Jimmy Engström

Platforma Blazor

Praktyczny przewodnik
Jak tworzyć interaktywne aplikacje internetowe z C# i .NET 7

Wydanie II

Przedmowa
Jeff Fritz, Principal Program Manager w Microsoft
i lider zespołu Live Video Technical Community Engagement

Helion

packt

Tytuł oryginału: Web Development with Blazor: A practical guide to start building interactive UIs with C# 11 and .NET 7, 2nd Edition

Tłumaczenie: Andrzej Watrak

ISBN: 978-83-289-0419-4

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
https://helion.pl/user/opinie/blazp2
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Pliki z przykładami omawianymi w książce można znaleźć pod adresem:
https://ftp.helion.pl/przyklady/blazp2.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

O autorze **13**

O recenzentach **14**

Przedmowa **16**

Wprowadzenie **17**

ROZDZIAŁ 1.
Witaj, Blazor! **21**

Wymagania techniczne 22
Dlaczego Blazor? 22
Poprzednicy platformy Blazor 23
Model WebAssembly 24
Wprowadzenie do platformy .NET 7 26
Wprowadzenie do platformy Blazor 27
Blazor Server 28
WebAssembly 30
Blazor WebAssembly kontra Blazor Server 33
Blazor Hybrid/.NET MAUI 34
Podsumowanie 34
Dalsza lektura 35

ROZDZIAŁ 2.
Utworzenie pierwszej aplikacji Blazor **36**

Wymagania techniczne 36
Przygotowanie środowiska programistycznego 36
Windows 37
macOS 38
Linux (macOS lub Windows) 39
Utworzenie pierwszej aplikacji Blazor 39
Szablony projektów 40
Utworzenie aplikacji Blazor Server 41
Utworzenie aplikacji Blazor WebAssembly 44

Korzystanie z wiersza poleceń 48
Utworzenie projektu 48
Struktura projektu 49
Program.cs 49
Index/_Host 52
App 56
MainLayout 57
Bootstrap 57
CSS 59
Podsumowanie 59

ROZDZIAŁ 3.
Udostępnianie kodu i zasobów 60
Wymagania techniczne 60
Utworzenie magazynu danych 60
Utworzenie nowego projektu 61
Utworzenie klas danych 61
Utworzenie interfejsu 63
Zaimplementowanie interfejsu 64
Utworzenie interfejsu API 70
Podsumowanie 71

ROZDZIAŁ 4.
Podstawowe komponenty platformy Blazor 72
Wymagania techniczne 72
Informacje o komponentach 73
Komponent Counter 73
Komponent FetchData 75
Składnia Razora 78
Bloki kodu 78
Wyrażenia niejawne 79
Wyrażenia jawne 79
Kodowanie wyrażeń 80
Dyrektywy 80
Wstrzykiwanie zależności 82
Singleton 83
Obiekt zakresowy 84
Obiekt przejściowy 84
Wstrzykiwanie usługi 84

Miejsca umieszczania kodu ... 85
Plik .razor ... 85
Klasa częściowa ... 86
Klasa pochodna ... 86
Osobny plik ... 87
Zdarzenia w cyklu życia komponentu ... 88
Metody OnInitialized i OnInitializedAsync ... 88
Metody OnParametersSet i OnParametersSetAsync ... 88
Metody OnAfterRender i OnAfterRenderAsync ... 89
Metoda ShouldRender ... 89
Parametry komponentu ... 89
Parametry kaskadowe ... 90
Utworzenie pierwszego komponentu ... 91
Utworzenie biblioteki komponentów ... 91
Wykorzystanie biblioteki komponentów ... 92
Utworzenie komponentu ... 93
Podsumowanie ... 94

ROZDZIAŁ 5.
Tworzenie zaawansowanych komponentów Blazora ... 95
Wymagania techniczne ... 95
Wiązanie danych ... 95
Wiązania jednokierunkowe ... 96
Wiązanie dwukierunkowe ... 97
Akcje i struktura EventCallback ... 98
Metoda RenderFragment ... 99
Fragment ChildContent ... 100
Wartość domyślna ... 100
Utworzenie komponentu alarmu ... 100
Nowe wbudowane komponenty ... 103
Ustawianie fokusu na elementach interfejsu graficznego ... 103
Modyfikowanie sekcji <head> ... 104
Wirtualizacja komponentów ... 107
Granice błędów ... 109
Podsumowanie ... 110

ROZDZIAŁ 6.
Tworzenie formularzy z weryfikacją danych 111
Wymagania techniczne 111
Elementy formularza 111
EditForm 112
InputBase 113
InputCheckbox 114
InputDate 114
InputNumber 114
InputSelect 114
InputText 114
InputTextArea 114
InputRadio 114
InputRadioGroup 114
InputFile 115
Implementacja weryfikacji poprawności danych 115
Komponent ValidationMessage 116
Komponent ValidationSummary 116
Niestandardowe klasy stylów weryfikacji 117
Wiązanie danych 120
Wiązanie danych w elementach HTML 120
Wiązanie danych w komponentach 121
Utworzenie interfejsu administracyjnego 121
Wyświetlanie i edytowanie kategorii 123
Wyświetlanie i edytowanie tagów 126
Wyświetlanie i edytowanie wpisów 129
Podsumowanie 138

ROZDZIAŁ 7.
Tworzenie interfejsu API 139
Wymagania techniczne 139
Utworzenie usługi 139
Implementacja dostępu do danych 140
Minimalistyczny interfejs API 141
Utworzenie kontrolerów API 142
Utworzenie klienta 146
Podsumowanie 151

ROZDZIAŁ 8.
Uwierzytelnianie i autoryzacja 152
Wymagania techniczne 153
Implementacja uwierzytelniania użytkowników 153
Konfiguracja aplikacji Blazor Server 154
Uwierzytelnianie w aplikacji Blazor Server 156
Uwierzytelnianie w aplikacji Blazor WebAssembly 159
Dostosowanie usługi Auth0 163
Uwierzytelnianie w interfejsie API 163
Konfiguracja usługi Auth0 163
Konfiguracja interfejsu API 164
Implementacja autoryzacji 165
Konfiguracja ról w usłudze Auth0 165
Implementacja ról w aplikacji Blazor Server 166
Implementacja ról w aplikacji Blazor WebAssembly 167
Podsumowanie 169

ROZDZIAŁ 9.
Udostępnianie kodu i zasobów 170
Wymagania techniczne 170
Pliki statyczne 171
Wybór platformy 171
Tworzenie stylu 172
Dodanie stylów CSS do projektu BlazorServer 172
Dodanie stylów CSS do projektu BlazorWebAssembly 173
Ulepszenie interfejsu administracyjnego 173
Ulepszenie menu 174
Dostosowanie wyglądu bloga 175
Izolacja stylów CSS 175
Podsumowanie 178

ROZDZIAŁ 10.
Interakcja z JavaScriptem 179
Wymagania techniczne 179
Dlaczego potrzebny jest kod JavaScript? 180
Odwołania z C# do JavaScriptu 180
Tryb globalny 181
Tryb izolowany 181

Odwołania z JavaScriptu do C# ... 183
Wywołanie metody statycznej ... 183
Wywołanie metody instancji ... 184
Wykorzystanie istniejącej biblioteki JavaScript ... 186
Interakcja z JavaScriptem w aplikacji Blazor WebAssembly ... 188
Wywoływanie kodu JavaScript w kodzie C# ... 189
Wywoływanie kodu C# w kodzie JavaScript ... 190
Podsumowanie ... 191

ROZDZIAŁ 11.
Zarządzanie stanem — część II ... 192
Wymagania techniczne ... 193
Przechowywanie danych po stronie serwera ... 193
Przechowywanie danych w adresie URL ... 193
Restrykcje ścieżek ... 194
Kwerendy ... 195
Mniej typowe przypadki ... 195
Przechowywanie danych w pamięci przeglądarki ... 196
Utworzenie interfejsu ... 197
Implementacja interfejsu w aplikacji Blazor Server ... 197
Implementacja interfejsu w aplikacji Blazor WebAssembly ... 199
Wspólny kod ... 200
Przechowywanie danych w usłudze kontenera stanu w pamięci ... 202
Aktualizacje na bieżąco w aplikacji Blazor Server ... 203
Aktualizacje na bieżąco w aplikacji Blazor WebAssembly ... 206
Podsumowanie ... 208

ROZDZIAŁ 12.
Debugowanie kodu ... 209
Wymagania techniczne ... 209
Wprowadzanie błędów ... 210
Debugowanie aplikacji Blazor Server ... 210
Debugowanie aplikacji Blazor WebAssembly ... 212
Debugowanie aplikacji Blazor WebAssembly w przeglądarce ... 213
Przeładowywanie kodu na gorąco ... 214
Podsumowanie ... 215

ROZDZIAŁ 13.
Testowanie 216
Wymagania techniczne 217
Co to jest bUnit? 217
Przygotowanie projektu testowego 217
Imitowanie interfejsu API 220
Tworzenie testów 223
Testowanie uwierzytelniania użytkowników 225
Testowanie JavaScriptu 226
Podsumowanie 227

ROZDZIAŁ 14.
Wdrażanie w środowisku produkcyjnym 228
Wymagania techniczne 228
Ciągłe dostarczanie oprogramowania 228
Opcje hostingu 229
Hosting Blazor Server 229
Hosting Blazor WebAssembly 230
Hosting IIS 230
Podsumowanie 230

ROZDZIAŁ 15.
Migracja i integracja istniejącej witryny 231
Wymagania techniczne 232
Web Components 232
Custom Elements 232
Komponent Blazora 233
Wykorzystanie platformy Blazor w witrynie Angular 234
Wykorzystanie platformy Blazor w witrynie React 235
Wykorzystanie platformy Blazor w witrynie MVC/Razor Pages 236
Wykorzystanie komponentów Web Components w witrynie Blazor 238
Migracja z Web Forms 239
Podsumowanie 240

ROZDZIAŁ 16.
WebAssembly od środka 241
Wymagania techniczne 241
Narzędzia kompilacyjne .NET WebAssembly 242
Kompilacja AOT 242

WebAssembly SIMD 243
Przycinanie kodu 243
Leniwe ładowanie kodu 244
Aplikacje PWA 246
Natywne zależności 246
Typowe problemy 248
Wskaźnik postępu 248
Wstępne renderowanie na serwerze 248
Wstępne ładowanie strony i zapisywanie stanu aplikacji 250
Podsumowanie 251

ROZDZIAŁ 17.
Generatory kodu źródłowego 252
Wymagania techniczne 252
Czym są generatory kodu? 252
Pierwsze kroki z generatorem 254
Projekty społeczności 256
InterfaceGenerator 256
Blazorators 256
Generatory kodu C# 257
Przykłady Roslyn SDK 257
Microsoft Learn 257
Podsumowanie 257

ROZDZIAŁ 18.
Wprowadzenie do .NET MAUI 258
Wymagania techniczne 258
Co to jest .NET MAUI? 259
Utworzenie projektu 260
Aplikacja platformy .NET MAUI 260
Biblioteka klas platformy .NET MAUI 260
Aplikacja platformy .NET MAUI Blazor 260
Szczegóły szablonu 261
Uruchamianie aplikacji w systemie Android 265
Uruchomienie aplikacji na emulatorze 265
Uruchomienie aplikacji na fizycznym urządzeniu 267
Uruchamianie aplikacji w systemie iOS 267
Gorący restart 268
Uruchamianie aplikacji w systemie macOS 272

Uruchamianie aplikacji w systemie Windows ... 273
Uruchamianie aplikacji w systemie Tizen ... 273
Podsumowanie ... 273

ROZDZIAŁ 19.
Dalsze kroki ... 274
Wymagania techniczne ... 274
Doświadczenia w uruchamianiu aplikacji Blazor w środowisku produkcyjnym ... 274
Problemy z pamięcią ... 275
Problemy z wielowątkowością ... 276
Błędy ... 276
Stare przeglądarki ... 276
Dalsze kroki ... 277
Społeczność ... 277
Komponenty ... 278
Podsumowanie ... 279

Rozdział 2

Utworzenie pierwszej aplikacji Blazor

W tym rozdziale przygotujesz środowisko programistyczne, które pozwoli Ci tworzyć aplikacje Blazor. Poznasz strukturę projektu ze szczególnym uwzględnieniem różnic między aplikacjami Blazor Server i Blazor WebAssembly. Pod koniec rozdziału będziesz posiadać skonfigurowane środowisko programistyczne i gotowe aplikacje obu rodzajów.

W tym rozdziale są opisane następujące zagadnienia:

- Przygotowanie środowiska programistycznego.
- Utworzenie pierwszej aplikacji Blazor.
- Korzystanie z wiersza poleceń.
- Struktura projektu.

Wymagania techniczne

Utworzysz nowy projekt, silnik bloga, nad którym będziesz pracować podczas lektury tej książki.

Wykorzystane w tym rozdziale kody źródłowe znajdziesz w dołączonych do książki plikach w katalogu *Chapter02*.

Przygotowanie środowiska programistycznego

W tej książce skupimy się na programowaniu w systemie Windows, a wszystkie zrzuty ekranu — o ile nie zaznaczono inaczej — będą pochodzić z programu Visual Studio 2022. Ponieważ wersja platformy .NET 7 jest wielosystemowa, dowiesz się, jak przygotować środowisko programistyczne również w systemach macOS i Linux.

Odnośniki do programów instalacyjnych wszystkich rodzajów środowisk, tj. Visual Studio 2022, Visual Studio Code i Visual Studio for MAC, znajdziesz pod adresem *https://visualstudio.microsoft.com*.

Windows

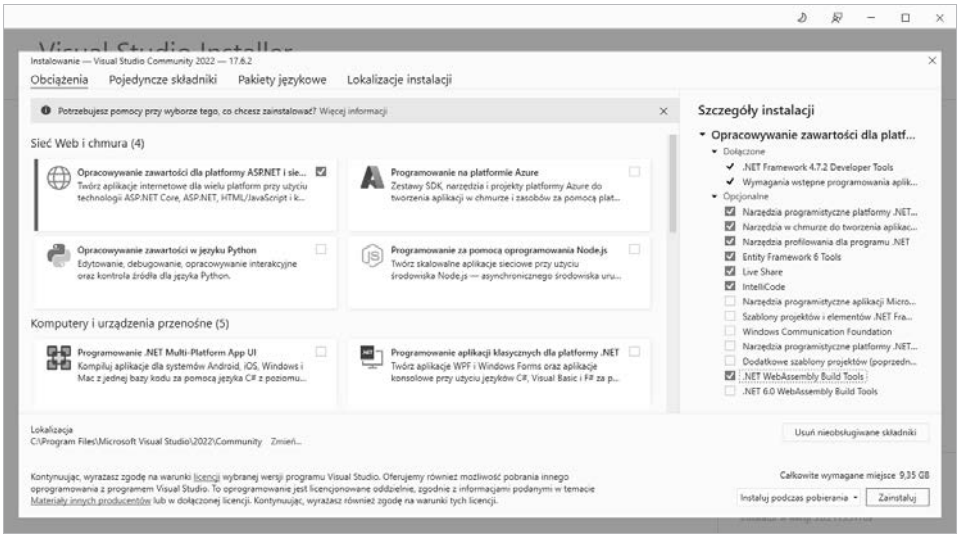
W systemie Windows jest kilka opcji tworzenia aplikacji Blazor. Najlepszym przeznaczonym do tego celu narzędziem jest oprogramowanie Visual Studio 2022, dostępne w trzech edycjach:

- Community,
- Professional,
- Enterprise.

W skrócie, edycja Community jest darmowa i ma pewne ograniczenia. Dwie pozostałe są płatne. Porównanie wszystkich edycji znajdziesz na stronie *https://visualstudio.microsoft.com/vs/compare*. W tej książce możesz używać dowolnej z nich.

Wykonaj poniższe kroki:

1. Otwórz stronę *https://visualstudio.microsoft.com/vs* i pobierz program instalacyjny odpowiedniej dla siebie edycji oprogramowania Visual Studio 2022.
2. Uruchom program instalacyjny, a następnie zaznacz opcję *Opracowywanie zawartości dla platformy ASP.NET i sieci Web*, jak na rysunku 2.1.



Rysunek 2.1. Instalacja środowiska Visual Studio 2022 w systemie Windows

3. W widocznej po prawej stronie okna liście komponentów przeznaczonych do zainstalowania zaznacz *.NET WebAssembly Build Tools* (narzędzia kompilacyjne .NET WebAssembly).

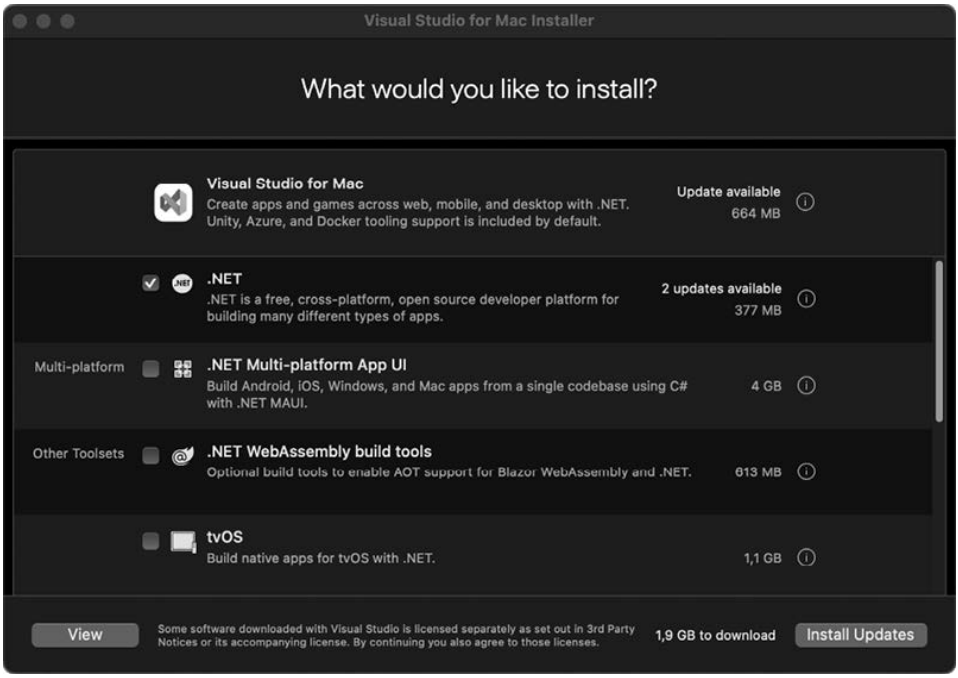
Aplikacje Blazor dla systemu Windows możesz również tworzyć przy użyciu środowiska Visual Studio Code. Nie będziemy się jednak zajmować jego instalacją.

macOS

W systemie macOS również jest kilka opcji. Najlepszym środowiskiem jest Visual Studio for MAC, dostępne na stronie *https://visualstudio.microsoft.com/vs/mac*.

Wykonaj poniższe kroki:

1. Na stronie kliknij przycisk *Pobierz*.
2. Otwórz pobrany plik.
3. Zaznacz opcję *.NET*, jak na rysunku 2.2.



Rysunek 2.2. Instalacja środowiska Visual Studio for Mac

Możesz również użyć środowiska Visual Studio Code, które też jest wielosystemowe.

Linux (macOS lub Windows)

Środowisko Visual Studio Code jest wielosystemowe. Oznacza to, że można z niego korzystać w systemach Linux, macOS i Windows. Wszystkie odmiany są dostępne na stronie *https://code.visualstudio.com/Download*.

Po zainstalowaniu środowiska dodaj do niego dwa rozszerzenia. W tym celu wykonaj poniższe kroki:

1. Otwórz środowisko i naciśnij klawisze *Ctrl+Shift+X*.
2. W polu wyszukiwania wpisz frazę **`C# Dev Kit`** i kliknij przycisk *Zainstaluj*.
3. Wyszukaj rozszerzenie *JavaScript Debugger (Nightly)* i kliknij przycisk *Zainstaluj*.

Projekty tworzy się za pomocą narzędzia .NET CLI, do którego wrócimy w dalszej części książki. Na razie nie będziemy się nim zajmować.

Teraz, gdy wszystko jest skonfigurowane, możesz utworzyć swoją pierwszą aplikację.

Utworzenie pierwszej aplikacji Blazor

Podczas pracy z tą książką będziesz tworzyć silnik bloga. Nie będziesz jednak implementować skomplikowanego algorytmu, który musiałbyś poznać. Będziesz za to korzystać z różnych technologii i rozwiązań, powszechnie stosowanych podczas tworzenia aplikacji Blazor.

Użytkownicy odwiedzający Twoją witrynę będą mogli czytać wpisy i wyszukiwać je. Witryna będzie również zawierała chronioną hasłem stronę administracyjną, na której będziesz mógł tworzyć wpisy.

Aplikację zbudujesz zarówno w modelu Blazor Server, jak i Blazor WebAssembly. Pokażę Ci kroki, jakie trzeba wykonać w obu przypadkach, i jakie są między nimi różnice.

Ważna uwaga

Od tej chwili będziemy korzystać ze środowiska Visual Studio 2022 dla systemu Windows, ale projekty w innych systemach tworzy się podobnie.

Szablony projektów

Platforma .NET 7 zawiera więcej szablonów projektów niż jej poprzednie wersje. Poznasz je dokładniej w rozdziale 4., „Podstawowe komponenty platformy Blazor". Ten rozdział zawiera tylko ich ogólny przegląd. Są to cztery szablony aplikacji Blazor Server, dwa Blazor WebAssembly i jeden Blazor Hybrid (.NET MAUI), którym zajmiemy się w rozdziale 18., „Wprowadzenie do .NET MAUI".

Aplikacja Blazor Server

Szablon *Aplikacja Blazor Server* (*Blazor Server App*) służy — jak sama nazwa wskazuje — do tworzenia aplikacji w modelu Blazor Server. Zawiera kilka komponentów, dzięki którym można dowiedzieć się, jak tego typu aplikacja funkcjonuje. Oprócz tego szablon obejmuje podstawowe ustawienia, menu oraz kod wykorzystujący bibliotekę Bootstrap, izolację stylów CSS itp. (więcej na ten temat dowiesz się w rozdziale 9., „Udostępnianie kodu i zasobów"). Użyjesz tego szablonu, aby dowiedzieć się dokładniej, jak wygląda aplikacja Blazor Server.

Aplikacja zestawu WebAssembly platformy Blazor

Szablon *Aplikacja zestawu WebAssembly platformy Blazor* (*Blazor WebAssembly App*) służy — jak sama nazwa wskazuje — do tworzenia aplikacji w modelu Blazor WebAssembly. Podobnie jak szablon *Aplikacja Blazor Server*, zawiera kilka komponentów, dzięki którym można dowiedzieć się, jak tego typu aplikacja funkcjonuje. Oprócz tego szablon obejmuje podstawowe ustawienia, menu oraz kod wykorzystujący bibliotekę Bootstrap, izolację stylów CSS itp. Użyjesz tego szablonu, aby dowiedzieć się dokładniej, jak wygląda aplikacja Blazor WebAssembly.

Pusta aplikacja modelu hostowania Blazor Server

Pusta aplikacja modelu hostowania Blazor Server (*Blazor Server App Empty*) to prosty szablon zawierający komponenty niezbędne do uruchomienia aplikacji w modelu Blazor Server. Nie wykorzystuje izolacji stylów CSS ani innych funkcjonalności. Tego szablonu zazwyczaj używa się do tworzenia projektów produkcyjnych. Trzeba jednak samodzielnie implementować funkcjonalności, które są standardowo zawarte w innych przykładowych szablonach.

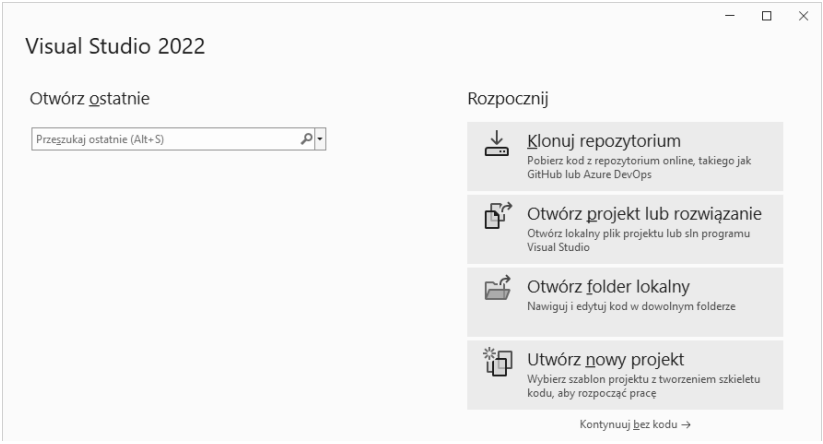
Pusta aplikacja zestawu WebAssembly platformy Blazor

Pusta aplikacja modelu hostowania Blazor Server (*Blazor WebAssembly App Empty*) to prosty szablon zawierający komponenty niezbędne do uruchomienia aplikacji Blazor WebAssembly. Nie wykorzystuje izolacji stylów CSS ani innych funkcjonalności. Tego szablonu zazwyczaj używa się, tworząc projekt z prawdziwego zdarzenia. Programista musi jednak zaimplementować funkcjonalności, które zawierają inne przykładowe szablony.

Utworzenie aplikacji Blazor Server

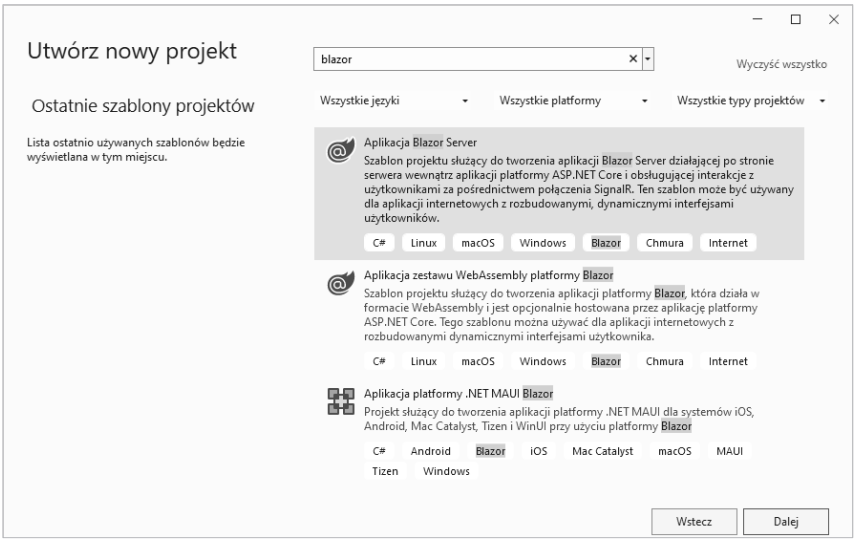
Teraz utwórz aplikację Blazor Server i zapoznaj się z nią. Wykonaj następujące kroki:

1. Uruchom Visual Studio 2022. Pojawi się okno jak na rysunku 2.3.



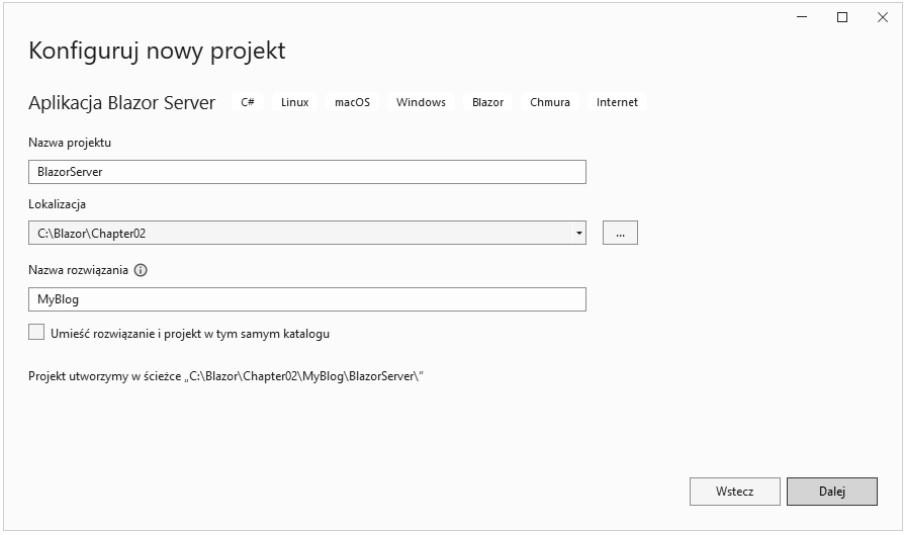
Rysunek 2.3. Okno startowe środowiska Visual Studio 2022

2. Kliknij przycisk *Utwórz nowy projekt*, a następnie w polu wyszukiwania wpisz **`blazor`**.
3. W liście wyników wyszukiwania zaznacz pozycję *Aplikacja Blazor Server* i kliknij przycisk *Dalej*, jak na rysunku 2.4.



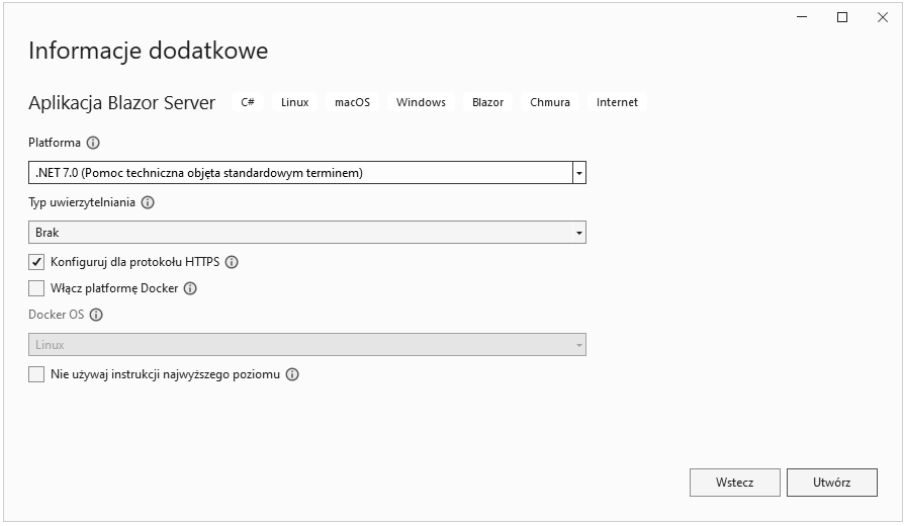
Rysunek 2.4. Tworzenie nowego projektu Blazor Server

4. Wpisz nazwę projektu. To najtrudniejsze zadanie w każdym projekcie, ale nie martw się, zrobiłem to za Ciebie. Wpisz **`BlazorServer`**, nazwę rozwiązania zmień na *MyBlog* i kliknij przycisk *Dalej*, jak na rysunku 2.5.



Rysunek 2.5. Konfigurowanie pierwszego projektu

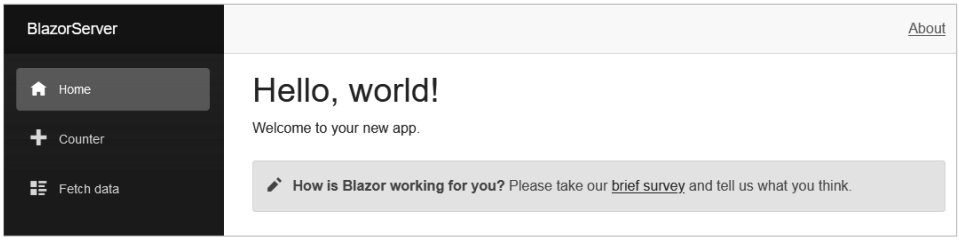
5. W następnym oknie wybierz rodzaj aplikacji Blazor. Z rozwijanej listy wybierz opcję *.NET 7.0 (Pomoc techniczna objęta standardowym terminem)* i kliknij przycisk *Utwórz*, jak na rysunku 2.6.



Rysunek 2.6. Utworzenie projektu Blazor Server

6. Uruchom aplikację, używając klawiszy *Ctrl+F5* lub polecenia menu *Debuguj/Uruchom bez debugowania*.

Gratulacje! Właśnie utworzyłeś swoją pierwszą aplikację Blazor Server. Powinna wyglądać jak na rysunku 2.7.

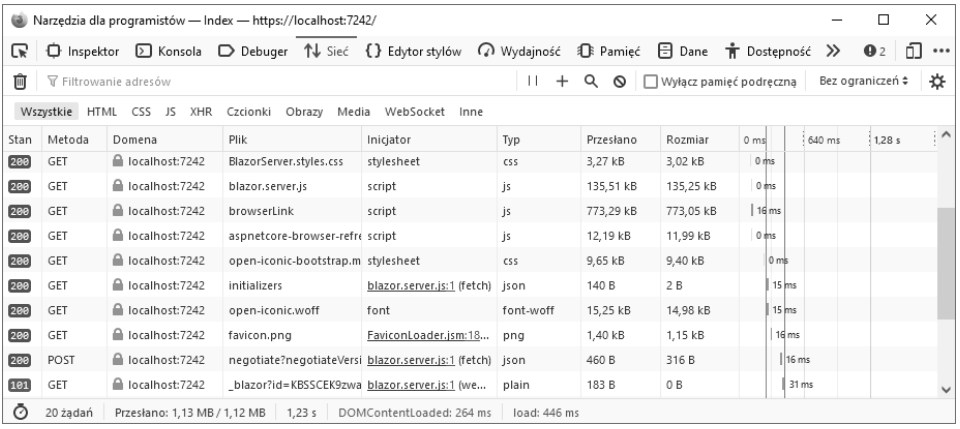


Rysunek 2.7. Pierwsza aplikacja Blazor Server

Zapoznaj się ogólnie z aplikacją, kliknij odnośniki *Counter* (licznik) i *Fetch data* (pobierz dane). Zwróć uwagę na szybkość ładowania stron i wykonywane operacje. Aplikacja zawiera przykładowe dane na potrzeby testów.

Jest to projekt Blazor Server, co oznacza, że każde zdarzenie, na przykład kliknięcie przycisku, powoduje wysłanie do serwera polecenia przy użyciu technologii SignalR. Serwer wtedy ponownie renderuje komponent i wysyła zmiany do klienta, który aktualizuje interfejs graficzny aplikacji.

Naciśnij przycisk *F12* lub otwórz za pomocą menu narzędzia dla deweloperów. Kliknij zakładkę *Sieć* i naciśnij klawisz *F5*, aby przeładować stronę. Pojawi się lista plików pobranych przez przeglądarkę, jak na rysunku 2.8.



Rysunek 2.8. Lista plików pobranych przez aplikację Blazor Server

Przeglądarka pobiera plik strony, style CSS i skrypt *blazor.server.js*, który odwołuje się do punktu końcowego *negotiate* w celu nawiązania połączenia SignalR z serwerem.

Ścieżka *_blazor?id=* wraz z serią liter to wywołanie WebSocket i nawiązanie połączenia między klientem a serwerem.

Kliknij odnośnik *Counter*, a następnie przycisk *Click me* (kliknij mnie). Zwróć uwagę, że strona nie przeładuje się. Zdarzenie (kliknięcie) jest za pomocą technologii SignalR wysyłane do serwera, który ponownie renderuje stronę, porównuje ją z drzewem renderowania i za pomocą protokołu WebSocket wysyła do przeglądarki jedynie zmiany.

Gdy klikniesz przycisk, wykonywane są trzy operacje:

- przeglądarka generuje zdarzenie (kliknięcie przycisku),
- serwer wysyła zmiany,
- przeglądarka potwierdza aktualizację modelu DOM (ang. *Document Object Model*, obiektowy model dokumentu).

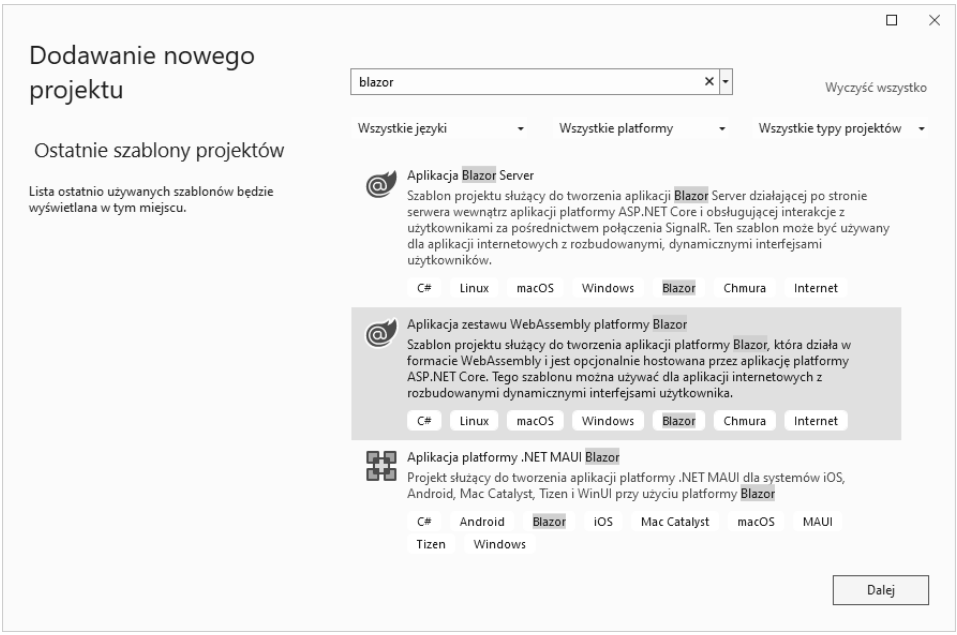
W sumie po kliknięciu przycisku na stronie *Counter* w obie strony jest przesyłanych ok. 600 bajtów danych.

Utworzyłeś i przetestowałeś rozwiązanie i projekt Blazor Server. Teraz dodaj do rozwiązania aplikację Blazor WebAssembly.

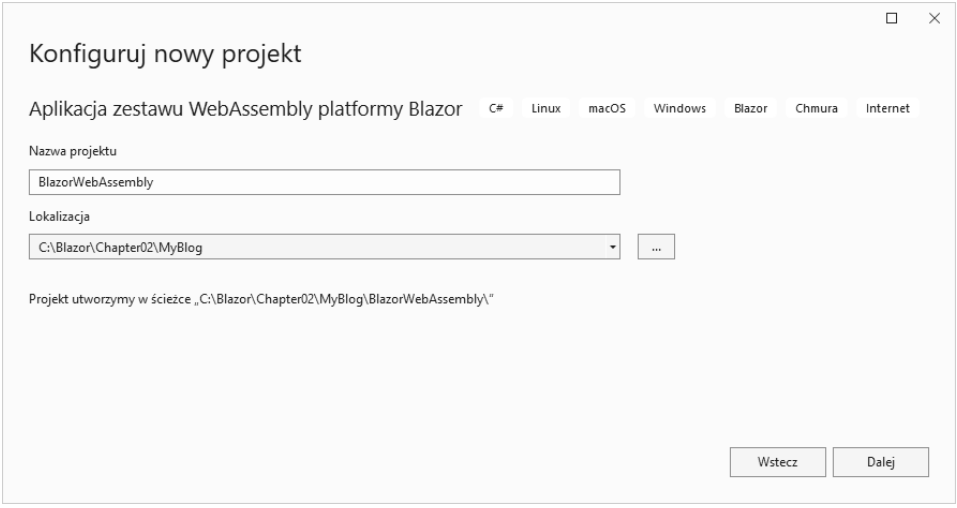
Utworzenie aplikacji Blazor WebAssembly

Pora przyjrzeć się aplikacji Blazor WebAssembly. Utwórz ją w tym samym rozwiązaniu co Blazor Server:

1. Kliknij prawym przyciskiem myszy rozwiązanie *MyBlog*, a następnie polecenie *Dodaj/Nowy projekt*.
2. W polu wyszukiwania wpisz **`blazor`**, w liście wyników kliknij *Aplikacja zestawu WebAssembly platformy Blazor*, a następnie przycisk *Dalej*, jak na rysunku 2.9.
3. Nadaj projektowi nazwę *BlazorWebAssembly*, lokalizację pozostaw bez zmian (środowisko Visual Studio 2022 domyślnie umieści projekt w tym samym katalogu co poprzednio) i kliknij przycisk *Dalej*, jak na rysunku 2.10.
4. W następnym oknie wybierz z rozwijanej listy opcję *.NET 7.0 (Pomoc techniczna objęta standardowym terminem)*.
5. W tym oknie są dwie opcje, których nie było podczas tworzenia aplikacji Blazor Server. Pierwsza to *ASP.NET Core Hosted* (hosting ASP.NET Core). Jej zaznaczenie powoduje utworzenie backendu ASP.NET, w którym będzie hostowana aplikacja Blazor WebAssembly. Ta opcja jest wymagana, jeżeli aplikacja ma udostępniać interfejsy API.

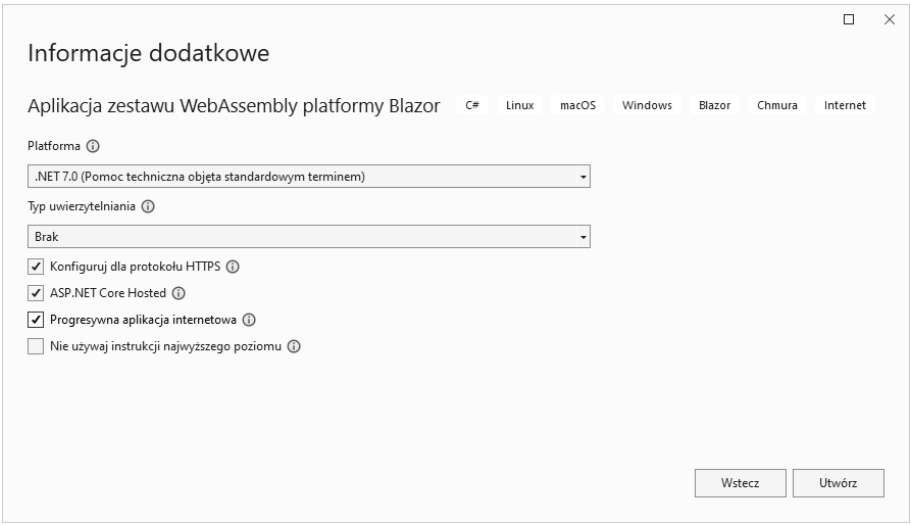


Rysunek 2.9. Tworzenie nowego projektu Blazor WebAssembly



Rysunek 2.10. Konfigurowanie kolejnego projektu

6. Druga opcja to *Progresywna aplikacja internetowa*. Jej zaznaczenie powoduje utworzenie plików *manifest.json* i *service-worker.js* niezbędnych w aplikacji PWA. Dlatego zaznacz ją, a następnie kliknij przycisk *Utwórz*, jak na rysunku 2.11.



Rysunek 2.11. Utworzenie projektu Blazor WebAssembly

7. Kliknij prawym przyciskiem myszy projekt *BlazorWebAssembly.Server*. a następnie polecenie *Ustaw jako projekt startowy*.

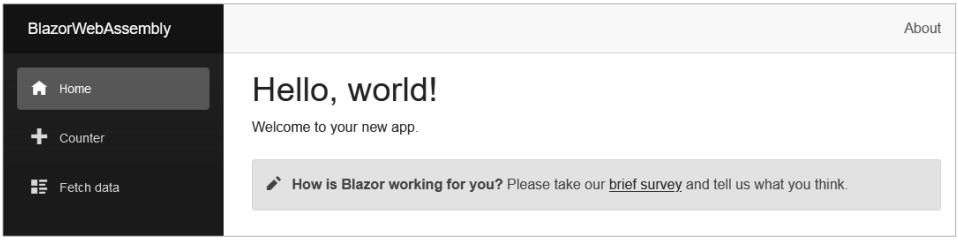
Uwaga

Nazwa projektu zawiera słowo *Server*, które może być mylące. Ponieważ podczas tworzenia projektu zaznaczyłeś opcję *ASP.NET Core Hosted*, został utworzony backend *BlazorWebAssembly.Server* dla aplikacji Blazor WebAssembly. Nie ma on jednak nic wspólnego z projektem Blazor Server.

Pamiętaj, że w celu uruchomienia aplikacji Blazor WebAssembly należy uruchomić projekt *BlazorWebAssembly.Server*, czyli backend ASP.NET Core.

8. Uruchom aplikację, używając klawiszy *Ctrl+F5* lub polecenia menu *Debuguj/Uruchom bez debugowania*.

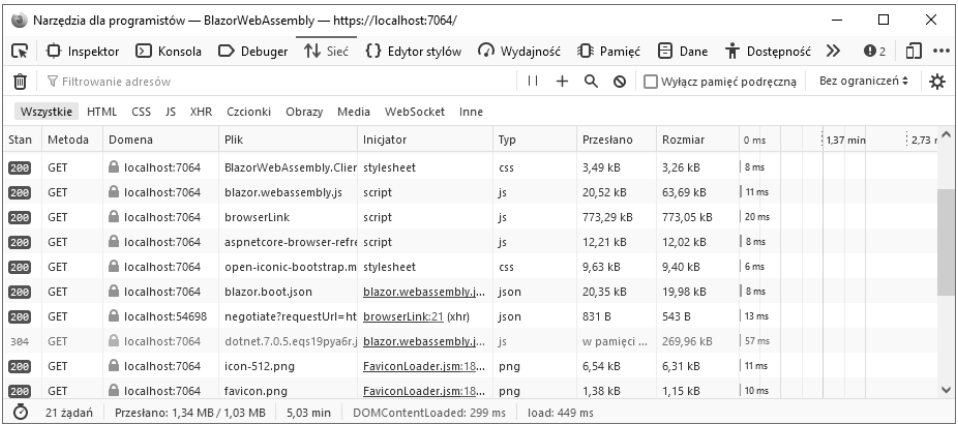
Gratulacje! Utworzyłeś swoją pierwszą aplikację Blazor WebAssembly. Powinna wyglądać jak na rysunku 2.12.



Rysunek 2.12. Pierwsza aplikacja Blazor WebAssembly

Kliknij odnośniki *Counter* i *Fetch data*, aby zapoznać się z aplikacją. Powinna prezentować się tak samo jak Blazor Server.

Naciśnij przycisk *F12* lub otwórz za pomocą menu narzędzia dla deweloperów. Kliknij zakładkę *Sieć* i naciśnij klawisz *F5*, aby przeładować stronę. Pojawi się lista plików pobranych przez przeglądarkę, jak na rysunku 2.13.



Rysunek 2.13. Lista plików pobranych przez aplikację Blazor WebAssembly

Tym razem przeglądarka pobrała najpierw plik *blazor.webassembly.js*, a następnie *blazor.boot.json*, którego fragment przedstawia rysunek 2.14.


```
{
  "cacheBootResources": true,
  "config": [ ],
  "debugBuild": true,
  "entryAssembly": "BlazorWebAssembly.Client",
  "icuDataMode": 0,
  "linkerEnabled": false,
  "resources": {
    "assembly": {
      "Microsoft.AspNetCore.Authorization.dll": "sha256-\/EkWRHMHBhfvvlx5hLVtXGPm8+Gyp4WkPiKKiiJIrqQ=",
      "Microsoft.AspNetCore.Components.dll": "sha256-SvjxqyTqc68zPsYivPRkvjKRhUg5zYufzqwGNS83bwg=",
      "Microsoft.AspNetCore.Components.Forms.dll": "sha256-eoQ2CgfBGkvbK4QUMBgFDPIBqJem+g8247Fb3ZmnmCI=",
      "Microsoft.AspNetCore.Components.Web.dll": "sha256-p3BQFPwUbOgu5j9V+1HlUGwDcVzB7iYNO4dIsDCOAgA=",
      "Microsoft.AspNetCore.Components.WebAssembly.dll": "sha256-55+LOYcA1yHHPQEjl6eI0UGgteq41y+8js6DQ5PhXDk=",
      "Microsoft.AspNetCore.Metadata.dll": "sha256-eIu14RWCV9bKCAARda4hnWGsOKJfmnGfQpkmeRnSibQ=",
      "Microsoft.Extensions.Configuration.dll": "sha256-PqQvp77oZ4+uuy2ElXk8AU9I6RfZ8f18UGTrd4rulOo=",
      "Microsoft.Extensions.Configuration.Abstractions.dll": "sha256-CnS3b9EMFQmETBUVEgtcron4DBsfFdcVt3zfCP6Uflg=",
      "Microsoft.Extensions.Configuration.Binder.dll": "sha256-7GY9+7Cm0Z8DJ+JCFwgCx9OfFtUvXlFHaaLlBYmXfIE=",
      "Microsoft.Extensions.Configuration.FileExtensions.dll": "sha256-S86mGNxJnkVJ\/qolp6cBN7xwXQ\/YVtHy7QTaPO93AIA=",
      "Microsoft.Extensions.Configuration.Json.dll": "sha256-k525Vc8hbMpPjxYUYZNPuzJIuy+E1Is2XRTMFbUmlpE=",
      "Microsoft.Extensions.DependencyInjection.dll": "sha256-\/+vk9BsQP4bCVt1Y6aXakSztSMAli200ER6untxHLBg=",
      "Microsoft.Extensions.DependencyInjection.Abstractions.dll": "sha256-jrAm+30mcWoI54hsUTOr+RMOzHIq+zO8ZuRBVrBvCoo=",
      "Microsoft.Extensions.FileProviders.Abstractions.dll": "sha256-Zt6OY6gg\/1Tzt9oFOQBkezPvUVkFK4dyM6Pfk+MTUvg=",
```


Rysunek 2.14. Fragment pliku blazor.boot.json

Najważniejszą częścią pliku *blazor.boot.json* jest klucz `entryAssembly` zawierający nazwę biblioteki DLL, którą przeglądarka ma uruchomić. Oprócz tego w pliku tym znajduje się lista wszystkich bibliotek platformy, niezbędnych do uruchomienia aplikacji.

Przeglądarka pobiera skrypt *dotnet.7.0.*.js*, który z kolei pobiera wszystkie zasoby wymienione w pliku *blazor.boot.json*, m.in. zawierającą skompilowany kod bibliotekę .NET Standard DLL, kod platformy Microsoft .NET Framework oraz inne, zewnętrzne biblioteki DLL. Następnie pobierany jest plik *dotnet.wasm*, czyli skompilowane w modelu WebAssembly środowisko Mono, w którym jest uruchamiana aplikacja.

Jeżeli przeładujesz stronę i przyjrzysz się jej uważnie, zauważysz napis *Loading* (ładowanie). Oznacza on, że przeglądarka pobiera pliki JSON, JavaScript, WebAssembly i DLL, a następnie uruchamia aplikację. Ta operacja w trybie diagnostycznym i niezoptymalizowanym zajmuje przeglądarce ok. 1,8 s.

W tym momencie masz bazowe rozwiązanie zawierające projekty Blazor Server i Blazor WebAssembly.

W tej książce będziemy używać środowiska Visual Studio 2022, ale są też inne sposoby uruchamiania aplikacji Blazor, m.in. za pomocą wiersza poleceń, który jest niezwykle przydatnym narzędziem. W następnym podrozdziale dowiesz się, jak go użyć w celu utworzenia nowego projektu.

Korzystanie z wiersza poleceń

Wersje platformy .NET 5 i nowsze zawierają niezwykle użyteczne narzędzie *dotnet.exe*. Programiści, którzy korzystają z platformy .NET Core, już je znają, ale począwszy od wersji .NET 5 jest ono przeznaczone nie tylko dla nich.

Za pomocą tego narzędzia można wykonywać wiele takich samych operacji jak w środowisku Visual Studio 2022, m.in. tworzyć projekty, dodawać i budować pakiety NuGet. W następnym przykładzie utworzysz w ten sposób projekty Blazor Server i Blazor WebAssembly.

Utworzenie projektu

Opisany niżej przykład demonstruje siłę narzędzia *dotnet.exe*. Nie będziemy go używać w dalszej części książki, więc jeżeli nie chcesz go wypróbować, możesz pominąć ten podrozdział.

Aby utworzyć projekty Blazor Server i Blazor WebAssembly, takie jak wcześniej, wpisz następujące polecenia:

```
dotnet new blazorserver -o BlazorServer
dotnet new blazorwasm -o BlazorWebAssembly --pwa --hosted
```

W tym przykładzie `dotnet` jest poleceniem, `new` jego argumentem, `blazorserver` nazwą szablonu, a `-o` argumentem wskazującym katalog wyjściowy, w którym będzie umieszczony projekt (w tym przypadku *BlazorServer*).

W drugim poleceniu jest użyty argument `blazorwasm` oznaczający szablon Blazor WebAssembly. Argumenty `--pwa` i `--hosted` powodują utworzenie, odpowiednio, aplikacji PWA i backendu ASP.NET hostującego aplikację.

Oprócz tego musisz utworzyć rozwiązanie dla obu projektów. W tym celu użyj szablonu `sln`:

```
dotnet new sln --name MyBlog
```

Do rozwiązania dodaj utworzony wcześniej projekt Blazor Server i trzy projekty Blazor WebAssembly:

```
dotnet sln MyBlog.sln add
./BlazorWebAssembly\Server\BlazorWebAssembly.Server.csproj
dotnet sln MyBlog.sln add
./BlazorWebAssembly\Client\BlazorWebAssembly.Client.csproj
dotnet sln MyBlog.sln add
.\BlazorWebAssembly\Shared\BlazorWebAssembly.Shared.csproj
dotnet sln MyBlog.sln add .\BlazorServer\BlazorServer.csproj
```

Polecenie `dotnet` jest niezwykle przydatne i w niektórych sytuacjach warto je stosować. Prawdopodobnie będziesz głównie korzystać z graficznego środowiska Visual Studio 2022, ale warto pamiętać, że takie narzędzie istnieje.

Uwaga na temat wiersza poleceń

Z założenia narzędzie *dotnet.exe* ma umożliwiać wykonywanie w wierszu poleceń wszystkich operacji. Jeżeli preferujesz takie podejście, więcej informacji znajdziesz na stronie *https://docs.microsoft.com/pl-pl/dotnet/core/tools*.

Wróćmy do szablonów Blazora, które utworzyły mnóstwo plików. W następnym podrozdziale dowiesz się, co wygenerowało środowisko Visual Studio 2022.

Struktura projektu

Przyjrzyjmy się teraz plikom, które zostały utworzone w podrozdziale „Utworzenie pierwszej aplikacji Blazor" w obu projektach, różnicom między nimi oraz kodom.

Program.cs

Plik *Program.cs* zawiera kod, który jest uruchamiany w pierwszej kolejności. Jest on inny w projektach Blazor Server i Blazor WebAssembly.

WebAssembly Program.cs

Projekt *BlazorWebAssembly.Client* zawiera następujący plik *Program.cs*:

```
var builder = WebAssemblyHostBuilder.CreateDefault(args);
builder.RootComponents.Add<App>("#app");
builder.RootComponents.Add<HeadOutlet>("head::after");
builder.Services.AddScoped(sp => new HttpClient { BaseAddress =
↪new Uri(builder.HostEnvironment.BaseAddress) });
await builder.Build().RunAsync();
```

Plik zawiera instrukcje najwyższego poziomu, bez żadnych klas ani metod. Oznacza to, że ten kod stanowi punkt wejścia do aplikacji. Wyszukuje element `div` z identyfikatorem `"#app"`, umieszcza w nim (renderuje) komponent `App`, a w nim z kolei stronę aplikacji. Do tego komponentu wrócimy w dalszej części rozdziału.

Następnie kod tworzy komponent `HeadOutlet`, który służy do modyfikowania elementu `head`. Element ten zawiera m.in. elementy `title` i `meta`. Za pomocą komponentu `HeadOutlet` można modyfikować tytuł strony oraz elementy `meta`.

Kod tworzy również komponent `HttpClient` jako zależność zakresową. Wstrzykiwaniem zależności zajmiemy się w rozdziale 3., „Zarządzanie stanem — część 1”. Na razie wiedz, że jest to sposób tworzenia abstrakcyjnych obiektów i typów, dzięki którym nie trzeba tworzyć obiektów wewnątrz strony. Obiekty są przekazywane do stron/klas, co ułatwia testowanie aplikacji, a klasy nie mają żadnych nieznanych zależności.

W przeglądarce jest uruchamiana aplikacja WebAssembly, która może uzyskiwać dane tylko za pomocą zewnętrznych połączeń, na przykład z serwerem. Dlatego potrzebny jest komponent `HttpClient`. Ponieważ aplikacja nie może bezpośrednio pobierać danych, powyższy komponent jest zaimplementowany w szczególny sposób, z wykorzystaniem interakcji JavaScriptu.

Jak wspomniałem wcześniej, aplikacja WebAssembly działa w izolowanym środowisku i komunikuje się z zewnętrznym otoczeniem za pomocą odpowiednich interfejsów JavaScriptu przeglądarki.

Blazor Server Program.cs

Projekt Blazor Server różni się nieco od Blazor WebAssembly, ale implementuje bardzo podobne operacje. Jego plik *Program.cs* zawiera następujący kod:

```
var builder = WebApplication.CreateBuilder(args);
builder.Services.AddRazorPages();
builder.Services.AddServerSideBlazor();
builder.Services.AddSingleton<WeatherForecastService>();
var app = builder.Build();

if (!app.Environment.IsDevelopment())
```

```
{
  app.UseExceptionHandler("/Error");
  app.UseHsts();
}

app.UseHttpsRedirection();
app.UseStaticFiles();
app.UseRouting();
app.MapBlazorHub();
app.MapFallbackToPage("/_Host");
app.Run();
```

Począwszy od wersji platformy .NET 6 nie ma pliku *Startup.cs*, a cały kod startowy jest zawarty w pliku *Program.cs*.

W tym miejscu warto wspomnieć o kilku kwestiach. Najpierw są dodawane wszystkie zależności, których wymaga aplikacja. W tym przypadku są to strony Razor (pliki *.cshtml*) aplikacji Blazor. Następnie jest tworzony komponent `ServerSideBlazor`, który daje dostęp do wszystkich obiektów potrzebnych do uruchomienia aplikacji Blazor Server. Tworzony jest też komponent `WeatherForcastService` wykorzystywany po otwarciu strony *Fetch data*.

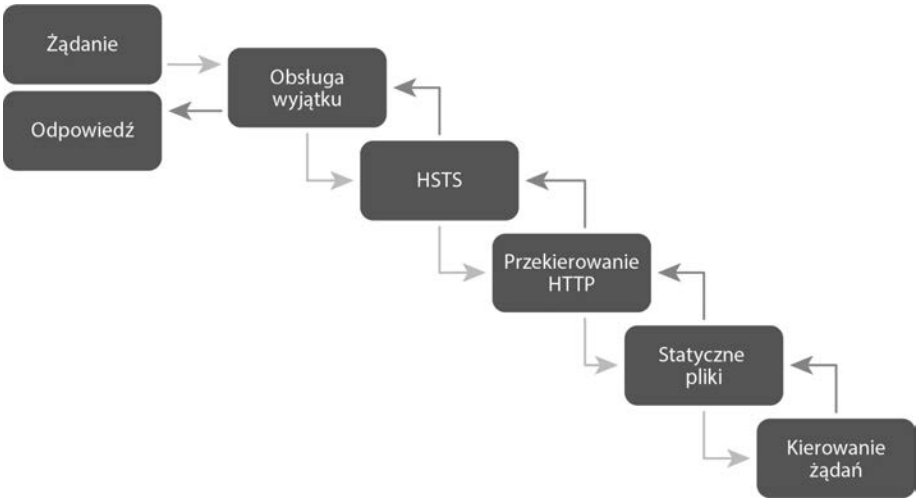
Dalej jest konfigurowany komponent **HSTS** (ang. *HTTP Strict Transport Security*, ścisłe zabezpieczenie transmisji HTTP), wykorzystujący protokół HTTPS, aby zasoby były przesyłane w zaszyfrowanej formie. Komponent ten wymusza również przejście z protokołu HTTP na HTTPS.

Metoda `UseStaticFiles` pobiera statyczne zasoby, m.in. pliki stylów CSS i obrazów. Metody o nazwach rozpoczynających się od `Use` dodają delegatów do potoku żądania lub potoku modułu pośredniczącego. Delegaci żądań (`UseExceptionHandler`, `UseHttpsRedirection`, `UseStaticFiles` itd.) są wywoływani kolejno od góry do dołu i z powrotem. Dlatego na początku jest umieszczona metoda obsługująca wyjątki. Jeżeli w którymkolwiek z delegatów żądań wystąpi wyjątek, metoda będzie mogła go obsłużyć, ponieważ żądanie przejdzie z powrotem przez potok. Ilustruje to rysunek 2.15.

Jeżeli jeden z delegatów obsłuży żądanie, na przykład udostępnienia statycznego pliku, pozostali delegaci, m.in. kierujący żądaniami, nie są wywoływani. Nie trzeba przecież kierować żądania udostępnienia statycznego pliku. Ważne jest zatem umieszczenie delegatów w odpowiedniej kolejności.

Uwaga

Więcej szczegółowych informacji na ten temat znajdziesz na stronie *https://docs.microsoft.com/pl-pl/aspnet/core/fundamentals/middleware/?view=aspnetcore-7.0*.



Rysunek 2.15. Potok żądań

Na końcu kodu są tworzone punkty końcowe, m.in. dla huba Blazor SignalR. Jeżeli pojawi się żądanie otwarcia nieistniejącej strony, uruchamiany zostaje kod zawarty w pliku *_host*, a żądanie jest kierowane do głównej strony aplikacji.

Index/_Host

W dalszej części pliku jest uruchamiany kod zawarty w pliku *Index* lub *_host*. Pliki te zawierają informacje niezbędne do załadowania potrzebnego kodu JavaScript.

_Host (Blazor Server)

W projekcie Blazor Server, w katalogu *Pages*, znajduje się plik *_Host.cshtml*. Jest to strona Razora, która nie jest tym samym co komponent Razora:

- **Strona Razora** zawiera kod widoku. Nie jest to komponent, który może być częścią strony lub innego komponentu.
- **Komponent Razora** reprezentuje współdzielony widok wykorzystywany w aplikacji. Na przykład można utworzyć reprezentujący tabelę komponent `Grid` i stosować go w aplikacji albo umieścić w bibliotece, aby mogli z niego korzystać inni programiści. Komponent może również reprezentować całą stronę. W takim wypadku poprzedza się go znakiem `@`. Więcej na ten temat dowiesz się w dalszej części rozdziału.

W większości przypadków projekt Blazor Server zawiera tylko jeden plik *.cshtml* strony. W pozostałych plikach znajdują się komponenty Razora.

Na początku strony są umieszczone dyrektywy rozpoczynające się od znaku `@`, np. `@page`, `@namespace`, `@addTagHelper`, jak niżej:

```
@page "/"
@using Microsoft.AspNetCore.Components.Web
@namespace BlazorServer.Pages
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers

<!DOCTYPE html>
<html lang="en">
<head>
    ...
    <component type="typeof(HeadOutlet)" render-mode="ServerPrerendered" />
</head>
<body>
    <component type="typeof(App)" render-mode="ServerPrerendered" />
```

Zwróć uwagę na kilka cech tego pliku. Na początku znajdują się dyrektywy `@`, które definiują adres URL strony, przestrzeń nazw, pomocnicze znaczniki i stronę `Layout`. Dyrektywy zostaną opisane w rozdziale 4., „Podstawowe komponenty platformy Blazor".

Dalej mamy znacznik `component`:

```
<component type="typeof(App)" render-mode="ServerPrerendered" />
```

W tym miejscu jest renderowana strona aplikacji. Zajmuje się tym komponent `App`. W ten sposób można również za pomocą znaczników pomocniczych dodawać komponenty do istniejącej aplikacji, która nie jest oparta na platformie Blazor.

Komponent `App` może być renderowany w jednym z pięciu trybów:

- `ServerPrerendered` (domyślny): treść jest w całości renderowana na serwerze i wysyłana do przeglądarki przy pierwszym otwarciu strony. Wykorzystywany jest przy tym hub Blazor SignalR, który wysyła zmiany do serwera. Serwer renderuje zmienione komponenty i odsyła je, wykorzystując technologię SignalR. Użytkownik zazwyczaj tego nie widzi. Jeżeli jednak na serwerze są wykorzystywane pewne zdarzenia, mogą być zgłaszane dwukrotnie i wykonywać niepotrzebne operacje, na przykład odwoływać się do bazy danych.
- `Server`: w tym trybie serwer wysyła do przeglądarki całą stronę zawierającą symbole zastępcze komponentów. Następnie, gdy wykona jakąś operację, na przykład pobierze dane z bazy, wysyła zmiany za pośrednictwem huba Blazor SignalR.
- `Static`: komponent jest renderowany, po czym następuje przerwanie połączenia. Oznacza to, że komponent nie reaguje na zdarzenia i nie można go modyfikować. Jest to właściwy tryb dla statycznych danych.
- `WebAssembly`: renderowany jest marker aplikacji WebAssembly, a nie komponent.

- `WebAssemblyPrerendered`: komponent jest renderowany do statycznego kodu HTML i uruchamiana jest aplikacja WebAssembly. Ten tryb wykorzystamy w rozdziale 5., „Tworzenie zaawansowanych komponentów Blazora".

Strona najszybciej pojawia się w przeglądarce, gdy jest stosowany tryb `ServerPrerendered`. Jest to dobry wybór, jeżeli strona ma się ładować szybko. Jeżeli natomiast strona ma się szybko wyświetlać i ładować dane, które serwer pobiera z bazy, lepszy jest moim zdaniem tryb `Server`. Preferuję ten tryb, ponieważ postrzegany czas ładowania strony powinien być krótki. Zmiana trybu na `Server` to pierwsza rzecz, którą robię, tworząc nową aplikację Blazor Server. Lepszym wyjściem jest moim zdaniem, aby dane na stronie pojawiały się kilka milisekund później, a użytkownik miał wrażenie, że cała strona ładuje się szybko.

W okolicach początku kodu znajduje się znacznik `base`:

```
<base href="~/" />
```

Znacznik ten wskazuje główny katalog aplikacji Blazor. Gdyby go nie było, aplikacja nie miałaby dostępu do statycznych zasobów, takich jak obrazy, skrypty JavaScript czy style CSS.

W dalszej części kodu ładowane są style CSS: domyślnie dwa statyczne, jeden Bootstrap i jeden właściwy dla danej strony:

```
<link rel="stylesheet" href="css/bootstrap/bootstrap.min.css" />
<link href="css/site.css" rel="stylesheet" />
<link href="BlazorServer.styles.css" rel="stylesheet" />
```

Oprócz tego jest generowany plik *BlazorServer.styles.css* zawierający wszystkie izolowane style. Izolacja stylów zostanie dokładniej opisana w rozdziale 9., „Udostępnianie kodu i zasobów".

Kod strony zawiera komponent `HeadOutlet`, który jest renderowany. Dzięki niemu można zmieniać zawartość nagłówka strony, m.in. znaczniki `title` i `meta`:

```
<component type="typeof(HeadOutlet)" render-mode="ServerPrerendered" />
```

Komponent ten wykorzystamy w rozdziale 5., „Tworzenie zaawansowanych komponentów Blazora" do dodania metadanych i zmiany tytułu strony.

Poniższy niewielki kod wyświetla komunikat, gdy pojawi się błąd:

```
<div id="blazor-error-ui">
    <environment include="Staging,Production">
        An error has occurred. This application may no longer respond until
        ↪reloaded.
    </environment>
    <environment include="Development">
        An unhandled exception has occurred. See browser dev tools for details.
```

```
        </environment>
        <a href="" class="reload">Reload</a>
        <a class="dismiss">✕</a>
    </div>
```

Zalecam, aby po całkowitym zmodyfikowaniu układu strony pozostawić powyższy kod (lub jego zmienioną wersję), ponieważ do aktualizacji interfejsu graficznego jest wykorzystywany skrypt JavaScript. Może się zdarzyć, że strona nie załaduje się, skrypt JavaScript przestanie działać i połączenie SignalR zostanie przerwane. W takim wypadku powyższy kod wyświetli w konsoli przeglądarki czytelny komunikat. Gdy na stronie pojawi się błąd, będzie to wskazówka, aby sprawdzić zawartość konsoli.

Ostatnią ważną rzeczą na tej stronie jest miejsce, w którym dzieje się cała magia: skrypt JavaScript zbierający wszystko w całość:

```
<script src="_framework/blazor.server.js"></script>
```

Skrypt nawiązuje połączenie SignalR z serwerem, wysyła do niego dane i modyfikuje model DOM po odebraniu odpowiedzi.

Index (WebAssembly)

Projekt Blazor WebAssembly jest bardzo podobny do Blazor Server. W projekcie *Blazor WebAssembly.Client* znajduje się plik *wwwroot\index.html* zawierający wyłącznie kod HTML. Nie ma w nim dyrektyw jak w projekcie Blazor Server, natomiast jest znacznik `base`:

```
<base href="/" />
```

Jeżeli aplikacja Blazor WebAssembly będzie hostowana na przykład w serwisie GitHub Pages, powyższy znacznik trzeba zmienić, ponieważ strona zostanie umieszczona w podfolderze serwera.

W odróżnieniu od projektu Blazor Server, plik nie zawiera znacznika `component`. Zamiast niego jest przedstawiony niżej `div`, a w pliku *Program.cs* (opisanym w podrozdziale „WebAssembly Program.cs") znajduje się wiersz, który podłącza do tego znacznika komponent `App`.

```
<div id="app">Loading...</div>
```

Ciąg `Loading...` można zmienić na dowolny inny. Jest to napis, który pojawia się podczas uruchamiania aplikacji.

Nieco inaczej wygląda również kod wyświetlający komunikat o błędzie. Nie ma w nim rozróżnienia środowisk programistycznego i produkcyjnego. Komunikat jest tylko jeden:

```
<div id="blazor-error-UI">
    An unhandled error has occurred.
```

```
    <a href="" class="reload">Reload</a>
    <a class="dismiss">✕</a>
</div>
```

Na końcu pliku jest znacznik ładujący kod JavaScript. Skrypt ten z kolei ładuje cały kod niezbędny do uruchomienia aplikacji Blazor WebAssembly:

```
<script src="_framework/blazor.webassembly.js"></script>
```

Podobnie jak w projekcie Blazor Server, gdzie skrypt komunikuje się z serwerem i modelem DOM, w projekcie Blazor WebAssembly mamy komunikację między środowiskiem uruchomieniowym .NET a modelem.

W tym miejscu jest uruchamiana aplikacja i komponent Razora, taki sam w obu projektach.

App

Komponent `App` jest taki sam w projektach Blazor Server i Blazor WebAssembly. Zawiera on komponent `Router`:

```
<Router AppAssembly="@typeof(App).Assembly">
    <Found Context="routeData">
        <RouteView RouteData="@routeData" DefaultLayout="@typeof(MainLayout)" />
        <FocusOnNavigate RouteData="@routeData" Selector="h1" />
    </Found>
    <NotFound>
        <PageTitle>Not found</PageTitle>
        <LayoutView Layout="@typeof(MainLayout)">
            <p role="alert">Sorry, there's nothing at this address.</p>
        </LayoutView>
    </NotFound>
</Router>
```

Powyższy kod kieruje żądaniami i wykorzystując dyrektywę `@page`, wyszukuje odpowiedni komponent do wyświetlenia. Jeżeli żądanie dotyczy nieistniejącej strony, wyświetlany jest komunikat o błędzie. W rozdziale 8., „Uwierzytelnienie i autoryzacja”, zmodyfikujesz ten plik, gdy będziesz implementować uwierzytelnianie użytkowników.

Komponent `App` zawiera również domyślny układ strony, który można zastąpić innym. Każda strona może mieć inny układ, ale zazwyczaj stosuje się jeden w całej witrynie. Tutaj domyślny układ ma nazwę `MainLayout`.

MainLayout

Układ `MainLayout` zawiera wszystkie domyślne komponenty strony. Składa się z dwóch znaczników `div`, po jednym dla bocznego panelu i głównej treści:

```
@inherits LayoutComponentBase

<PageTitle>BlazorServer</PageTitle>

<div class="page">
    <div class="sidebar">
        <NavMenu />
    </div>

    <main>
        <div class="top-row px-4">
           <a href="https://docs.microsoft.com/aspnet/" target="_blank">About</a>
        </div>

        <article class="content px-4">
            @Body
        </article>
    </main>
</div>
```

Dwie rzeczy w tym pliku, o których należy pamiętać, to dyrektywy `@inherits` i `@Body`. Reszta to znaczniki biblioteki Bootstrap. Pierwsza dyrektywa importuje z komponentu `LayoutComponentBase` kod wykorzystywany w układzie, a druga renderuje komponent w chwili wyświetlenia strony.

Bootstrap

Bootstrap to jedna z najpopularniejszych bibliotek stylów CSS, przeznaczona do tworzenia responsywnych stron internetowych, głownie dla urządzeń mobilnych. Została opracowana przez programistów Twittera. Informacje o niej można znaleźć na stronie *https://getbootstrap.com*.

Odwołanie do biblioteki Bootstrap znajduje się w pliku *wwwroot\index.html*. Na początku układu `MainLayout` znajduje się komponent Razor o nazwie `NavMenu`. Plik, w którym jest zdefiniowany, jest zapisany w katalogu *Shared* i zawiera następujący kod:

```
<div class="top-row ps-3 navbar navbar-dark">
    <div class="container-fluid">
        <a class="navbar-brand" href="">BlazorServer</a>
        <button title="Navigation menu" class="navbar-toggler"
        ↪@onclick="ToggleNavMenu">
            <span class="navbar-toggler-icon"></span>
```

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

<div class="@NavMenuCssClass nav-scrollable" @onclick="ToggleNavMenu">
    <nav class="flex-column">
        <div class="nav-item px-3">
            <NavLink class="nav-link" href="" Match="NavLinkMatch.All">
                <span class="oi oi-home" aria-hidden="true"></span> Home
            </NavLink>
        </div>
        <div class="nav-item px-3">
            <NavLink class="nav-link" href="counter">
                <span class="oi oi-plus" aria-hidden="true"></span> Counter
            </NavLink>
        </div>
        <div class="nav-item px-3">
            <NavLink class="nav-link" href="fetchdata">
                <span class="oi oi-list-rich" aria-hidden="true"></span>
                ↪Fetch data
            </NavLink>
        </div>
    </nav>
</div>

@code {
    private bool collapseNavMenu = true;

    private string? NavMenuCssClass => collapseNavMenu ? "collapse" : null;

    private void ToggleNavMenu()
    {
        collapseNavMenu = !collapseNavMenu;
    }
}
```

Powyższy kod wyświetla po lewej stronie okna standardowe menu złożone z trzech opcji. Jeżeli strona jest otwierana na urządzeniu mobilnym, menu jest przekształcane w ikonę z trzema kreskami. Tego rodzaju menu zazwyczaj implementuje się w języku JavaScript. Tutaj użyte są jedynie style CSS i kod C#.

Oprócz tego powyższy kod zawiera komponent `NavLink`, również zawarty w bibliotece Bootstrap. Reprezentuje on znacznik `a` i sprawdza aktualną ścieżkę. Jeżeli jest taka sama jak wskazana, komponent automatycznie dodaje do znacznika klasę `active`.

W dalszej części książki poznasz jeszcze kilka innych komponentów ułatwiających pracę. Szablon zawiera również kilka stron, którymi na razie nie będziemy się zajmować. Przyjrzymy się im w następnym rozdziale, kiedy przejdziemy do komponentów.

CSS

W katalogu *Shared* znajdują się m.in. dwa pliki stylów CSS: *NavMenu.razor.css* i *Main Layout.razor.css*. Zgodnie z nazwami dotyczą one tylko określonych komponentów. Izolowaniem stylów zajmiemy się w rozdziale 9., „Udostępnianie kodu i zasobów".

Podsumowanie

W tym rozdziale przygotowałeś środowisko programistyczne i utworzyłeś swoje pierwsze aplikacje Blazor WebAssembly i Blazor Server. Dowiedziałeś się, w jakiej kolejności są stosowane klasy, komponenty i układy, co ułatwi Ci analizowanie kodu. Poznałeś kilka różnic między projektami Blazor Server a Blazor WebAssembly.

W następnym rozdziale zostawimy na chwilę platformę Blazor, aby przyjrzeć się zarządzaniu stanem aplikacji i skonfigurować repozytorium wpisów na blogu.

Poznaj przenośność i moc platformy Blazor!

Do niedawna tworzenie interaktywnych stron internetowych wymagało programowania w JavaScripcie. Obecnie można się posłużyć platformą Blazor, która umożliwia efektywne tworzenie dynamicznych aplikacji internetowych opartych na technologii .NET i kodzie C#. Jeśli chcesz za pomocą Blazora tworzyć złożone aplikacje i wdrażać je w środowisku produkcyjnym, musisz dobrze poznać zasady pracy z tą platformą.

Oto praktyczny, przystępnie napisany przewodnik, który stanowi wprowadzenie do pracy z technologią Blazor. Opisuje możliwości modeli Server i WebAssembly, przedstawia także krok po kroku proces powstawania aplikacji internetowej. Dzięki temu płynnie przejdziesz do tworzenia projektów Blazor, nauczysz się składni języka Razor, będziesz też budować własne komponenty i weryfikować zawartość formularzy. W tym wydaniu omówiono również generatory kodu źródłowego i zasady przenoszenia komponentów witryn utworzonych w innych technologiach do platformy Blazor. W trakcie lektury dowiesz się, jak tworzyć uniwersalne aplikacje za pomocą wersji Blazor Hybrid wraz z platformą .NET MAUI.

Z tą książką nauczysz się:

- tworzenia prostych i zaawansowanych komponentów Blazor
- właściwego stosowania projektów Blazor Server i Blazor WebAssembly
- pisania interfejsów Minimal API
- korzystania z interoperacyjnych bibliotek JavaScript zawartych w platformie Blazor
- diagnozowania aplikacji Blazor
- testowania komponentów Blazor za pomocą biblioteki bUnit

Jimmy Engström programuje od siódmego roku życia. Używa platformy Blazor w środowiskach produkcyjnych od chwili udostępnienia jej pierwszej oficjalnej wersji. Wśród programistów jest znany jako światowej klasy ekspert w dziedzinie technologii .NET. Docenił to Microsoft, który przez dziewięć lat z rzędu przyznawał mu tytuł Microsoft Most Valuable Professional.

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 978-83-289-0419-4
9 788328 904194
Cena: 69,00 zł
packt