Wydanie VII

packt

C# 11 i .NET 7

dla programistów aplikacji wieloplatformowych

Twórz aplikacje, witryny WWW oraz serwisy sieciowe za pomocą ASP.NET Core 7, Blazor i EF Core 7

Helion

Mark J. Price

Tytuł oryginału: C# 11 and .NET 7 — Modern Cross-Platform Development Fundamentals: Start building websites and services with ASP.NET Core 7, Blazor, and EF Core 7, 7th Edition

Tłumaczenie: Wojciech Moch

ISBN: 978-83-8322-687-3

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
https://helion.pl/user/opinie/c11n77
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Pliki z przykładami omawianymi w książce można znaleźć pod adresem:
https://ftp.helion.pl/przyklady/c11n77.zip

Helion S.A.
ul. Kościuszki 1c, 44-100 Gliwice
tel. 32 230 98 63
e-mail: *helion@helion.pl*
WWW: *https://helion.pl* (księgarnia internetowa, katalog książek)

Printed in Poland.

Spis treści

ROZDZIAŁ 4

Wprowadzenie do aplikacji sieciowych w ASP.NET Core

Rozdział 12

Trzecia część tej książki została poświęcona tworzeniu aplikacji sieciowych za pomocą ASP.NET Core. Dowiesz się, jak tworzyć pełne wieloplatformowe aplikacje, takie jak witryny WWW albo serwisy internetowe, a także aplikacje dla komputerów stacjonarnych i dla urządzeń mobilnych.

Firma Microsoft nazywa platformy przeznaczone do budowania aplikacji **modelami aplikacji** (ang. *app models*).

Zalecam sekwencyjne przejście przez ten i kolejne rozdziały, ponieważ w dalszych rozdziałach będziemy korzystać z projektów przygotowanych we wcześniejszych rozdziałach. Poza tym podczas lektury kolejnych rozdziałów zgromadzisz informacje i umiejętności ułatwiające radzenie sobie z bardziej złożonymi problemami, które pojawią się w późniejszych rozdziałach.

W tym rozdziale zostaną omówione następujące zagadnienia:

- czym jest ASP.NET Core;
- nowe funkcje w ASP.NET Core;
- struktury projektów;
- tworzenie modelu encji dla kolejnych rozdziałów;
- sposoby tworzenia aplikacji sieciowych.

Czym jest ASP.NET Core?

Ta książka jest poświęcona językowi C# i platformie .NET, dlatego przedstawię tutaj różne modele aplikacji, których można użyć do tworzenia praktycznych aplikacji, z jakimi zetkniemy się w pozostałych rozdziałach.

Uwaga

Firma Microsoft udostępnia rozbudowany przewodnik implementowania różnych modeli aplikacji. Jest to część dokumentacji .NET Application Architecture Guidance, dostępnej pod adresem: *https://www.microsoft.com/net/learn/architecture*.

Microsoft ASP.NET Core jest częścią historii tworzonych przez Microsoft technologii, przeznaczonych do projektowania aplikacji i serwisów WWW, które ewoluowały przez całe lata:

- **Active Server Pages** (**ASP**) powstała już w 1996 r. jako pierwsza próba Microsoftu stworzenia platformy pozwalającej na dynamiczne wykonywanie kodu aplikacji WWW po stronie serwera. Pliki ASP tworzone były w języku VBScript.
- **ASP.NET Web Forms** została wydana w 2002 r. jako część .NET Framework. Biblioteka ta miała pozwolić programistom niezwiązanym z siecią WWW, ale np. znającym język Visual Basic, na szybkie tworzenie aplikacji WWW przez przeciąganie i upuszczanie komponentów graficznych i dopisywanie do nich kodu reagującego na zdarzenia, wykorzystującego język Visual Basic lub C#. W nowych projektach należy unikać stosowania tej technologii i wybierać raczej ASP.NET MVC.
- **Windows Communication Foundation** (**WCF**) została wydana w 2006 r. Umożliwia programistom tworzenie serwisów SOAP oraz REST. Technologia SOAP ma bardzo duże możliwości, ale jest też bardzo skomplikowana, dlatego należy unikać jej stosowania, chyba że naprawdę potrzebne są oferowane przez nią funkcje, takie jak transakcje rozproszone albo złożone topologie wysyłania wiadomości.
- **ASP.NET MVC** powstała w 2009 r. i została zaprojektowana tak, żeby tworzyć jasny podział pomiędzy zadaniami dla programistów. I tak: **model** przechowuje tymczasowo dane, **widok** prezentuje je, stosując różne formaty w ramach interfejsu użytkownika, natomiast **kontroler** pobiera dane z modelu i przekazuje je do widoku. Takie rozdzielenie zadań umożliwia skuteczniejsze ponowne wykorzystywanie kodu oraz ułatwia tworzenie testów jednostkowych.

- **ASP.NET Web API** została wydana w 2012 r. Pozwala programistom na tworzenie serwisów HTTP lub REST, które są znacznie prostsze i pozwalają się lepiej skalować niż serwisy SOAP.
- **ASP.NET SignalR** powstała w 2013 r. Umożliwia komunikację w czasie rzeczywistym w ramach aplikacji WWW, ukrywając wszystkie niższe technologie i techniki, takie jak *Web Sockets* lub *Long Polling*. Pozwala to implementować na stronach WWW takie funkcje jak komunikacja na żywo, aktualizowanie na bieżąco ważnych danych, np. cen akcji. To wszystko działa w wielu różnych przeglądarkach, nawet jeżeli te nie obsługują nowoczesnych technologii, takich jak WebSocket.
- **ASP.NET Core** została wydana w 2016 r. i łączy nowoczesne implementacje starszych technologii pochodzących z .NET Framework, takich jak MVC, Web API i SignalR, z nowszymi technologiami, takimi jak Razor Pages, gRPC lub Blazor, dzięki czemu wszystkie one mogą działać w ramach nowoczesnego .NET. W ten sposób ta technologia może być stosowana niezależnie od platformy. W ASP.NET Core istnieje wiele szablonów pozwalających na rozpoczęcie pracy z wieloma obsługiwanymi technologiami.

Wskazówka

Dobra praktyka: Do tworzenia aplikacji i serwisów WWW wybieraj bibliotekę ASP.NET Core, ponieważ zawiera ona wiele technologii związanych z siecią WWW, jest nowoczesna i wieloplatformowa.

Klasyczna ASP.NET kontra ASP.NET Core

Jak dotąd cała biblioteka tworzona była jako jeden zestaw w ramach .NET Framework — *System.Web.dll*. Jest on ściśle powiązany z serwerem **IIS** (ang. *Internet Information Services*) tworzonym przez Microsoft wyłącznie dla systemów Windows. Przez lata zgromadziło się tam wiele różnych funkcji. Niestety, część z nich zupełnie nie nadaje się do tworzenia rozwiązań wieloplatformowych.

ASP.NET Core to całkowicie przebudowana ASP.NET. Usunięto z niej zależność od zestawu *System.Web.dll* i serwera IIS, a zamiast tego biblioteka składa się z modularnych, lekkich pakietów, tak jak całość nowoczesnego .NET. ASP.NET Core nadal pozwala używać IIS jako serwera WWW, ale udostępnia też znacznie lepsze rozwiązanie.

Wieloplatformowe aplikacje ASP.NET Core można tworzyć w systemach Windows, macOS oraz Linux. Microsoft przygotował nawet wieloplatformowy, bardzo wydajny serwer WWW o nazwie **Kestrel**. Całość stosu serwera została udostępniona w postaci otwartych źródeł.

Projekty tworzone na bazie ASP.NET Core 2.2 domyślnie stosują nowy wbudowany model hostowania stron. Pozwala to uzyskać zwiększenie wydajności o 400% w porównaniu do hostowania stron na serwerze Microsoft IIS, mimo to firma Microsoft zaleca stosowanie serwera Kestrel, gdyż pozwala on uzyskać jeszcze lepszą wydajność.

Tworzenie stron WWW za pomocą ASP.NET Core

Witryny WWW składają się z wielu stron ładowanych statycznie z systemu plików albo generowanych dynamicznie przez technologie działające po stronie serwera, takie jak ASP.NET Core. Przeglądarka WWW wysyła żądania `GET`, podając w nich adresy URL identyfikujące poszczególne strony. Może też manipulować danymi przechowywanymi na serwerze, stosując żądania `POST`, `PUT` i `DELETE`.

W przypadku witryn WWW przeglądarka traktowana jest jak warstwa prezentacji, a niemal wszystkie prace wykonywane są po stronie serwera. Po stronie klienta mogą się pojawiać niewielkie porcje kodu JavaScript, implementujące różne funkcje prezentacyjne, takie jak karuzele obrazów.

ASP.NET Core udostępnia kilka technologii tworzenia witryn WWW:

- **ASP.NET Core Razor Pages** i **biblioteki klas Razor** są metodą dynamicznego generowania kodu HTML dla prostych stron WWW. Więcej informacji na ten temat znajdziesz w rozdziale 13., „Tworzenie witryn WWW przy użyciu ASP.NET Core Razor Pages".
- **ASP.NET Core MVC** jest implementacją wzorca Model-View-Controller, czyli bardzo popularnej metody tworzenia złożonych witryn WWW. Więcej informacji na ten temat znajdziesz w rozdziale 14., „Tworzenie aplikacji WWW przy użyciu ASP.NET Core MVC".
- **Technologia Blazor** umożliwia tworzenie serwerowych i klienckich komponentów i interfejsów użytkownika za pomocą języka C#, a nie za pomocą frameworków zbudowanych w języku JavaScript, takich jak Angular, React lub Vue. **Blazor Web Assembly** uruchamia nasz kod w przeglądarce, gdzie działa on tak samo jak kod z frameworków języka JavaScript. **Blazor Server** uruchamia kod na serwerze, gdzie dynamicznie aktualizuje zawartość strony WWW. Więcej informacji na temat technologii Blazor znajdziesz w rozdziale 16., „Tworzenie interfejsów użytkownika w technologii Blazor". Ta technologia nie jest przeznaczona wyłącznie do tworzenia witryn WWW, lecz można jej też używać do tworzenia hybrydowych aplikacji mobilnych i stacjonarnych, po umieszczeniu jej w aplikacji .NET MAUI.

Tworzenie witryn WWW za pomocą systemu zarządzania treścią

Niektóre witryny zawierają bardzo dużo treści, a każdorazowe angażowanie programistów przy zmianach tych treści byłoby bardzo niepraktyczne. **Systemy zarządzania treścią** (ang. *content management system* — **CMS**) umożliwiają zdefiniowanie struktury i szablonów zapewniających spójność wyglądu, którymi później zarządzają właściciele witryny. Mogą oni tworzyć zupełnie nowe strony albo inne bloki treści, aktualizować już istniejące i mieć niezbitą pewność, że całość będzie świetnie wyglądać na ekranach urządzeń użytkowników.

Istnieje wiele różnych systemów zarządzania treścią działających na różnych platformach sieciowych, takich jak WordPress dla PHP albo Django dla Pythona. Rozbudowane systemy CMS zbudowane w nowoczesnym .NET to Optimizely Content Cloud, Piranha CMS albo Orchard Core.

Główną zaletą stosowania systemu CMS jest możliwość korzystania z przyjaznego interfejsu użytkownika do zarządzania treściami. Właściciele witryn mogą się zalogować do swojego systemu i sami zarządzać wszystkimi treściami. Tak przygotowany materiał jest następnie renderowany i pokazywany użytkownikom za pośrednictwem kontrolerów i widoków ASP.NET MVC albo serwisowych punktów końcowych w sieci WWW, nazywanych też **headless CMS**, które dostarczają treści dla aplikacji zaimplementowanych dla urządzeń mobilnych lub stacjonarnych i innych.

W tej książce nie omawiam systemów CMS tworzonych w .NET, dlatego pod poniższym adresem podaję linki do artykułów omawiających ten temat:

> *https://github.com/markjprice/cs11dotnet7/blob/main/book-links.md#net-content-management-systems*

Tworzenie aplikacji WWW za pomocą framework SPA

Aplikacje WWW, znane też pod nazwą **aplikacji jednostronicowych** (ang. *Single-Page application* — **SPA**), składają się z pojedynczej strony WWW zbudowanej za pomocą technologii działającej w przeglądarce, takiej jak Blazor WebAssembly, Angular, React, Vue, albo innych własnościowych bibliotek JavaScript, które są w stanie wysyłać żądania do serwera sieciowego, aby pobierać dodatkowe informacje i przesyłać zaktualizowane dane. Wykorzystują do tego standardowe formaty serializacji, takie jak XML lub JSON. Typowym przykładem tego rodzaju aplikacji mogą być aplikacje Google, np.: Gmail, Maps lub Docs.

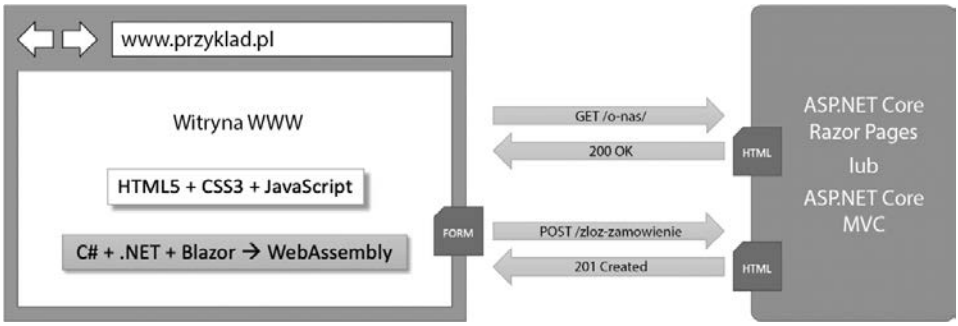
W przypadku aplikacji WWW używane są biblioteki języka JavaScript działające po stronie klienta, gdzie obsługują złożone interakcje z użytkownikiem. Mimo to główne procesy przetwarzania i dostępu do danych realizowane są po stronie serwera, ponieważ przeglądarki mają tylko ograniczony dostęp do zasobów systemowych.

JavaScript jest językiem o luźnym typowaniu, który nie jest przygotowany do obsługi złożonych projektów, dlatego większość bibliotek JavaScriptu używa dzisiaj języka

Microsoft TypeScript, który wprowadził silne typowanie oraz wiele funkcji nowoczesnych języków programowania umożliwiających realizację złożonych implementacji.

W .NET SDK dostępne są szablony projektów dla aplikacji SPA w języku JavaScript i Type-Script, jednak w tej książce nie będziemy zajmować się metodami tworzenia aplikacji SPA w tym języku, mimo że zazwyczaj wykorzystują one system ASP.NET Core po stronie serwera. To jest jednak książka o C#, a nie o innych językach programowania.

Podsumowując: język C# i platforma .NET mogą być używane zarówno po stronie serwera, jak i po stronie klienta, tworząc rozbudowane witryny, tak jak na rysunku 12.1.



Rysunek 12.1. Sposób użycia języka C# i środowiska .NET do tworzenia serwisów sieciowych po stronie serwera i po stronie klienta

Tworzenie serwisów sieciowych

Co prawda nie będziemy zajmować się tworzeniem aplikacji SPA w języku JavaScript lub TypeScript, ale nauczymy się budować serwisy sieciowe za pomocą **ASP.NET Core Web API** i wywoływać je z kodu działającego po stronie serwera w aplikacjach witryn ASP.NET Core. Później wywołamy przygotowaną usługę sieciową z aplikacji systemu Windows oraz wieloplatformowych aplikacji mobilnych.

Nie istnieją żadne formalne definicje, ale usługi są często opisywane pod kątem stopnia ich złożoności:

- **Serwis** — wszystkie funkcje używane przez aplikację kliencką znajdują się w jednym, monolitycznym serwisie.
- **Mikroserwis** — wiele serwisów skupiających się na realizacji małego zbioru funkcji.
- **Nanoserwis** — dany serwis realizuje tylko jedną funkcję. W przeciwieństwie do serwisów i mikroserwisów, które muszą działać 24/7/365, nanoserwisy są nieaktywne i są aktywowane dopiero w momencie ich wywołania, co pozwala oszczędzać zasoby i koszty.

Nowe funkcje w ASP.NET Core

Przez kilka ostatnich lat Microsoft bardzo szybko rozbudowywał możliwości biblioteki ASP.NET Core. Na poniższej liście podaję informacje, które wersje są obsługiwane na poszczególnych platformach .NET:

- ASP.NET 1.0 do 2.2 działają zarówno na .NET Core, jak i .NET Framework.
- ASP.NET Core 3.0 działa wyłącznie na .NET Core 3.0.

ASP.NET Core 1.0

ASP.NET Core 1.0 została wydana w czerwcu 2016 r. i skupiała się na implementowaniu interfejsu API odpowiedniego dla budowania nowoczesnych, wieloplatformowych stron WWW i serwisów sieciowych dla systemów Windows, macOS i Linux.

ASP.NET Core 1.1

ASP.NET Core 1.1 została wydana w listopadzie 2016 r. i była zbiorem poprawek błędów oraz ogólnych usprawnień wydajności i istniejących funkcji.

ASP.NET Core 2.0

ASP.NET Core 2.0 została wydana w sierpniu 2017 r. i skupiała się na wprowadzaniu nowych funkcji, takich jak Razor Pages, na łączeniu zestawów w ramach metapakietu `Microsoft.AspNetCore.All`, na zgodności z .NET Standard 2.0, udostępnianiu nowego modelu uwierzytelniania oraz zwiększeniu wydajności pracy.

Najważniejszymi funkcjami wprowadzonymi w ASP.NET Core 2.0 były ASP.NET Core Razor Pages, którą opiszę dokładnie w rozdziale 13., „Tworzenie witryn WWW przy użyciu ASP.NET Core Razor Pages" oraz ASP.NET Core OData.

ASP.NET Core 2.1

ASP.NET Core 2.1 została wydana w maju 2018 r., w ramach wydania z długoterminową obsługą (**LTS**), co oznacza, że była obsługiwana przez 3 lata aż do 21 sierpnia 2021 roku. Oznaczenie LTS zostało dodane do tego wydania dopiero w sierpniu 2018 roku razem z wersją 2.1.3.

To wydanie skupiało się na dodawaniu nowych funkcji, takich jak technologia **SignalR** związana z komunikacją w czasie rzeczywistym, **bibliotek klas związanych ze stronami Razor**, funkcji **ASP.NET Core Identity**, jak również na lepszej obsłudze protokołu HTTPS oraz regulacji związanych z europejską ustawą RODO. Najważniejsze tematy związane z tą wersją zostały zebrane w poniższej tabeli:

Funkcja	Rozdział	Temat
Biblioteki klas Razor	13.	Używanie bibliotek klas Razor
Obsługa RODO	14.	Tworzenie i używanie witryny ASP.NET Core MVC
Biblioteki Indentity UI	14.	Witryny ASP.NET Core MVC
Testy integracyjne	14.	Testowanie witryny ASP.NET Core MVC
Typ `[ApiController]`, `ActionResult<T>`	15.	Tworzenie projektu ASP.NET Core Web API
Szczegóły problemów	15.	Implementowanie kontrolera Web API
Interfejs `IHttpClientFactory`	15.	Konfigurowanie klientów HTTP za pomocą typu `HttpClientFactory`

ASP.NET Core 2.2

ASP.NET Core 2.2 została wydana w grudniu 2018 r. i skupiała się na usprawnianiu i tworzeniu API RESTful HTTP, aktualizowaniu szablonów projektów do wersji Bootstrap 4 i Angular 6, optymalizowaniu konfiguracji prowadzenia aplikacji w usłudze Azure, zwiększeniu wydajności pracy oraz tematach zebranych w poniższej tabeli:

Funkcja	Rozdział	Temat
HTTP/2 w serwerze Kestrel	13.	Klasyczne ASP.NET i nowoczesne ASP.NET Core
Model hosta	13.	Tworzenie projektu ASP.NET Core
Trasowanie punktów końcowych	13.	Czym jest trasowanie punktu końcowego
API Health Check	15.	Implementowanie API Health Check
Analizatory Open API	15.	Implementowanie analizatorów Open API i stosowanie konwencji

ASP.NET Core 3.0

ASP.NET Core 3.0 została wydana we wrześniu 2019 r. i skupiała się na jak najlepszym wykorzystaniu możliwości .NET Core 3.0 i .NET Standard 2.1, co oznacza, że zarzucona została obsługa .NET Framework. Oprócz tego dodano kilka przydatnych rozszerzeń oraz funkcje wymienione w poniższej tabeli:

Funkcja	Rozdział	Temat
Statyczne elementy w bibliotekach klas Razor	13.	Używanie bibliotek klas Razor
Nowe opcje rejestracji serwisów MVC	14.	Rozruch systemu ASP.NET Core MVC
Technologia Blazor po stronie serwera	16.	Technologia Blazor

ASP.NET Core 3.1

ASP.NET Core 3.1 została wydana w grudniu 2019 roku jako wydanie LTS, co oznacza, że będzie otrzymywała aktualizacje do grudnia 2022 roku. W tym wydaniu skupiono się na wprowadzaniu różnych usprawnień, takich jak obsługa klas częściowych w komponentach Razor i nowy znacznik pomocniczy `<component>`.

Blazor WebAssembly 3.2

Technologia Blazor WebAssembly 3.2 została udostępniona w maju 2020 roku. Jest to wydanie z linii Current (która teraz nosi nazwę *standard*), co oznacza, że projekty musiały zostać zaktualizowane do wersji .NET 5 w ciągu trzech miesięcy od pojawienia się tej wersji, czyli do lutego 2021 roku. Microsoft w końcu spełnił obietnicę dostarczenia całościowej technologii tworzenia aplikacji WWW w .NET. Dodatkowe informacje na temat Blazor Server i Blazor WebAssembly znajdziesz w rozdziale 16., „Tworzenie interfejsów użytkownika w technologii Blazor".

ASP.NET Core 5.0

ASP.NET Core 5.0 została wydana w listopadzie 2020 roku i skupiała się na poprawianiu błędów, zwiększaniu wydajności kodu przez zastosowanie pamięci podręcznej dla mechanizmu uwierzytelniania certyfikatów, wprowadzeniu mechanizmu dynamicznego kompresowania HPACK w nagłówkach protokołu HTTP/2 wysyłanych przez serwer Kestrel, wprowadzeniu nullowalnych atrybutów dla zestawów ASP.NET Core, zredukowaniu wielkości obrazów kontenerów oraz tematach zebranych w poniższej tabeli:

Funkcja	Rozdział	Temat
Metoda rozszerzająca, która umożliwia anonimowy dostęp do punktu końcowego	15.	Zabezpieczanie usług sieciowych
Metoda rozszerzająca JSON dla typów `HttpRequest` i `HttpResponse`	15.	Pobieranie w kontrolerze listy klientów w formacie JSON

ASP.NET Core 6.0

ASP.NET Core 6.0 została wydana w listopadzie 2021 roku i skupiała się na usprawnianiu wydajności pracy przez minimalizowanie ilości kodu niezbędnego do zaimplementowania serwisów i stron WWW, wprowadzaniu techniki .NET Hot Reload, nowych opcji hostowania dla technologii Blazor, takich jak aplikacje hybrydowe używające .NET MAUI. Oprócz tego wprowadzono tematy zebrane w poniższej tabeli:

Funkcja	Rozdział	Temat
Nowy szablon projektu pustej strony	13.	Szablon pustej strony
Minimalne API	15.	Implementowanie minimalnych Web API
Kompilacja AOT w Blazor WebAssembly	16.	Włączanie kompilacji wczesnej w Blazor WebAssembly

ASP.NET Core 7.0

ASP.NET Core 7.0 została wydana w listopadzie 2022 roku i skupiała się na znanych brakach w udostępnianych funkcjach, takich jak obsługa protokołu HTTP/3 i pamięć podręczna dla danych wyjściowych, a także na drobnych poprawkach w technologii Blazor oraz tematach zebranych w poniższej tabeli:

Funkcja	Rozdział	Temat
Dekompresja żądań HTTP	13.	Włączanie obsługi dekompresji żądań
Obsługa protokołu HTTP/3	13.	Włączanie obsługi protokołu HTTP/3
Zapisywanie danych wyjściowych w pamięci podręcznej	14.	Używanie filtra do umieszczania danych wyjściowych w pamięci podręcznej
Protokołowanie dodatkowych nagłówków	15.	Obsługa protokołowania dodatkowych nagłówków żądań w systemie W3CLogger
Obsługa klientów HTTP/3	15.	Włączanie obsługi HTTP/3 w klasie `HttpClient`
Puste szablony projektów tworzonych z użyciem Blazora	16.	Porównanie szablonów projektów tworzonych z użyciem Blazora
Obsługa zmiany lokalizacji	16.	Obsługa zdarzenia zmiany lokalizacji

Uwaga

Pełną informację o planach na przyszłość dla ASP.NET Core znajdziesz pod adresem *https://github.com/dotnet/aspnetcore/issues/39504*.

Struktury projektów

Jak należy zatem przygotowywać struktury projektów? Dotychczas tworzyliśmy tylko proste aplikacje konsoli, aby zilustrować funkcje języka lub biblioteki. W dalszej części tej książki utworzymy wiele projektów wykorzystujących różne technologie, które będą ze sobą współpracowały tworząc jedno rozwiązanie.

W wielkich i złożonych rozwiązaniach nawigowanie w kodzie bywa kłopotliwe. Podstawowym powodem planowania struktury projektów jest ułatwienie wyszukiwania komponentów. Dobrze jest też wybrać ogólną nazwę rozwiązania lub przestrzeni roboczej odzwierciedlającą przeznaczenie tworzonej aplikacji.

Zbudujemy teraz kilka projektów dla fikcyjnej firmy **Northwind**. Rozwiązaniu lub przestrzeni roboczej nadamy nazwę `PraktyczneAplikacje`, natomiast nazwy projektów uzupełnimy przedrostkiem `Northwind`.

Istnieje wiele sposobów tworzenia struktury projektu i nadawania nazw projektom i rozwiązaniom. Na przykład można wykorzystywać hierarchię folderów albo stosować wybrane konwencje nazewnicze. Jeżeli pracujesz w zespole, to upewnij się, że wiesz, jak robi to cały zespół.

Struktura projektów w rozwiązaniu lub przestrzeni roboczej

Dobrze jest mieć konwencję nazywania projektów w rozwiązaniu lub przestrzeni roboczej, dzięki której każdy programista będzie od razu wiedzieć, do czego służy dany projekt. Często stosowaną metodą jest użycie typu projektu, na przykład biblioteka klas lub aplikacja konsoli, strona WWW itd., podobnie jak w projektach z poniższej tabeli:

Nazwa	Opis
`Northwind.Wspolne`	Projekt biblioteki klas przechowującej wspólne typy, takie jak interfejsy, enumeracje, klasy, rekordy i struktury, które są używane w wielu projektach
`Northwind.Wspolne.ModeleEncji`	Projekt biblioteki klas przechowującej modele encji EF Core; modele encji zwykle są używane przez klienta i przez serwer, dlatego lepiej jest oddzielić zależności od konkretnych dostawców baz danych
`Northwind.Wspolne.KontekstDanych`	Projekt biblioteki klas przechowującej kontekst bazy danych EF Core z zależnościami od konkretnych dostawców

Nazwa	Opis
`Northwind.Web`	Projekt ASP.NET Core tworzący prostą witrynę WWW wykorzystującą mieszankę statycznych plików HTML i dynamicznych stron Razor Pages.
`Northwind.Razor.Komponenty`	Projekt biblioteki klas na potrzeby stron Razor wykorzystywany w wielu projektach
`Northwind.MVC`	Projekt ASP.NET Core dla złożonej witryny zbudowanej zgodnie ze wzorcem MVC, ułatwiającym tworzenie testów jednostkowych
`Northwind.WebAPI`	Projekt ASP.NET Core dla serwisu HTTP API; jest dobrym rozwiązaniem do zintegrowania z witrynami WWW, ponieważ umożliwia komunikację z dowolną biblioteką JavaScript albo Blazor.
`Northwind.BlazorServer`	Projekt serwera ASP.NET Core Blazor Server
`Northwind.BlazorWasm.Client`	Projekt klienta ASP.NET Core Blazor WebAssembly
`Northwind.BlazorWasm.Server`	Projekt serwera ASP.NET Core Blazor WebAssembly

Tworzenie modelu danych dla bazy danych Northwind

Praktyczne aplikacje zazwyczaj muszą pracować z danymi zapisanymi w relacyjnej bazie danych albo w innej składnicy danych. W tym rozdziale zdefiniujemy model danych dla bazy Northwind przechowywanej na serwerze SQL Server lub SQLite. Przygotowany tu model danych będzie używany w większości aplikacji, jakie będziemy tworzyć w kolejnych rozdziałach.

Pliki skryptów *Northwind4SQLServer.sql* i *Northwind4SQLite.sql* nie są identyczne. Skrypt dla SQL Server tworzy 13 tabel oraz powiązane z nimi widoki i procedury składowane. Skrypt dla SQLite jest jego uproszczoną wersją, tworzącą tylko 10 tabel. Wynika to z faktu, że SQLite nie obsługuje wielu funkcji dostępnych w SQL Server. Główne projekty z tej książki wykorzystują jednak tylko 10 tabel, dlatego wszystkie prezentowane tu zadania można wykonać z dowolnym serwerem baz danych.

Dokładny opis procedury instalowania serwera SQL Server lub SQLite znajduje się w rozdziale 10., „Praca z bazami danych przy użyciu Entity Framework Core”. Znajdziesz w nim też instrukcje dotyczące instalowania narzędzia `dotnet-ef`, które wykorzystamy tutaj do utworzenia modelu encji na podstawie istniejącej bazy danych.

Instrukcję instalowania serwera SQL Server można znaleźć w repozytorium GitHub dla tej książki pod adresem *https://github.com/markjprice/cs11dotnet7/blob/main/docs/sql-server/README.md*.

Wskazówka

Dobra praktyka: Dobrze jest utworzyć osobny projekt biblioteki klas przeznaczony na model encji. Znacznie ułatwia to współdzielenie modelu przez centralny serwer oraz wszystkie aplikacje klienckie, mobilne oraz klientów Blazor WebAssembly.

Tworzenie biblioteki klas dla modelu encji Northwind

Teraz zdefiniujemy model encji w bibliotece klas, tak aby można było go używać w innych rodzajach projektów, w tym w modelach aplikacji klienckich. Jeżeli nie korzystasz z serwera SQL Server, to musisz utworzyć jedynie bibliotekę klas dla SQLite. Jeżeli jednak używasz serwera SQL Server, to lepiej będzie utworzyć dwie biblioteki klas — jedną dla SQL Server, a drugą dla SQLite. Da Ci to później możliwość wybierania serwera zależnie od sytuacji.

Wygenerujemy teraz automatycznie część modeli encji za pomocą narzędzi wiersza poleceń:

1. Użyj swojego edytora kodu, aby utworzyć nowy projekt, używając ustawień z następującej listy:
   - Szablon projektu: *Biblioteka klas* lub `classlib`.
   - Plik lub folder projektu: *Northwind.Wspolne.ModeleEncji.Sqlite*.
   - Plik rozwiązania lub folder obszaru roboczego: *PraktyczneAplikacje*.
2. W pliku projektu *Northwind.Wspolne.ModeleEncji.Sqlite* dodaj referencję pakietu dostawcy danych SQLite oraz pakietu projektowania w EF Core:

```
<ItemGroup>
  <PackageReference
    Include="Microsoft.EntityFrameworkCore.Sqlite"
    Version="7.0.0" />
  <PackageReference
    Include="Microsoft.EntityFrameworkCore.Design"
    Version="7.0.0">
  <PrivateAssets>all</PrivateAssets>
  <IncludeAssets>runtime; build; native; contentfiles; analyzers;
  ↪buildtransitive</IncludeAssets>
  </PackageReference>
</ItemGroup>
```

3. Usuń plik *Class1.cs*.
4. Skompiluj projekt.

5. Utwórz plik *Northwind.db*, kopiując plik *Northwind4SQLite.sql* do folderu *PraktyczneAplikacje*, a następnie wprowadzając poniższe polecenie w oknie terminala lub wiersza poleceń:

```
sqlite3 Northwind.db -init Northwind4SQLite.sql
```

6. Poczekaj cierpliwie, ponieważ ten skrypt potrzebuje kilku chwil na utworzenie całej struktury bazy danych, co widać też na poniższym wydruku:

```
-- Loading resources from Northwind4SQLite.sql
SQLite version 3.35.5 2022-04-19 14:49:49
Enter ".help" for usage hints.
sqlite>
```

7. Naciśnij klawisze *Ctrl+C* w systemie Windows lub *Cmd+D* w systemie macOS, aby zakończyć tryb poleceń SLQite.

8. Otwórz okno wiersza poleceń lub terminala w folderze *Northwind.Wspolne.ModeleEncji.Sqlite*.

9. W wierszu poleceń wygeneruj model danych encji dla wszystkich tabel, używając poniższego polecenia:

```
dotnet ef dbcontext scaffold "Filename=../Northwind.db"
↪Microsoft.EntityFramework Core.Sqlite --namespace BibliotekaWspolna
↪--data-annotations
```

   Zwróć uwagę na następujące szczegóły:

   - Uruchamiane jest polecenie `dbcontext scaffold`.
   - Ciąg znaków połączenia to `"Filename=../Northwind.db"`.
   - Dostawca bazy danych to `Microsoft.EntityFrameworkCore.Sqlite`.
   - Przestrzeń nazw to `--namespace BibliotekaWspolna`.
   - Włączenie użycia atrybutów i płynnego API: `--data-annotations`.

10. Przejrzyj wypisywane komunikaty i ostrzeżenia, takie jak te z poniższego wydruku:

```
Build started...
Build succeeded.
To protect potentially sensitive information in your connection string,
you should move it out of source code. You can avoid scaffolding the
connection string by using the Name= syntax to read it from configuration
- see https://go.microsoft.com/fwlink/?linkid=2131148. For more
guidance on storing connection strings, see http://go.microsoft.com/
fwlink/?LinkId=723263.
```

Usprawnianie odwzorowania klas na tabele

Polecenie `dotnet-ef` generuje inny kod dla serwera SQL Server, a inny dla baz SQLite, ponieważ oba rozwiązania obsługują odmienne zestawy funkcji.

Na przykład w SQL Server kolumny tekstowe mogą mieć ograniczoną liczbę znaków. SQLite nie obsługuje tej funkcji. W związku z tym polecenie `dotnet-ef` wygeneruje atrybuty kontrolne zapewniające, że właściwości typu `string` będą zgodne z ograniczeniem liczby znaków wyłączenie dla serwera SQL Serve, ale pominie je dla SQLite, tak jak w poniższym kodzie:

```
// kod wygenerowany przez dostawcę SQLite
[Column(TypeName = "nvarchar (15)")]
public string CategoryName { get; set; } = null!;

// kod wygenerowany przez dostawcę SQL Server
[StringLength(15)]
public string CategoryName { get; set; } = null!;
```

Żaden z dostawców baz danych nie oznaczy nienullowalnej właściwości string jako wymaganej:

```
// brak kontroli dla właściwości nienullowalnych
public string CategoryName { get; set; } = null!;

// właściwość nullowalna
public string? Description { get; set; }

// dodaj atrybut, aby wymusić kontrolę w trakcie pracy aplikacji
[Required]
public string CategoryName { get; set; } = null!;
```

Wprowadzimy tu kilka małych zmian, aby usprawnić odwzorowania modelu encji i dodać reguły kontroli poprawności dla SQLite.

Uwaga

Pamiętaj, że całość kodu jest dostępna na serwerze FTP wydawnictwa Helion. Samodzielnie wpisując całość kodu wiele się nauczysz, ale wcale nie musisz tego robić. Możesz też pobrać kod z tej książki z adresu *https://ftp.helion.pl/przyklady/c11n77.zip*.

Po pierwsze, dodamy wyrażenie regularne sprawdzające, czy wartość pola `CustomerId` składa się z pięciu wielkich liter. Po drugie, wprowadzimy wymogi dotyczące długości ciągu znaków w tych modelach encji, co do których znamy już maksymalną dozwoloną długość wartości tekstowych.

1. Otwórz plik *Customer.cs* i dodaj wyrażenie regularne sprawdzające poprawność klucza głównego i pozwalające stosować wyłącznie wielkie litery alfabetu łacińskiego:

```
[Key]
[Column(TypeName = "nchar (5)")]
[RegularExpression("[A-Z]{5}")]
public string CustomerId { get; set; }
```

2. W swoim edytorze kodu wywołaj funkcję wyszukiwania i zamiany (w Visual Studio 2022 wybierz z menu pozycję *Edycja/Znajdź i zamień/Szybkie zamienianie*), włącz opcję *Użyj wyrażeń regularnych*, a potem w polu wyszukiwania wpisz poniższe wyrażenie regularne:

```
\[Column\(TypeName = "(nchar|nvarchar) \((.*)\)"\)\]
```

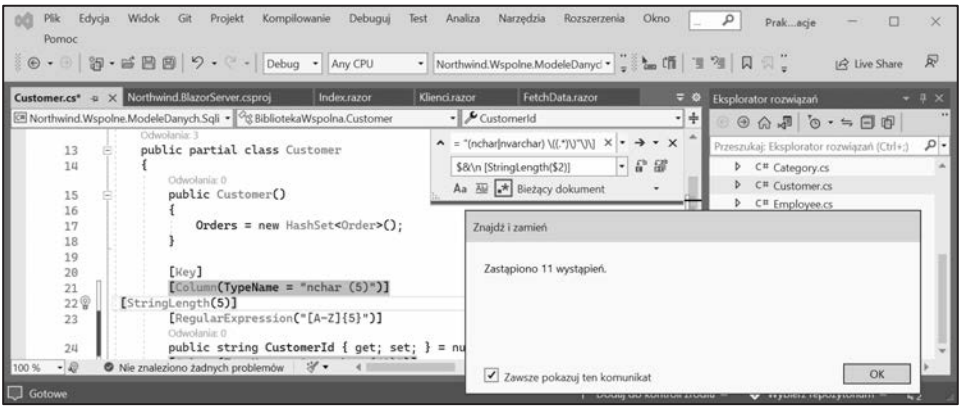
3. W polu *Zastąp...* wpisz nowe wyrażenie regularne:

```
$&\n    [StringLength($2)]
```

   Za znakiem nowego wiersza (`\n`) wstawiłem cztery znaki spacji, aby wprowadzić wcięcia właściwe dla mojego systemu, w którym używam dwóch spacji na każdy poziom wcięcia. W swoim kodzie możesz użyć tylu spacji, ile będzie Ci potrzebne.

4. Zaznacz opcję wyszukiwania i zamieniania we wszystkich plikach aktualnego projektu.

5. Uruchom wyszukiwanie i zamienianie, klikając przycisk *Zamień wszystkie*, tak jak na rysunku 12.2.



Rysunek 12.2. Wyszukaj i zamień wszystkie dopasowania wyrażenia regularnego w Visual Studio 2022

6. Zmień wszystkie właściwości związane z datą lub czasem, na przykład w pliku *Employee.cs*, tak żeby kod używał nullowalnego typu `DateTime`, a nie tablicy bajtów:

```
// przed zmianą
[Column(TypeName = "datetime")]
public byte[] BirthDate { get; set; }

// po zmianie
[Column(TypeName = "datetime")]
public DateTime? BirthDate { get; set; }
```

Wskazówka

Więcej informacji: Użyj w swoim edytorze kodu funkcji *Wyszukaj i zamień*, aby poszukać tekstu „datetime" i znaleźć wszystkie właściwości wymagające modyfikacji.

7. Zmień wszystkie właściwości związane z typem `money`, na przykład w pliku *Order.cs*, tak żeby kod używał nullowalnego typu `Decimal`, a nie tablicy bajtów:

```
// przed zmianą
[Column(TypeName = "money")]
public byte[] Freight { get; set; }

// po zmianie
[Column(TypeName = "money")]
public decimal? Freight { get; set; }
```

Wskazówka

Więcej informacji: Użyj w swoim edytorze kodu funkcji *Wyszukaj i zamień*, aby poszukać tekstu „money" i znaleźć wszystkie właściwości wymagające modyfikacji.

8. W pliku *Product.cs* zmień typ właściwości `Discontinued` z tablicy bajtów na `bool` i usuń inicjalizator przypisujący jej początkową wartość `null`:

```
[Column(TypeName = "bit")]
public bool Discontinued { get; set; }
```

9. W pliku *Category.cs* zmień typ właściwości `CategoryId` na `int`:

```
[Key]
public int CategoryId { get; set; }
```

10. W pliku *Category.cs* dopisz atrybut `Requred` do właściwości `CategoryName`:

```
[Required]
[Column(TypeName = "nvarchar (15)")]
[StringLength(15)]
public string CategoryName { get; set; }
```

11. W pliku *Customer.cs* dopisz atrybut `Requred` do właściwości `CompanyName`:

```
[Required]
[Column(TypeName = "nvarchar (40)")]
[StringLength(40)]
public string CompanyName { get; set; }
```

12. W pliku *Employee.cs*:
    - Zmień typ właściwości `EmployeeId` z `long` na `int`.
    - Dopisz atrybut `Required` do właściwości `FirstName` i `LastName`.
    - Zmień typ właściwości `ReportsTo` z `long?` na `int?`.
13. W pliku *EmployeeTerritory.cs*:
    - Zmień typ właściwości `EmployeeId` z `long` na `int`.
    - Dopisz atrybut `Required` do właściwości `TerritoryId`.
14. W pliku *Order.cs*:
    - Zmień typ właściwości `OrderId` z `long` na `int`.
    - Dopisz do właściwości `CustomerId` atrybut wyrażenia regularnego wymuszającego stosowanie pięciu wielkich liter.
    - Zmień typ właściwości `EmployeeId` z `long?` na `int?`.
    - Zmień typ właściwości `ShipVia` z `long?` na `int?`.
15. W pliku *OrderDetail.cs*:
    - Zmień typ właściwości `OrderId` z `long` na `int`.
    - Zmień typ właściwości `Productd` z `long` na `int`.
    - Zmień typ właściwości `Quantity` z `long` na `short`.
16. W pliku *Product.cs*:
    - Zmień typ właściwości `ProductId` z `long` na `int`.
    - Dopisz atrybut `Required` do właściwości `ProductName`.
    - Zmień typ właściwości `SupplierId` i `CategoryId` z `long?` na `int?`.
    - Zmień typ właściwości `UnitsInStock`, `UnitsOnOrder` i `ReorderLevel` z `long?` na `short?`.
17. W pliku *Shipper.cs*:
    - Zmień typ właściwości `ShipperId` z `long` na `int`.
    - Dopisz atrybut `Required` do właściwości `CompanyName`.
18. W pliku *Supplier.cs*:
    - Zmień typ właściwości `SupplierId` z `long` na `int`.
    - Dopisz atrybut `Required` do właściwości `CompanyName`.

19. W pliku *Territory.cs*:
    - Zmień typ właściwości `RegionId` z `long` na `int`.
    - Dopisz atrybut `Required` do właściwości `TerritoryId` i `TerritoryDescription`.

Skoro mamy już bibliotekę klas przechowującą klasy encji, możemy utworzyć bibliotekę klas z kontekstem bazy danych.

Tworzenie biblioteki klas z kontekstem bazy danych Northwind

Teraz zdefiniujemy kontekst bazy danych.

1. Do rozwiązania lub obszaru roboczego dodaj projekt biblioteki klas, używając następujących ustawień:
    - Szablon projektu: *Biblioteka klas* lub `classlib`.
    - Plik lub folder projektu: *Northwind.Wspolne.KontekstDanych.Sqlite*.
    - Plik rozwiązania lub folder przestrzeni roboczej: *PraktyczneAplikacje*.
2. W Visual Studio jako projekt startowy wybierz opcję *Bieżące zaznaczenie*. W Visual Studio Code wybierz projekt *Northwind.Wspolne.KontekstDanych.Sqlite* jako aktywny dla usługi OmniSharp.
3. Zmień zawartość pliku *Northwind.Wspolne.KontekstDanych.Sqlite.csproj*, dodając referencję do projektu *Northwind.Wspolne.ModeleEncji.Sqlite* oraz pakietu *Entity Framework Core for SQLite*, tak jak w poniższym kodzie:

```
<ItemGroup>
  <PackageReference
    Include="Microsoft.EntityFrameworkCore.SQLite"
    Version="7.0.0" />
</ItemGroup>

<ItemGroup>
  <ProjectReference
    Include="..\Northwind.Wspolne.ModeleEncji.Sqlite\
    ↪Northwind.Wspolne.ModeleEncji.Sqlite.csproj" />
</ItemGroup>
```

Wskazówka

Ostrzeżenie: W pliku projektu ścieżka zapisana w referencji projektu nie powinna zawierać znaków końca wiersza.

4. W projekcie *Northwind.Wspolne.KontekstDanych.Sqlite* usuń plik *Class1.cs*.
5. Skompiluj projekt *Northwind.Wspolne.KontekstDanych.Sqlite.*

6. Przenieś plik *NorthwindContext.cs* z folderu projektu *Northwind.Wspolne.ModeleEncji.Sqlite* do folderu projektu *Northwind.Wspolne.KontekstDanych.Sqlite*.

Wskazówka

Więcej informacji: W Visual Studio w okienku *Eksplorator rozwiązań* przeciągnięcie pliku między projektami spowoduje jego skopiowanie. Jeżeli podczas przeciągania przytrzymasz przycisk *Shift*, to plik zostanie przeniesiony. W Visual Studio Code w okienku *Explorer* przeciągnięcie pliku między projektami spowoduje jego przeniesienie. Aby skopiować plik, musisz go przeciągnąć, przytrzymując klawisz *Ctrl*.

7. W pliku *NorthwindContext.cs*, w metodzie `OnConfiguring`, usuń ostrzeżenie kompilatora dotyczące ciągu znaków połączenia.

Wskazówka

Dobra praktyka: Domyślny ciąg znaków połączenia będziemy pokrywać w tworzonych projektach, takich jak projekty stron WWW, które muszą współpracować z bazą danych Northwind. Oznacza to, że klasa wywiedziona z klasy `DbContext` musi udostępniać konstruktor przyjmujący parametr typu `DbContextOptions`, taki jak w poniższym kodzie:

```
public NorthwindContext(DbContextOptions<NorthwindContext> options)
  : base(options)
{
}
```

8. W pliku *NorthwindContext.cs*, w metodzie OnConfiguring, dopisz instrukcje sprawdzające aktualny katalog, aby dopasować się do sytuacji, gdy aplikacja uruchamiana jest w Visual Studio 2022 lub w wierszu poleceń Visual Studio Code, zgodnie z wyróżnieniami w poniższym kodzie:

```
protected override void OnConfiguring(DbContextOptionsBuilder
↪optionsBuilder)
{
  if (!optionsBuilder.IsConfigured)
  {
    string katalog = Environment.CurrentDirectory;
    string sciezka = string.Empty;

    if (katalog.EndsWith("net7.0"))
    {
      // Uruchomienie w katalogu <projekt>\bin\<Debug|Release>\net7.0
      sciezka = Path.Combine("..", "..", "..", "..", "Northwind.db");
```

```
        }
        else
        {
          // Uruchomienie w katalogu <projekt>
          sciezka = Path.Combine("..", "Northwind.db");
        }

        optionsBuilder.UseSqlite($"Filename={path}");
      }
    }
```

9. W metodzie `OnModelCreating` usuń wszystkie instrukcje płynnego API wywołujące metodę `ValueGeneratorNever` w celu skonfigurowania właściwości klucza głównego, takich jak `SupplierId`, tak jak w poniższym kodzie. Chodzi o to, żeby nigdy nie generować automatycznie wartości klucza. Można też wywołać metodę `HasDefaultValueSql`.

```
modelBuilder.Entity<Supplier>(entity =>
{
  entity.Property(e => e.SupplierId).ValueGeneratedNever();
});
```

Wskazówka

Więcej informacji: Jeżeli nie usuniemy z konfiguracji pokazanych wyżej instrukcji, to podczas dodawania nowego dostawcy właściwość `SupplierId` będzie zawsze miała wartość `0`, przez co będziemy mogli dodać tylko jednego dostawcę z takim identyfikatorem. Wszystkie kolejne próby dodania dostawcy będą kończyły się rzuceniem wyjątku.

10. W encji `Product` poinformuj serwer SQLite, że w przypadku kolumny `UnitPrice` można zmienić typ `decimal` na `double`. Metoda `OnModelCreating` powinna zostać uproszczona, tak jak w poniższym kodzie:

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
  modelBuilder.Entity<OrderDetail>(encja =>
  {
    encja.HasKey(e => new { e.OrderId, e.ProductId });

    encja.HasOne(d => d.Order)
      .WithMany(p => p.OrderDetails)
      .HasForeignKey(d => d.OrderId)
      .OnDelete(DeleteBehavior.ClientSetNull);

    encja.HasOne(d => d.Product)
```

```
        .WithMany(p => p.OrderDetails)
        .HasForeignKey(d => d.ProductId)
        .OnDelete(DeleteBehavior.ClientSetNull);
    });

    modelBuilder.Entity<Product>()
      .Property(produkt => produkt.UnitPrice)
      .HasConversion<double>();

    OnModelCreatingPartial(modelBuilder);
  }
```

11. W projekcie *Northwind.Wspolne.KontekstDanych.Sqlite* dodaj klasę o nazwie *NorthwindContextExtensions.cs* i zmień jej zawartość tak, żeby zdefiniować metodę rozszerzającą, która doda kontekst bazy danych Northwind do kolekcji zależnych serwisów:

```
using Microsoft.EntityFrameworkCore; // UseSqlite
using Microsoft.Extensions.DependencyInjection; // IServiceCollection

namespace BibliotekaWspolna;

public static class NorthwindContextExtensions
{
  /// <summary>
  /// Dodaje obiekt NorthwindContext do wskazanej kolekcji typu
  /// IServiceCollection. Używa dostawcy bazy danych Sqlite.
  /// </summary>
  /// <param name="serwisy"></param>
  /// <param name="sciezkaWzgledna">Ta wartość podmienia domyślną
  /// ścieżkę ".."</param>
  /// <returns>Kolekcja typu IServiceCollection,
  /// której można użyć do dodawania kolejnych serwisów.</returns>
  public static IServiceCollection AddNorthwindContext(
    this IServiceCollection serwisy, string sciezkaWzgledna = "..")
  {
    string sciezkaBazyDanych = Path.Combine(
      sciezkaWzgledna, "Northwind.db");

    serwisy.AddDbContext<NorthwindContext>(opcje =>
    {
      opcje.UseSqlite($"Data Source={sciezkaWzgledna}")
      opcje.LogTo(WriteLine, // Console
        new[] { Microsoft.EntityFrameworkCore
        .Diagnostics.RelationalEventId.CommandExecuting });
    });
```

```
        return serwisy;
      }
    }
```

12. Skompiluj dwie biblioteki klas i popraw ewentualne błędy kompilatora.

Tworzenie biblioteki klas modelu encji dla SQL Server

Jeżeli masz już skonfigurowaną bazę danych Northwind zgodnie z instrukcjami z rozdziału 10., „Praca z bazami danych przy użyciu Entity Framework Core", to nie musisz już nic więcej robić. Pozostaje tylko przygotować model danych za pomocą narzędzia `dotnet-ef`.

1. Użyj swojego edytora kodu, aby dodać projekt biblioteki klas, stosując ustawienia z następującej listy:
   - Szablon projektu: *Biblioteka klas* lub `classlib`.
   - Plik lub folder projektu: *Northwind.Wspolne.ModeleEncji.SqlServer*.
   - Plik rozwiązania lub folder obszaru roboczego: *PraktyczneAplikacje*.
2. W projekcie *Northwind.Wspolne.ModeleEncji.SqlServer* dodaj referencję pakietu dostawcy baz danych SQL Server oraz pakietu z funkcjami EF Core, tak jak w poniższym kodzie:

```
<ItemGroup>
  <PackageReference
    Include="Microsoft.EntityFrameworkCore.SqlServer"
    Version="7.0.0" />
  <PackageReference
    Include="Microsoft.EntityFrameworkCore.Design"
      Version="7.0.0">
      <PrivateAssets>all</PrivateAssets>
      <IncludeAssets>runtime; build; native; contentfiles; analyzers;
        buildtransitive</IncludeAssets>
  </PackageReference>
</ItemGroup>
```

3. Usuń plik *Class1.cs*.
4. Skompiluj projekt.
5. Otwórz okno wiersza poleceń lub terminala w folderze *Northwind.Wspolne.ModeleEncji.SqlServer*.
6. W wierszu poleceń wygeneruj model klas encji dla wszystkich tabel, stosując poniższe polecenie:

```
dotnet ef dbcontext scaffold "Data Source=.;Initial
Catalog=Northwind;Integrated Security=true;" Microsoft.
EntityFrameworkCore.SqlServer --namespace BibliotekaWspolna
➥--data-annotations
```

Zwróć uwagę na następujące szczegóły:

- Uruchamiane polecenie: `dbcontext scaffold`.
- Ciąg znaków połączenia: `"Data Source=.;Initial Catalog=Northwind;Integrated Security=true;"`.
- Dostawca bazy danych: `Microsoft.EntityFrameworkCore.SqlServer`.
- Przestrzeń nazw: `--namespace BibliotekaWspolna`.
- Włączenie użycia atrybutów i płynnego API: `--data-annotations`.

7. W pliku *Customer.cs* dopisz wyrażenie regularne sprawdzające poprawność wartości klucza głównego i pozwalające stosować wyłącznie wielkie litery alfabetu łacińskiego:

```
[Key]
[StringLength(5)]
[RegularExpression("[A-Z]{5}")]
public string CustomerId { get; set; } = null!;
```

8. W pliku *Customer.cs* dopisz atrybut `Required` do właściwości `CustomerId` i `CompanyName`.

9. Dodaj projekt biblioteki klas używając ustawień z następującej listy:

- Szablon projektu: *Biblioteka klas* lub `classlib`.
- Plik lub folder projektu: *Northwind.Wspolne.KontekstDanych.SqlServer*.
- Plik rozwiązania lub folder obszaru roboczego: *PraktyczneAplikacje*.
- W Visual Studio Code wybierz ten projekt jako aktywny dla usługi OmniSharp.

10. W projekcie *Northwind.Wspolne.KontekstDanych.SqlServer* dodaj referencję do projektu *Northwind.Wspolne.ModeleEncji.SqlServer* oraz do pakietu EF Core z dostawcą danych SQL Server:

```
<ItemGroup>
  <PackageReference
    Include="Microsoft.EntityFrameworkCore.SqlServer"
    Version="7.0.0" />
  </ItemGroup>

<ItemGroup>
  <ProjectReference Include=
    "..\Northwind.Wspolne.ModeleEncji.SqlServer\
    ➥Northwind.Wspolne.ModeleEncji.SqlServer.csproj" />
</ItemGroup>
```

Wskazówka

Ostrzeżenie: W pliku projektu ścieżka zapisana w referencji projektu nie powinna zawierać znaków końca wiersza.

11. W projekcie *Northwind.Wspolne.KontekstDanych.SqlServer* usuń plik klasy *Class1.cs*.
12. Skompiluj projekt *Northwind.Wspolne.KontekstDanych.SqlServer*.
13. Przenieś plik *NorthwindContext.cs* z folderu projektu *Northwind.Wspolne.ModeleEncji.SqlServer* do folderu projektu *Northwind.Wspolne.KontekstDanych.SqlServer*.
14. W projekcie *Northwind.Wspolne.KontekstDanych.SqlServer*, w pliku *NorthwindContext.cs*, usuń ostrzeżenie kompilatora dotyczące ciągu znaków połączenia.
15. W projekcie *Northwind.Wspolne.KontekstDanych.SqlServer* dodaj do projektu plik klasy o nazwie *NorthwindContextExtensions.cs* i zmień jego zawartość, definiując metodę rozszerzającą, która dodaje kontekst bazy danych Northwind do kolekcji zależnych serwerów:

```
using Microsoft.EntityFrameworkCore; // UseSqlServer
using Microsoft.Extensions.DependencyInjection; // IServiceCollection

namespace BibliotekaWspolna;

public static class NorthwindContextExtensions
{
  /// <summary>
  /// Dodaje obiekt typu NorthwindContext do wskazanej kolekcji
  /// IServiceCollection. Używa dostawcy danych SqlServer.
  /// </summary>
  /// <param name="serwisy"></param>
  /// <param name="polaczenie">Przypisz wartość, aby zmienić domyślną.</param>
  /// <returns>Kolekcja typu IServiceCollection, której można użyć do dodawania
  /// kolejnych serwisów.</returns>
  public static IServiceCollection DodajKontekstNorthwind(
    this IServiceCollection serwisy, string polaczenie =
      "Data Source=.;Initial Catalog=Northwind;"
      + "Integrated Security=true;MultipleActiveResultsets=true;")
  {
    serwisy.AddDbContext<NorthwindContext>(opcje =>
    {
      options.UseSqlServer(polaczenie);

      options.LogTo(WriteLine, // Console
```

```
            new[] { Microsoft.EntityFrameworkCore
            .Diagnostics.RelationalEventId.CommandExecuting });
        });

        return services;
      }
    }
```

16. Skompiluj obie biblioteki klas i popraw ewentualne błędy kompilacji.

Wskazówka

Dobra praktyka: Metoda `DodajKontekstNorthwind` przyjmuje parametry opcjonalne pozwalające zmienić domyślnie wpisane w kodzie nazwę plików bazy danych SQLite i ciąg znaków połączenia SQL Server. Pozwala to zachować elastyczność pracy, na przykład umożliwiając załadowanie tych wartości z pliku konfiguracyjnego.

Testowanie bibliotek klas

Przygotujmy teraz kilka testów jednostkowych, aby upewnić się, że nasze biblioteki klas działają poprawnie:

1. W swoim edytorze kodu dodaj nowy projekt o nazwie *Northwind.Wspolne.TestyJednostkowe*, umieszczając go w rozwiązaniu lub obszarze roboczym PraktyczneAplikacje. Do tworzenia projektu użyj szablonu *Projekt Testów xUnit [C#]* lub `xunit`.
2. W projekcie *Northwind.Wspolne.TestyJednostkowe* dodaj referencję projektu *Northwind.Wspolne.KontekstDanych* właściwą dla serwera SQLite lub SQL Server, zależnie od tego, którego z nich używasz. Zastosuj kod wyróżniony poniżej:

```
<ItemGroup>
  <!-- możesz zmienić Sqlite na SqlServer -->
  <ProjectReference Include="..\Northwind.Wspolne.KontekstDanych.Sqlite\
Northwind.Wspolne.KontekstDanych.Sqlite.csproj" />
</ItemGroup>
```

Wskazówka

Ostrzeżenie: Referencja projektu nie powinna zawierać znaków końca wiersza.

3. Skompiluj projekt *Northwind.Wspolne.TestyJednostkowe*.
4. Zmień nazwę pliku *UnitTest1.cs* na *TestyModeluEncji.cs*.

5. Zmień zawartość tego pliku, wpisując do niego dwa testy. Pierwszy z nich będzie próbował połączyć się z bazą danych, a drugi ma potwierdzać, że w bazie danych znajduje się osiem kategorii. Użyj poniższego kodu:

```
using BibliotekaWspolna; // NorthwindContext

namespace Northwind.Wspolne.TestyJednostkowe
{
   public class TestyModeluEncji
   {
      [Fact]
      public void TestPolaczeniaZBazaDanych()
      {
         using (NorthwindContext db = new())
         {
            Assert.True(db.Database.CanConnect());
         }
      }

      [Fact]
      public void TestLiczbyKategorii()
      {
         using (NorthwindContext db = new())
         {
            int oczekiwane = 8;
            int faktyczne = db.Categories.Count();
            Assert.Equal(oczekiwane, faktyczne);
         }
      }
   }
}
```

6. Uruchom testy jednostkowe:
   - Jeżeli używasz Visual Studio 2022, wybierz z menu pozycję *Test/Uruchom wszystkie testy*, a ich wyniki przejrzyj w okienku *Eksplorator testów*.
   - Jeżeli używasz Visual Studio Code, w oknie terminala dla projektu *Northwind.Wspolne.TestyJednostkowe* uruchom test, wydając polecenie `dotnet test`.
7. W wynikach testów powinna znaleźć się informacja, że oba testy zostały uruchomione i oba zakończyły się poprawnie. Jeżeli jakiś test zgłosi błąd, to spróbuj usunąć jego przyczynę. Na przykład, jeżeli używasz serwera SQLite, możesz sprawdzić, czy plik *Northwind.db* rzeczywiście znajduje się w katalogu rozwiązania (poziom wyżej względem katalogów projektów).

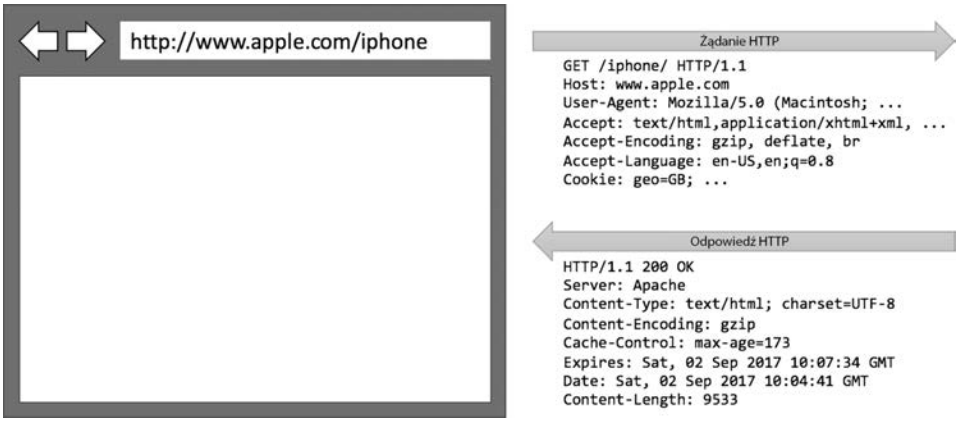
Tworzenie w sieci WWW

Tworzenie czegokolwiek w sieci WWW oznacza konieczność zaznajomienia się z protokołem **HTTP**, dlatego na początek omówię tę ważną technologię.

Protokół HTTP

Klient (lub **agent użytkownika**) komunikując się z serwerem WWW, wykonuje wywołania poprzez sieć, używając do tego protokołu **HTTP** (ang. *Hypertext Transfer Protocol*). Protokół HTTP stanowi techniczną bazę całej sieci WWW. Gdy zatem mówimy o aplikacjach i serwisach WWW, mamy na myśli, że komunikacja między klientem (często przeglądarką) a serwerem używa protokołu HTTP.

Klient wysyła żądanie HTTP dotyczące pewnego zasobu, takiego jak strona, identyfikując go adresem **URL** (ang. *Uniform Resource Locator*), a serwer odsyła odpowiedź HTTP. Całość odbywa się tak jak na rysunku 12.3.



Rysunek 12.3. Żądanie i odpowiedź HTTP

Możesz skorzystać z przeglądarki Google Chrome lub innej, aby zapisać poszczególne żądania i odpowiedzi.

Wskazówka

Dobra praktyka: Google Chrome jest przeglądarką dostępną dla większej liczby systemów operacyjnych niż jakakolwiek inna. Ma też wbudowane zaawansowane narzędzia dla programistów, dlatego często wybierana jest jako podstawowa przeglądarka. Swoje aplikacje testuj zawsze w Chrome oraz przynajmniej w dwóch innych przeglądarkach, takich jak Firefox oraz Safari dla macOS i iOS. Mniej ważne jest testowanie stron w przeglądarce Microsoft Edge, ponieważ od 2019 r. działa ona z mechanizmem renderującym Chromium. Microsoft Internet Explorer jest już używany niemal wyłącznie w intranetach istniejących w niektórych organizacjach.

Składniki adresu URL

Każdy adres URL (Uniform Resource Locator) składa się z kilku elementów:

- **Schemat**: *http* (jawny tekst) lub *https* (szyfrowane).
- **Domena**: W przypadku produkcyjnej strony WWW lub serwisu domeną najwyższego poziomu (ang. *top-level domain*) może być *przykład.pl*. Można też stosować różne poddomeny, takie jak *www*, *praca* lub *extranet*. W trakcie prac rozwojowych nad projektem zazwyczaj adresem rozwijanych stron WWW i serwisów jest *localhost*.
- **Numer portu**: W przypadku produkcyjnych stron WWW lub serwisów używany jest port 80 dla protokołu HTTP albo 443 dla HTTPS. Te numery portów są zwykle wnioskowane na podstawie używanego schematu. W trakcie prac rozwojowych stosowane są przeważnie inne numery portów, takie jak na przykład 5000 lub 5001. W ten sposób odróżnia się od siebie poszczególne witryny i serwisy prowadzone pod lokalnym adresem *localhost*.
- **Ścieżka**: Względna ścieżka do zasobu, na przykład `/klienci/niemcy`.
- **Zapytanie**: Metoda przekazywania wartości parametrów, na przykład `?kraj=Niemcy&tekst=buty`.
- **Fragment**: Referencja elementu na stronie WWW wyznaczana za pomocą identyfikatora, na przykład `#spistresci`.

Uwaga

Adres URL jest podzbiorem większego adresu **URI** (ang. *Uniform Resource Identifier*). Adres URL określa, gdzie można znaleźć wybrany element, z kolei adres URI umożliwia identyfikację elementu za pomocą adresu URL lub **URN** (ang. *Uniform Resource Name*).

Przypisywanie numerów portów do projektów z tej książki

W tej książce na potrzeby tworzonych stron WWW i serwisów sieciowych będziemy korzystać z domeny *localhost*. Z tego powodu musimy wybrać dla tych projektów różne numery portów, aby mieć możliwość uruchamiania ich jednocześnie. Użyjemy zatem numerów portów z poniższej tabeli:

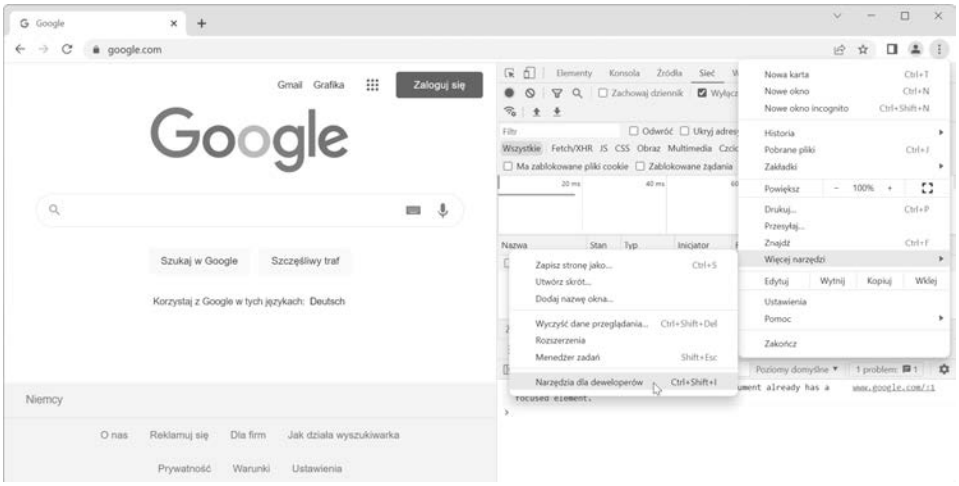
Projekt	Opis	Numery portów
Northwind.Web	Witryna ASP.NET Core Razor Pages	5000 HTTP, 5001 HTTPS
Northwind.Mvc	Witryna ASP.NET Core MVC	5000 HTTP, 5001 HTTPS
Northwind.WebApi	Serwis ASP.NET Core WebAPI	5002 HTTPS

Projekt	Opis	Numery portów
Minimal.WebApi	Serwis ASP.NET Core WebAPI (wersja minimalna)	5003 HTTPS
Northwind.BlazorServer	Aplikacja ASP.NET Core Blazor Server	5004 HTTP, 5005 HTTPS
Northwind.BlazorWasm	Aplikacja ASP.NET Core Blazor WebAssembly	5006 HTTPS, 5007 HTTPS

Używanie Google Chrome do wykonywania żądań HTTP

Sprawdźmy teraz, jak wykorzystać przeglądarkę Google Chrome do wykonywania żądań HTTP.

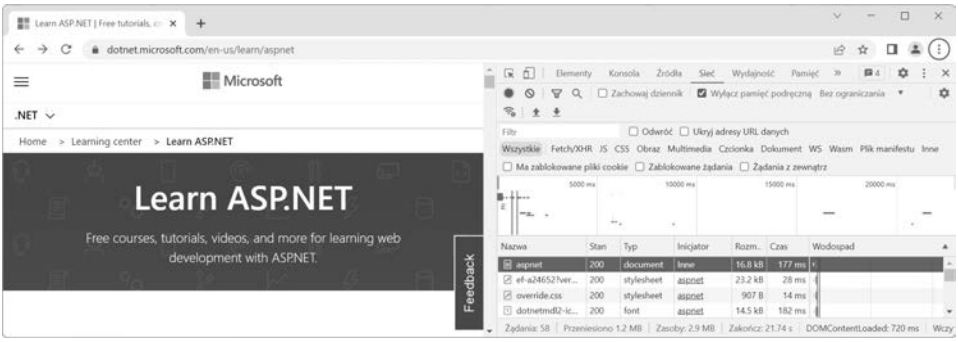
1. Uruchom Google Chrome.
2. Wybierz z menu pozycję *Więcej narzędzi/Narzędzia dla developerów*.
3. Kliknij kartę *Sieć*. Chrome powinna od razu zacząć rejestrować ruch sieciowy pomiędzy przeglądarką a dowolnym serwerem WWW (zwróć uwagę na czerwoną lampkę), tak jak na rysunku 12.4.



Rysunek 12.4. Narzędzia dla developerów w Google Chrome zapisują dane ruchu sieciowego

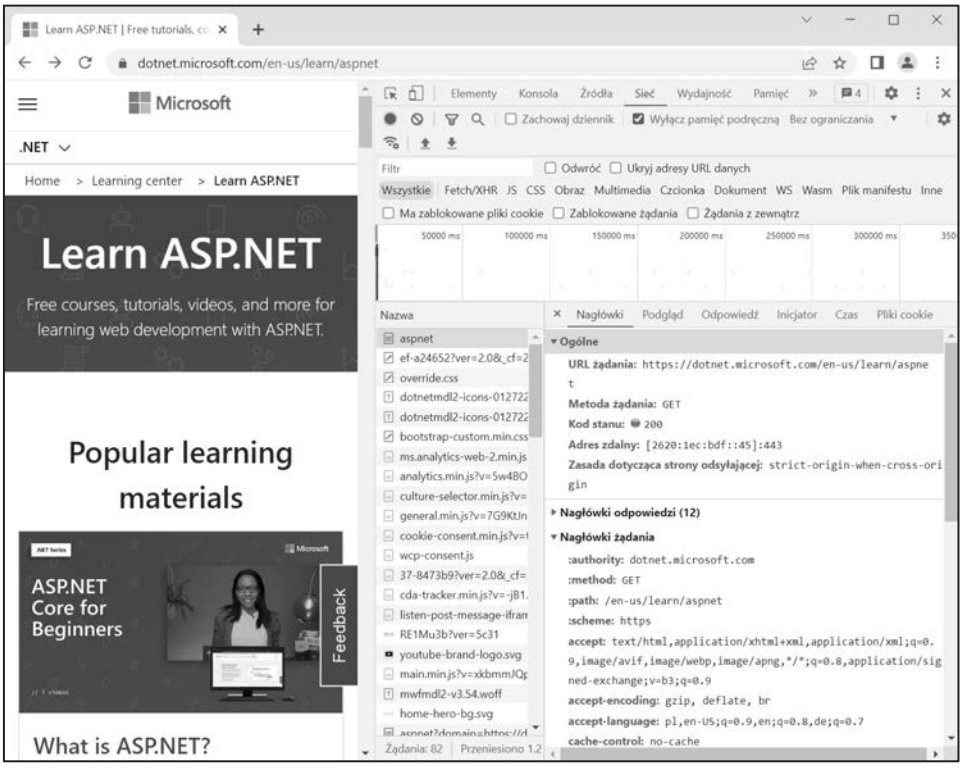
4. W pasku adresu Google Chrome wpisz adres URL witryny Microsoftu przeznaczonej do nauki korzystania z ASP.NET: *https://dotnet.microsoft.com/learn/aspnet*.

5. W okienku narzędzi dla programistów na liście zapisanych żądań kliknij pierwszą pozycję, w której w kolumnie *Typ* znajduje się wartość *dokument*, tak jak na rysunku 12.5.



Rysunek 12.5. Zapisane żądanie wyświetlone w oknie narzędzi dla developerów

6. Po prawej stronie kliknij kartę *Nagłówki*, a zobaczysz szczegóły wybranego żądania oraz odpowiedzi na nie, tak jak na rysunku 12.6.



Rysunek 12.6. Nagłówki żądania i odpowiedzi

Zwróć uwagę na następujące objaśnienia:

- **Metoda żądania** (ang. *Request Method*) to `GET`. Inne metody definiowane przez protokół HTTP to `POST`, `PUT`, `DELETE`, `HEAD` i `PATCH`.
- **Kod statusu** (ang. *Status Code*) to `200 OK`. Oznacza to, że serwer odszukał zasób żądany przez przeglądarkę i przesłał go w ciele odpowiedzi. Wśród innych kodów statusu znajdziemy też `301 Moved Permanently`, `400 Bad Request`, `401 Unauthorized` i `404 Missing`.
- Sekcja **nagłówków żądania** (ang. *Request Headers*) zawiera:
  - Nagłówek **Accept**, w której przeglądarka wymienia akceptowane formaty. W tym przypadku przeglądarka twierdzi, że może pracować z formatami HTML, XHTML, XML i pewnymi formatami obrazów, ale przyjmie też inne pliki oznaczone jako */*. Domyślną wagą dla formatów, zwaną też współczynnikiem jakości, jest wartość 1.0. Format XML otrzymał tu współczynnik jakości 0.9, co oznacza, że jest mniej pożądany od formatów HTML i XHTML. Wszystkie pozostałe typy plików otrzymały współczynnik jakości wynoszący 0.8, co oznacza, że są najmniej pożądanymi formatami danych.
  - Nagłówek **akceptowanych kodowań** (ang. *Accept-Encoding*) mówi nam, że przeglądarka przyjmie dane skompresowane algorytmami `GZIP`, `DEFLATE` lub `Brotli`.
  - Nagłówek **accept-language**, w którym przeglądarka informuje serwer, jakie języki ludzkie będzie w stanie obsłużyć. Będzie to język polski (o domyślnym współczynniku jakości wynoszącym `1.0`), ale też angielski w wariancie amerykańskim (ze współczynnikiem jakości `0.9`) albo dowolny dialekt angielskiego (ze współczynnikiem jakości `0.8`).
- Sekcja **nagłówków odpowiedzi** (ang. *Response Headers*) i nagłówek **content-encoding** mówią nam, że serwer odesłał w odpowiedzi stronę HTML skompresowaną algorytmem GZIP, ponieważ wiedział, że klient może obsłużyć ten format. (Tej informacji nie widać na rysunku 12.6, gdyż brakuje na nim miejsca do rozwinięcie sekcji *Nagłówki odpowiedzi*).

7. Zamknij przeglądarkę Google Chrome.

Tworzenie oprogramowania dla sieci WWW po stronie klienta

Podczas tworzenia aplikacji WWW programista musi znać coś więcej niż tylko język C# i platformę .NET. Po stronie klienta (czyli przeglądarki) będziemy używali kombinacji poniższych trzech elementów:

- **HTML5** — używany jest do definiowania treści i struktury strony WWW.
- **CSS3** — używany jest do definiowania stylów nakładanych na poszczególne elementy strony WWW.
- **JavaScript** — używany jest do zapisywania logiki biznesowej w ramach strony WWW, np. do sprawdzania poprawności danych wprowadzonych przez użytkownika albo do wywoływania funkcji w serwisach sieciowych w celu pobrania kolejnych danych używanych na stronie.

Mimo że HTML5, CSS3 i JavaScript są podstawowymi budulcami interfejsów użytkownika w sieci WWW, istnieje wiele bibliotek, które sprawiają, że tworzenie takich interfejsów jest znacznie bardziej produktywne. Można tu wymienić:

- **Bootstrap**, najpopularniejszy zestaw narzędzi frontendowych o otwartych źródłach.
- **SASS** i **LESS**, preprocesory stylów CSS.
- Język **TypeScript** zaprojektowany przez Microsoft, który pozwala na tworzenie bezpieczniejszego kodu.
- Biblioteki języka JavaScript: **jQuery**, **Angular**, **React** lub **Vue**.

Te technologie wyższego poziomu są na koniec przekształcane do jednej z trzech podstawowych technologii, które działają z wszystkimi nowoczesnymi przeglądarkami.

W ramach procesu kompilowania i wdrażania swoich aplikacji najprawdopodobniej użyjesz takich narzędzi jak:

- **Node.js**, czyli serwerowy framework używający języka JavaScript.
- **NPM** (ang. *Node Package Manager*) lub **Yarn**, czyli menedżery pakietów działające po stronie klienta.
- **Webpack**, czyli popularne narzędzie do wiązania modułów, kompilowania, transformowania i wiązania ze sobą plików źródłowych witryny.

Praktyka i ćwiczenia

Sprawdź swoją wiedzę i wiadomości, odpowiadając na kilka prostych pytań. Zyskaj trochę doświadczenia w zakresie tematów omawianych w tym rozdziale.

Ćwiczenie 12.1 — sprawdź swoją wiedzę

1. Jak nazywała się pierwsza technologia Microsoftu umożliwiająca tworzenie dynamicznych stron WWW działających po stronie serwera i dlaczego jest ona użyteczna również dzisiaj?
2. Jak nazywają się dwa serwery WWW firmy Microsoft?
3. Na czym polegają różnice między mikroserwisami a nanoserwisami?
4. Czym jest Blazor?
5. Która wersja ASP.NET Core jako pierwsza nie mogła działać na platformie .NET Framework?
6. Co to jest agent użytkownika?
7. Jakie znaczenie dla programistów aplikacji sieciowych ma model komunikacji HTTP bazujący na żądaniach i odpowiedziach?
8. Nazwij i opisz cztery elementy adresu URL.
9. Jakie możliwości dają nam narzędzia dla deweloperów w przeglądarce?
10. Podaj trzy główne technologie używane przy tworzeniu klienckiej strony aplikacji WWW. Jakie jest ich zadanie?

Ćwiczenie 12.2 — znasz te skrótowce?

Co oznaczają poniższe skrótowce?

1. URI
2. URL
3. WCF
4. TLD
5. API
6. SPA
7. CMS
8. Wasm
9. SASS
10. REST

Ćwiczenie 12.3 — dalsza lekura

Użyj linków udostępnionych na poniższej stronie, aby dowiedzieć się więcej na tematy poruszane w tym rozdziale:

https://github.com/markjprice/cs11dotnet7/blob/main/book-links.md#chapter-12---introducing-web-development-using-aspnet-core

Podsumowanie

W tym rozdziale:

- Przedstawiłem wprowadzenie do różnych modeli aplikacji, które można wykorzystać do tworzenia praktycznych aplikacji za pomocą języka C# i środowiska .NET.
- Przygotowaliśmy też dwie (lub cztery) biblioteki klas definiujących model encji umożliwiający pracę z bazą danych Northwind na serwerze SQL Server lub SQLite.

W kolejnych rozdziałach poznasz szczegóły tworzenia następujących projektów:

- proste strony WWW ze statycznym kodem HTML oraz z dynamicznym kodem Razor Pages;
- złożone witryny sieciowe budowane zgodnie ze wzorcem MVC (ang. *Model-View-Controller*);
- serwisy sieciowe, które można wywołać z dowolnej platformy zdolnej do generowania żądań HTTP, oraz witryny klienckie korzystające z tych serwisów;
- komponenty interfejsu użytkownika w technologii Blazor, które można hostować zarówno na serwerze, jak i w przeglądarce, a nawet wykorzystywać w aplikacjach dla urządzeń mobilnych i stacjonarnych.

C# i .NET: najlepsze środowisko do programowania w najlepszym języku!

C# jest flagowym dziełem Microsoftu. Podobnie jak wieloplatformowy framework .NET, jest rozwijany i wzbogacany, a z każdą kolejną wersją praca programisty staje się coraz efektywniejsza i bardziej satysfakcjonująca. Dzięki aktualnej wersji C# 11 i .NET 7 bez trudu zbudujesz złożone witryny internetowe czy aplikacje mobilne.

To kolejne, mocno przebudowane wydanie popularnego przewodnika, dzięki któremu zaczniesz skutecznie programować w języku C#. Nabierzesz wprawy w programowaniu zorientowanym obiektowo, pisaniu, testowaniu i debugowaniu funkcji, implementowaniu interfejsów i zarządzaniu danymi. Zobaczysz, w jaki sposób API środowiska .NET realizuje takie zadania jak monitorowanie wydajności i jej poprawianie, a także praca z systemem plików i serializacją. Wiedza zawarta w kolejnych rozdziałach pozwoli Ci tworzyć praktyczne aplikacje i serwisy z wykorzystaniem biblioteki ASP.NET Core, wzorca MVC i technologii Blazor.

Z tą książką nauczysz się:

- tworzyć własne typy w programowaniu zorientowanym obiektowo
- pisać, testować i debugować funkcje
- odczytywać dane i manipulować nimi za pomocą LINQ
- pracować z Entity Framework Core, Microsoft SQL Server i SQLite
- tworzyć usługi sieciowe i interfejsy użytkownika
- projektować aplikacje wieloplatformowe

Mark J. Price specjalizuje się w programowaniu w języku C#. Pracuje w Microsofcie, tworzy rozwiązania dla Microsoft Azure. Zdał ponad 80 egzaminów Microsoftu. Zajmuje się też dydaktyką: prowadzi szkolenia wprowadzające do usług Digital Experience Platform, wiodącego systemu CMS. Warto wspomnieć, że jego zespół przygotowywał pierwsze kursy języka C#, i to jeszcze w momencie, gdy ten był we wczesnej fazie *alfa*.

Helion
helion.pl
HELION SA
ul. Kościuszki 1c
44-100 Gliwice
tel.: 32 230 98 63
helion@helion.pl
KOD KORZYŚCI
Sięgnij po więcej!
ISBN 978-83-8322-687-3
9 788383 226873
Cena: 179,00 zł