content<string>(HttpStatusCode.OK,
            ➥moduleService.UpdateModule(module).Message);
        }

        [HttpPost]
        [Route("Module/AddModule")]
        public IHttpActionResult AddModule(Module module)
        {
            return Content<string>(HttpStatusCode.OK,
            ➥moduleService.AddModule(module).Message);
        }

        [HttpDelete]
        [Route("Module/DeleteModule/{name}")]
        public IHttpActionResult DeleteModule(string name)
        {
            return Content<string>(HttpStatusCode.OK,
            ➥moduleService.DeleteModule(name).Message);
        }
    }

    public class SearchHistoriesController : ApiController
    {
        private readonly ISearchHistoryService searchHistoryService;

        public SearchHistoriesController(ISearchHistoryService searchHistoryService)
        {
            this.searchHistoryService = searchHistoryService;
        }
```

```
        [HttpGet]
        [Route("SearchHistories/GetSearchHistory")]
        public IHttpActionResult GetSearchHistory()
        {
            return Content<IList<SearchHistory>>(HttpStatusCode.OK,
            ➥searchHistoryService.GetSearchHistories().Result);
        }

        [HttpPost]
        [Route("SearchHistories/AddSearchHistory")]
        [ResponseType(typeof(void))]
        public IHttpActionResult AddSearchHistory(SearchHistory searchHistory)
        {
            if (searchHistory == null)
            {
                return Content(HttpStatusCode.BadRequest, "Object searchHistory
                ➥is null");
            }

            var response = searchHistoryService.AddSearchHistory(searchHistory);

            if (response.Message.Equals("Success"))
            {
                return Content(HttpStatusCode.OK, "Success");
            }

            return Content(HttpStatusCode.BadRequest, "Error");
        }
    }

    public class ShowResultController : ApiController
    {
        private readonly IShowResultService showResultService;

        public ShowResultController(IShowResultService showResultService)
        {
            this.showResultService = showResultService;
        }

        [HttpPost]
        [Route("ShowResult/Get")]
        public IHttpActionResult GetCost(ShowResultDTO showResultDTO)
        {
            var result = this.showResultService.PresentResult
            ➥(showResultDTO.CityName, showResultDTO.ModuleListDTO);

            if (result.Cost == -1)
            {
                return Content<string>(HttpStatusCode.ExpectationFailed, "Error,
                ➥propably bad module name");
            }
            return Content<ResultCostDTO>(HttpStatusCode.OK, this.showResultService.
            ➥PresentResult(showResultDTO.CityName, showResultDTO.ModuleListDTO));
        }
}
```

Kup ksi k Pole ksi k

LISTING 1.22. Komendy tworzące commit

```
git add .
git commit -m  "Added controlers"
```

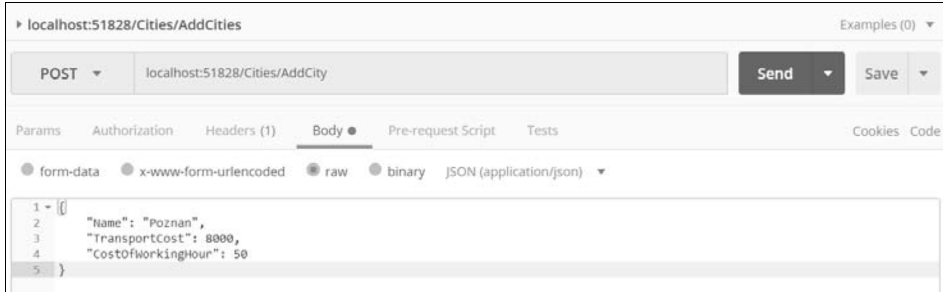
Wstępne testy przy użyciu narzędzi dedykowanych

Większość aplikacji posiada architekturę testową, sprawdzającą, czy kod działa poprawnie i zwraca odpowiednie rezultaty. Pomaga to zwłaszcza przy rozwijaniu aplikacji, gdy zmiany wprowadzane w kodzie wpływają na rezultaty z innego modułu i pisząc kod nie zauważamy tego, ale testy czuwają, by nas poinformować, że wprowadzone zmiany trzeba poprawić lub chociaż przeanalizować, czy na pewno uzyskiwane rezultaty we wszystkich modułach aplikacji są poprawne. Jednak pomiędzy fazą testowania a namiętnym klepaniem kodu jest jeszcze faza łącząca te dwa procesy; nie ma ona swojej fachowej nazwy, lecz to w niej podglądamy pierwsze rezultaty kontrolerów, które napisaliśmy. Możemy w tym celu wykorzystać jedno z kilku lub nawet kilkunastu narzędzi dedykowanych. W tym projekcie przyjrzymy się narzędziu o nazwie Postman. Można go pobrać za darmo w wersji desktopowej z oficjalnej strony narzędzia lub korzystać z jego wersji online, również darmowej, do której będziemy mieli dostęp — podobnie jak w przypadku wersji desktopowej — po utworzeniu darmowego konta i zalogowaniu się na nie.

Uogólniając, narzędzie to służy do generowania symulacji pracy aplikacji, a w szczególności symulacji obsługi endpointów z odpowiednio przez nas skonfigurowanymi parametrami i analizowania otrzymanych rezultatów pod kątem zgodności z oczekiwanymi. Sama obsługa programu Postman jest bardzo prosta.

Nasze testowanie zaczynamy od kontrolera `CitiesController`. Po uruchomieniu wersji desktopowej programu ustawiamy metodę na `Get` i podajemy adres taki jak w atrybucie konkretnego kontrolera — w tym przypadku pobierzemy wszystkie miasta z bazy danych, a zatem użyjemy ścieżki: *Cities/GetCities*. Oczywiście przed wysłaniem żądania uruchamiamy aplikację z poziomu Visual Studio. Oczekiwany wynik to pusta lista, gdyż nie mamy jeszcze żadnych miast w bazie danych. Żeby to zmienić, dodajmy dowolne miasto do bazy; użyjemy do tego metody `add` z omawianego kontrolera. Tym razem zmienimy typ używanej metody na `post`, zmieniamy więc ścieżkę do metody kontrolera, klikamy *Body*, zaznaczamy typ *raw* i w parametrach ustawiamy JSON, następnie definiujemy obiekt `City` w formacie json, tak jak to widać na poniższej ilustracji:

Kup ksi k Pole ksi k



RYSUNEK 1.3. *Przykładowa konfiguracja zapytania do API*

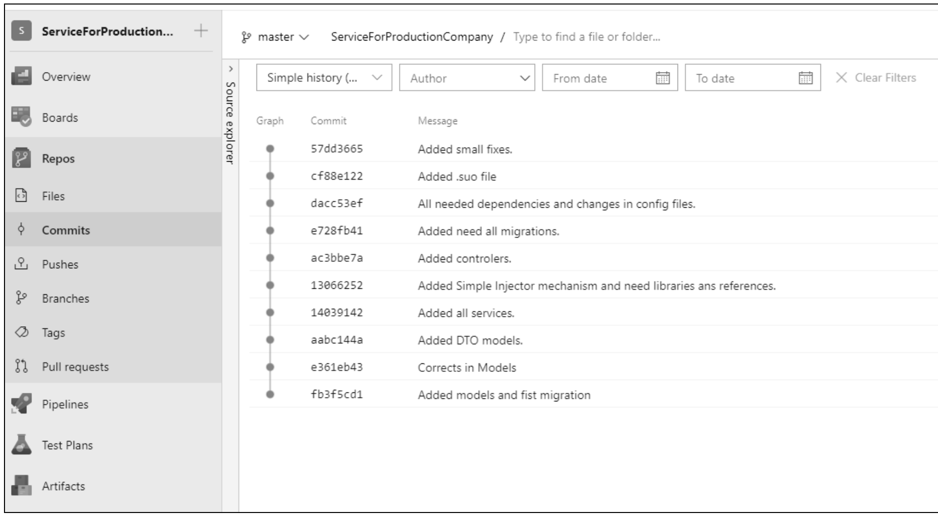
Powinniśmy otrzymać informację, że żądanie zakończyło się sukcesem. Analogicznie wykonujemy wszystkie pozostałe metody kontrolera. Wszystko powinno działać poprawnie. Dodatkowo rekomenduję, by zapisywać każde żądanie, żeby w przyszłości nie trzeba było ustawiać parametrów żądania od nowa; zamiast tego będziemy mogli użyć zapisanych wzorców. Zapisujemy żądanie klikając przycisk *Save*, znajdujący się po prawej stronie przycisku *Send*. Jeśli wszystkie metody działają poprawnie, to możemy otworzyć szampana i z dumą powiedzieć, że skończyliśmy realizację projektu, osiągając pełen sukces.

RYSUNEK 1.4. *Kolekcja zapytań do API*

Kup ksi k Pole ksi k

Podsumowanie projektu

Pierwszy projekt mamy za sobą. Mogłeś dzięki niemu nauczyć się używać kilku ważnych i przydatnych narzędzi z zasobów programisty .NET. Zobaczyłeś, jak wygląda tworzenie bazy danych przy użyciu frameworku EntityFramwok w opcji *Code first*. Zrobiliśmy migrację i wykonaliśmy podstawowe komendy w *package manager console*. W sposób optymalny użyliśmy kontekstu i zastosowaliśmy w serwisach jego funkcjonalności do działania na bazie danych. Dodatkowo opakowaliśmy zwracane obiekty w wewnętrzne klasy, które poprawiają czytelność kodu i sprawiają, że program jest bardziej przyjazny dla użytkownika, a także ułatwiają testowanie programiście. Na koniec przetestowaliśmy manualnie aplikacje przy użyciu Postmana. W API użyliśmy właściwości *dependency injector*, która ułatwia pracę programiście, oferując mu łatwe zarządzanie budowaniem obiektów. W cały projekt wpletliśmy praktyczne zastosowanie Gita, dzięki czemu mogłeś prześledzić krok po kroku, jak powstawało profesjonalne repozytorium kodu.



RYSUNEK 1.5. *Komplet commitów tworzących skończony projekt*

Powyższy obrazek napawa optymizmem. Klient na pewno jest zadowolony i bez żadnego „ale” odbierze projekt. My jednak nie zwalniamy tempa i przechodzimy do następnego projektu, który pozwoli nam odkryć kolejne możliwości w szeroko pojętym świecie .NET-u.

Kup ksi k Pole ksi k

PROGRAM PARTNERSKI

— GRUPY HELION —

1. ZAREJESTRUJ SIĘ
2. PREZENTUJ KSIĄŻKI
3. ZBIERAJ PROWIZJĘ

Zmień swoją stronę WWW w działający bankomat!

Dowiedz się więcej i dołącz już dzisiaj!
http://program-partnerski.helion.pl

Poznaj platformę .NET od strony praktycznej!

- **Projektuj rozwiązania, które zdobędą serca klientów**
- **Implementuj je zgodnie z najlepszymi wzorcami**
- **Testuj aplikacje przy użyciu właściwych narzędzi**

.NET stanowi jedną z najpopularniejszych platform do tworzenia i uruchamiania rozmaitych aplikacji, które można pisać w jednym z wielu wspieranych przez nią języków. Framework ten przez lata dojrzał i wzbogacił się o wiele przydatnych możliwości, które z powodzeniem spełniają oczekiwania nawet najbardziej wymagających programistów. .NET znajduje zastosowanie wszędzie tam, gdzie trzeba szybko i sprawnie dostarczyć działające rozwiązanie, niezależnie od tego, czy ma być ono aplikacją desktopową, czy internetową, utrzymywaną we własnej infrastrukturze lub w chmurze obliczeniowej.

Jeśli chcesz szybko poznać platformę .NET i dowiedzieć się, jak może Ci pomóc realizować różnego rodzaju projekty, sięgnij po książkę *Programuj z .NET. Praktyka ponad teorią*. Zgodnie z jej podtytułem autor przechodzi od razu do sedna, nie tracąc czasu na zbędne wstępy. Już od pierwszych stron przedstawia praktyczne sposoby zastosowania platformy .NET do rozwiązywania problemów napotykanych w codziennej pracy programistów. Dzięki tej książce sprawnie zdobędziesz wiedzę pozwalającą wkroczyć w świat profesjonalnego tworzenia aplikacji, które spełnią oczekiwania klientów.

- **Projektowanie aplikacji spełniających zadane wymagania**
- **Dobór odpowiednich technologii do zastanych problemów**
- **Tworzenie baz danych, back-endów i front-endów**
- **Integracja elementów składowych rozwiązania**
- **Zastosowanie zwinnych technik prowadzenia projektów**
- **Wdrażanie aplikacji w infrastrukturze chmurowej**
- **Zastosowanie narzędzi wspierających testy rozwiązań**

Twórz aplikacje jak prawdziwy profesjonalista!

ISBN 978-83-283-5644-3
9 788328 356443