Wrocław, 22 kwietnia 2025

Platformy **P**rogramistyczne `.NET` i `Java`

Laboratorium 4

Aplikacja webowa w technologii ***`ASP.NET Core`***

prowadzący: *Dr inż. Radosław Idzikowski, mgr inż. Michał Jaroszczuk*

1 Cel laboratorium

Celem laboratorium jest zapoznanie się z podstawami projektowania i implementacji aplikacji webowych w technologii `ASP.NET Core`. W ramach zajęć należy w pierwszej kolejności zapoznać się z technologią `Blazor` w ramach platformy `ASP.NET Core`, następnie stworzyć aplikację bazodanową oraz w ostatnim kroku opublikować ją w ramach chmury `Azure`. Praca będzie oceniania na bieżąco podczas zajęć. **Ukończenie każdego etapu powinno być zgłoszone prowadzącemu w celu akceptacji i odnotowania postępów**. Program należy umieścić na repozytorium `github` i wysłać zaproszenie do prowadzącego. Czas na wykonanie zadania to dwa zajęcia laboratoryjne. Podczas pierwszego spotkania trzeba wykonać co najmniej jedno zadanie zdefiniowane w Rozdziale 3.

2 Zadania

W ramach zajęć należy w zespołach wykonać następujące zadania:

1. Użycie platformy `Blazor`.
2. Bazodanowa aplikacja webowa.
3. Wdrożenie aplikacji w chmurze `Azure`.

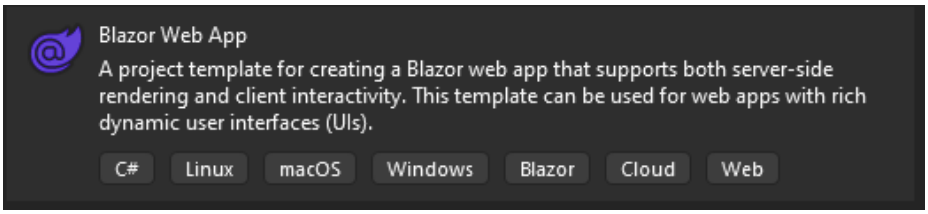
Za wykonanie zadania nr 1 jest ocena dostateczna, za każde kolejne zadanie jest +1 do oceny. Na ocenę bardzo dobrą (5.0) należy wykonać wszystkie trzy zadania. Link do repozytorium należy przesłać dopiero po oddaniu i ocenieniu pracy na laboratorium.

3 Opis Zadań

W tej sekcji zostaną kolejno omówione wszystkie zadania do wykonania podczas laboratorium. Komputery w laboratorium zostały przygotowane do przeprowadzenia zajęć.

3.1 Zadanie 1

W ostatnich latach widać trend polegający na wypieraniu aplikacji desktopowych przez aplikacje webowe. Wymaga to innego niż w przypadku aplikacji desktopowych podejścia, stąd powstanie framework'ów takich jak `MVC` (implementujący wzorzec Model - View - Controller) czy `Razor` (pozwalający na osadzanie kodu platformy `.NET` na stronach internetowych). Technologie te stały się jednak niewystarczające ze względu na konieczność kontroli procesu zarówno po stronie klienta jak i serwera. Stąd też niedawne wypuszczenie na rynek technologii `Blazor` przez firmę Microsoft, która osbługuje renderowanie zarówno po stronie klienta jak i serwera w jednym modelu programowania.



Rysunek 1: Tworzenie nowego projektu aplikacji webowej `Blazor`

W ramach zadania należy utworzyć projekt `Blazor Web App` wedle szablonu. Następnie trzeba zapoznać się z budową tego typu aplikacji na utworzonym przykładzie. Poniżej opisano krok po kroku kolejne elementy. Na koniec będzie trzeba wprowadzić dwie drobne modyfikacje: (1) prosty licznik oraz (2) filtr danych.

Aplikacja `Blazor`

W pierwszej kolejności należy utworzyć nowy projekt `Blazor Web App` (aplikacji webowej `Blazor` jak na Rys. 1). W kolejnym kroku należy wybrać zgodność z `.NET 8.0 (Long Term Support)` oraz:

- `Authentication type: None`,
- `Configure for HTTPS : True`,
- `Interactive render mode: Server`,
- `Interactivity location: Per page/component`,
- `Include sample pages : True`,
- `Do not use top-level statements : False`,

Nowo utworzony projekt zawiera niezbędne pliki do uruchomienia prostej aplikacji `Blazor`. Plik `Program.cs` jest punktem startowym dla aplikacji, który uruchamia serwer oraz w którym konfigurowane są usługi i oprogramowanie pośredniczące aplikacji. W folderze `Components` znajdują się elementarne pliki aplikacji jak:

- `App.razor` – główny komponent aplikacji,
- `Routes.razor` – odpowiedzialny za konfigurację routera `Blazor`,
- pozostałe pliki odpowiedzialne za `Layout` (wygląd) strony.

Ponadto katalog `Pages` zawiera kilka już przygotowanych, przykładowych stron internetowych dla aplikacji. Plik `launchSettings.json` wewnątrz katalogu `Properties` definiuje różne ustawienia profilu dla środowiska lokalnego. Numer portu jest automatycznie przypisywany podczas tworzenia projektu i zapisywany w tym pliku.

Niestety tak przygotowana aplikacja nie uruchomi się w każdej przeglądarce, co zasygnalizowane będzie kodem błędu `NET::ERR_CERT_INVALID`. Rozwiązaniem jest skonfigurowanie uwierzytelniania za pomocą certyfikatu. W tym celu należy dodać pakiet `Microsoft.AspNetCore.Authentication.Certificate` z wykorzystaniem menadżera pakietów `NuGet`. Następnie możemy dodać prosty domyślny certyfikat, wystarczający na potrzeby zadania (ale niezalecany w komercyjnych aplikacjach). W pierwszej kolejności trzeba skonfigurować odpowiedni serwis odpowiedzialny za uwierzytelnianie, koniecznie jeszcze przed zbudowaniem aplikacji `var app = builder.Build();`.

```
builder.Services.AddAuthentication(
        CertificateAuthenticationDefaults.AuthenticationScheme)
    .AddCertificate();
```

Następnie należy wywołać metody odpowiedzialne za uwierzytelnianie i autoryzacji. Koniecznie przed uruchomieniem już zbudowanej aplikacji `app.Run();`.

```
app.UseAuthentication();
app.UseAuthorization();
```

Tak przygotowaną Aplikację uruchamiamy standardowo zieloną strzałką *Start Debugging* przy użyciu protokołu `https`. Aplikacja może być oczywiście uruchomiona również z wykorzystaniem protokołu `http`. Zmiany w kodzie możemy aplikować bez wyłączania programu stosując przycisk `Hot Reload` (symbol czerwonego płomienia). W przypadku utworzenia projektu z włączoną opcją `Include sample pages`, otrzymamy trzy przykładowe strony, każda o rozszerzeniu `.razor`. Najprostszą konstrukcję ma strona startowa `Home.razor`.

```
@page "/"
<PageTitle>Home</PageTitle>
<h1>Hello, world!</h1>
Welcome to your new app.
```

Aplikacje `Blazor` opierają się na komponentach. Komponent w `Blazor` to element interfejsu użytkownika, taki jak strona, okno dialogowe czy formularz wprowadzania danych. Klasa komponentu jest zwykle pisana w formie strony znaczników `Razor` z rozszerzeniem pliku `.razor`. `Razor` to składnia do łączenia znaczników `HTML` z kodem `C#`. `Blazor` wykorzystuje naturalne znaczniki `HTML` do komponowania interfejsu użytkownika. Lepiej to jest widocznie na przykładzie z licznikiem kliknięć przy użyciu przycisku w komponencie `Counter.razor`.

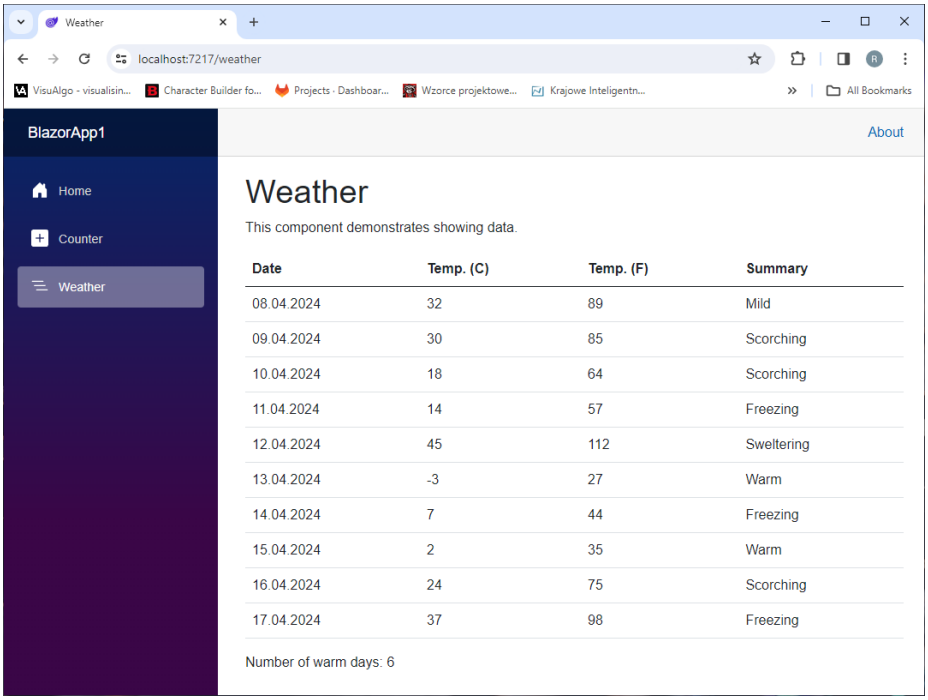
```
@page "/counter"
@rendermode InteractiveServer
<PageTitle>Counter</PageTitle>
<h1>Counter</h1>
<p role="status">Current count: @currentCount</p>
<button class="btn btn-primary" @onclick="IncrementCount">Click me</button>
@code {
    private int currentCount = 0;

    private void IncrementCount()
    {
        currentCount++;
    }
}
```

Dyrektywa `@rendermode` umożliwia interaktywne renderowanie komponentu na serwerze, dzięki czemu może obsługiwać zdarzenia interfejsu użytkownika przesyłane z przeglądarki. Przy każdym wejściu na stronę `counter` licznik będzie zerowany. Dużo ciekawszym i bardziej rozbudowanym przykładem jest strona `weather`, gdzie jest prognozowana (w tym wypadku losowo) pogoda na pięć najbliższych dni, a następnie wyświetlana w formie tabeli. Może się zdarzyć, że po wejściu na stronę `weather` tabela z wartościami temperatur zostanie dwukrotnie przeładowana. Dzieje się tak dlatego, że domyślnie uruchomione jest prerenderowanie strony. Strona jest raz renderowana na serwerze (prerendering), gdzie zwracany jest HTML, dzięki czemu przeglądarka może szybko i ładnie pokazać tabelę, a następnie komponent zostanie ponownie wyrenderowany przy użyciu preferowanego interaktywnego trybu renderowania (serwer lub WASM). Najprostszą opcją na naprawienie tego zjawiska jest wyłączenie interaktywnego trybu renderowania poprzez zastosowanie poniższej flagi:

```
@rendermode @(new InteractiveServerRenderMode(false))
```

Efekt można także usunąć przez zastosowanie utrwalenia stanu strony pomiędzy renderowaniami z użyciem składnika `PersistentComponentState`.

W ramach zadania należy w pierwszej kolejności wydłużyć termin prognozy do 10 dni. Następnie trzeba dodać w sekcji `@code` zmienną prywatną `warmDays` przechowującą liczbę ciepłych dni (powyżej 15^oC). W funkcji `OnInitializedAsync()` należy dopisać fragment kodu odpowiedzialny za ich zliczenie (bezpośrednio po wygenerowaniu). Na koniec jeszcze należy w części z kodem `HTML` dodać akapit z wyświetloną sumą (analogicznie jak status licznika) jak na rysunku 2. Dodanie nowych lub zmiana nazwy zmiennych czy funkcji wymusi reset aplikacji.



Rysunek 2: Strona z prognozą pogody uzupełniona o licznik ciepłych dni

Kolejnym krokiem jest dodanie przycisku, który pozwoli odfiltrować dni z temperaturą poniżej 15^oC. W pierwszej kolejności należy dodać funkcję (może być typu `void`), która przy użyciu składni `LINQ` pozwoli odfiltrować tablicę `forecasts`. Na koniec trzeba pamiętać, aby wymusić utworzenie tablicy metodą `ToArray()`. Teraz już wystarczy dodać przycisk na dole strony oraz na początku dyrektywę komponentu `@rendermode InteractiveServer`. Przycisk dodajemy używając znaczników `HTML <button>Text</button>`. Do zdefiniowana jego wyglądu możemy użyć na przykład klasy `btn btn-primary`. Zdefiniowaną wcześniej metodę należy podpiąć pod wydarzenie odpowiedzialne za kliknięcie `@onclick="WarmDaysFilter"` lub w przypadku funkcji z parametrem za pomocą delegata `@onclick="() => WarmDaysFilter(15)"`. Następnie należy dodać również przycisk `Restore`, który pozwoli przywrócić tabelę do pierwotnego stanu.

Ostatnim etapem jest dodanie opcji filtrowania po nazwie. W tym celu należy zdefiniować funkcję `private void Input(ChangeEventArgs arg)`, gdzie parametr `arg` pozwoli na przechwycenie wprowadzonego tekstu, ale trzeba jego pole `Value` zrzutować na `string` lub wywołać metodę `ToString()`. Do samego filtrowania tekstu warto skorzystać z `LINQ` oraz metody `Contains`. Na koniec należy dodać jeszcze komponent odpowiedzialny za wprowadzanie danych oraz podpiąć napisaną funkcję `<input class="form-control" @oninput="@Input" />`. Alternatywnie możemy skorzystać z wydarzenia `@onchange` jeśli chcemy potwierdzić wprowadzenie frazy enterem. Powyższe funkcje zostały opisane jako funkcje typu `void`. Dla funkcji, które zwracają wynik natychmiastowo takie rozwiązanie jest dopuszczalne. Jednak według zasad poprawnego projektowania funkcje powinny być asynchroniczne, co wymusza typ `Task`.

3.2 Zadanie 2

W ramach zadania należy w osobnym projekcie wykonać prostą aplikację bazodanową w technologii `Blazor` z wykorzystaniem `EntityFramework`. Wykonana aplikacja musi zawierać system logowania oraz powinna pozwalać na wprowadzanie ocen dla filmów, gier czy książek.

Do autoryzacji użytkowników można skorzystać z gotowego rozwiązania jakie oferuje nam projekt `Blazor Web App`. Wystarczy podczas tworzenia nowego projektu wybrać opcję `Individual Accounts` w polu `Authentication type`. W efekcie otrzymamy prawie gotowy system do autoryzacji użytkowników (potwierdzenie rejestracji poprzez kliknięcie w link po jej wykonaniu). Przed uruchomieniem aplikacji należy jeszcze stworzyć migrację oraz wykonać aktualizację bazy danych z poziomu `Package Manager Console`. Sposób obsługi bazy danych (jej definicji, tworzenia czy migracji) nie różni się od poznanego już sposobu przy okazji laboratorium nr 2. Docelowo dostęp do części stron powinien zostać zablokowany z użyciem atrybutu `Authorize` (sposób użycia pokazano w komponencie `Auth.razor`). Ponadto część podstron w `menu` nawigacyjnym również powinna być niewidoczna dla niezalogowanych użytkowników (komponent `NavMenu.razor`, sekcja `<AuthorizeView>`). Alternatywnie dla chętnych, może spróbować zaimplementować autoryzację z wykorzystaniem konta `Google` lub `Microsoft`, co może być przydatne w przyszłych studenckich bądź komercyjnych projektach.

W przypadku części bazodanowej w pierwszej kolejności należy utworzyć nową klasę pod elementy, które będziemy przechowywać. Można to zrobić bezpośrednio w folderze `Components` lub stworzyć wewnątrz niego dedykowany folder. Poniżej przedstawiono przykładową klasę przechowującą informację o filmach. Do poprawnego działania `EntityFramework` składowe klasy powinny być w formie własności. W celu uzyskania maksymalnej oceny z tej części, składowe powinny być o różnych typach. Składowa Id jest obowiązkowa, pozostałe mogą przyjąć wartość `null`. Przy zmiennej `DataTime?` warto dodać adnotację, która wymusi format daty, widoczne będzie to w późniejszym etapie.

```
public class Movie
{
    public int Id { get; set; }
    public string? Title { get; set; }
    public string? Description { get; set; }
    [DataType(DataType.Date)]
    public DateTime? RelaseDate { get; set; }
    public float? Rate { get; set; }
}
```

Następnie należy wygenerować odpowiednie podstrony do obsługi bazy danych. W tym celu skorzystać można z opcji dodania nowego elementu szkieletowego (`New Scaffolded Item`), następnie `Razor Compontens using Entity Framework (CRUD)`. Jako `Template` wybieramy `CRUD`, w `Model class` wybieramy naszą przygotowaną klasę, a jako `DbContext` wybieramy domyślnie utworzoną bazę danych przy tworzeniu kont. W efekcie otrzymamy podstrony z podstawowymi funkcjonalnościami (wylistowanie wszystkich elementów, dodawanie, edycja, usuwanie). W `NavMenu.razor` należy dodać odnośnik tylko do głównej podstrony `Index.razor`. Przed uruchomieniem aplikacji ponownie należy stworzyć nową migrację oraz zaktualizować bazę danych. Stworzona aplikacja powinna zawierać następujące modyfikacje i funkcjonalności:

- w widoku `Details` istnieje możliwość dodania oceny, ocena musi zostać zaktualizowana poprzez przeliczenie z już istniejącą oceną a nie podmieniona,
- w widoku `Index` istnieje możliwość sortowania po różnych kolumnach,
- w widoku `Index` nie wszystkie informacje powinny być widoczne (np. brak opisu),
- dostęp do części stron jest zablokowany dla niezalogowanego użytkownika,
- `menu` nawigacyjnego jest poprawione i zawiera dodane podstrony,

- w bazie danych przechowywany jest obrazek, dotyczący ocenianego elementu. Powinien być on przechowywany jako `url` i wyświetlany na podstronie `Details`.

W celu otrzymania maksymalnej oceny za to zadanie należy uwzględnić dodatkowo oprócz wcześniej wymienionych funkcjonalności minimum jeden punkt z poniższej listy. Brak dodatkowej funkcjonalności będzie skutkował obniżeniem oceny (przy wykonaniu wyszystkich zadań - 0,5 oceny w dół, przy wykonaniun zadań 1 i 2 - 0,3 oceny w dół). Po konsultacji z prowadzącym jest możliwość zastąpienia zaproponowanych funkcjonalności innymi - liczy się inwencja i chęć nauki nowych rozwiązań :-) Przykładowe inspiracje do rozwoju aplikacji, które będą dodatkowo punktowane:

- wcześniej wspomniane wykonanie autoryzacji użytkowników z wykorzystaniem konta `Google` lub `Microsoft`,
- zakotwiczenie webowego komponentu (np. mapy `Google`, `Twitter`, pogoda, itp.),
- przesył danych pomiędzy dwoma niezależnymi aplikacjami webowymi (swojej i kolegi lub koleżanki) zrobionymi podczas zajęć,
- możliwość wgrania pliku (z dysku, zdjęcia z kamerki komputerowej, nagrania dźwięku, itp).

3.3 Zadanie 3

Celem zadania jest wdrożenie aplikacji w ramach zasobów `Microsoft Azure`. Dzięki temu możliwy będzie dostęp do naszej strony z innego dowolnego urządzenia z dostępem do sieci Internet.

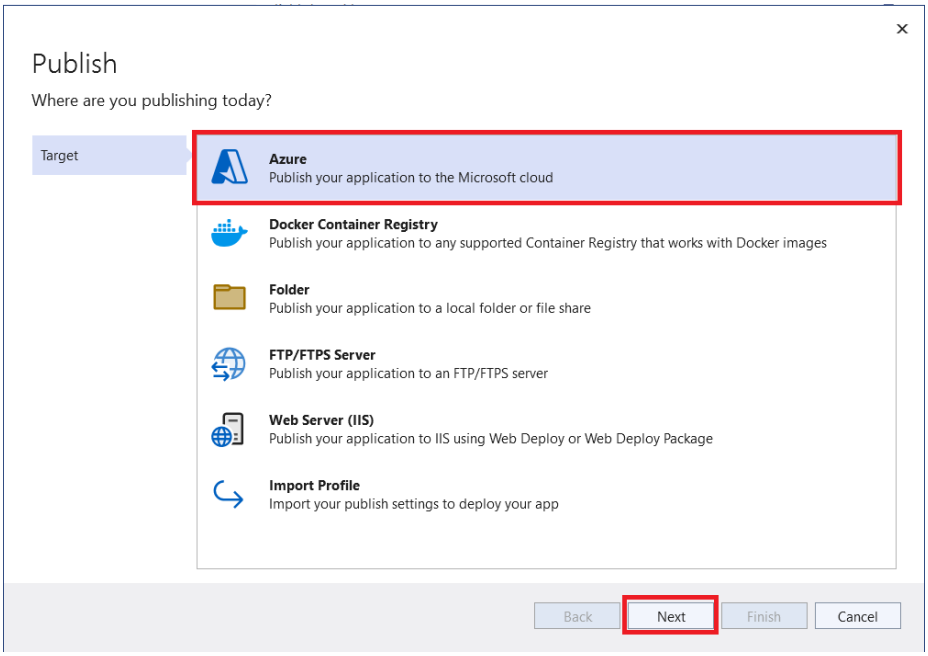
3.3.1 Publikowanie aplikacji webowej

Zanim zaczniemy, musimy się zalogować na studenckie konto `Microsoft` w programie Visual Studio w celu użycia subskrypcji, której aktywacja opisana będzie w dalszej części zadania. Jeśli `Visual Studio` było instalowane z licencji przypisanej do studenckiego adresu e-mail, to nie powinno to stanowić większego problemu. Proces publikowania aplikacji należy rozpocząć od wybrania odpowiedniej akcji z poziomu `Solution Explorer`, klikając prawym przyciskiem myszy na projekt aplikacji webowej i wybierając opcję `Publish...` lub z zakładki `Build`, również wybierając opcję `Publish...`. Interesująca nas opcja to `Azure - Host your application to the Microsoft cloud`, pokazana na Rysunku 3 (jak widać nie jest to jedyna możliwość). Chcąc hostować aplikację z zadania drugiego w serwisie `Azure`, należy hostować w nim również bazę danych (w poprzednim zadaniu była ona uruchomiona lokalnie).

Następnie należy wybrać opcję `Azure App Service (Windows)` lub alternatywnie `Azure App Service (Linux)` - definiuje ona pod jakim systemem uruchomiona będzie aplikacja webowa na serwerze. Do wykonania zadania będzie trzeba aktywować darmową subskrypcję `Azure` w ramach studenckiego konta. Jeśli subskrypcja została wcześniej aktywowana, powinna się automatycznie podpowiedzieć, w przeciwnym wypadku pojawi się link do portalu `Azure`, gdzie subskrypcję można aktywować. Co ważne, w momencie dodawania nowej subskrypcji, nie należy wybierać opcji `Start free`, która wymagać będzie podpięcia karty płatniczej, ale wybrać opcję `Azure for Students`. Przy korzystaniu z konta podpiętego pod studencki adres email otrzymujemy kredyt w wysokości 100$ do wykorzystania na hostowanie aplikacji, który będzie pomniejszany w zależności od naszej aktywności w serwisie `Azure`.

Po poprawnym aktywowaniu subskrypcji, należy utworzyć nowy `AppService` wybierając akcję `Create new`. Wiązać się to będzie z:

- nadaniem nowej nazwy serwisu,
- wyborem subskrypcji,
- utworzeniem nowej grupy zasobów,



Rysunek 3: Wybór miejsca publikacji projektu

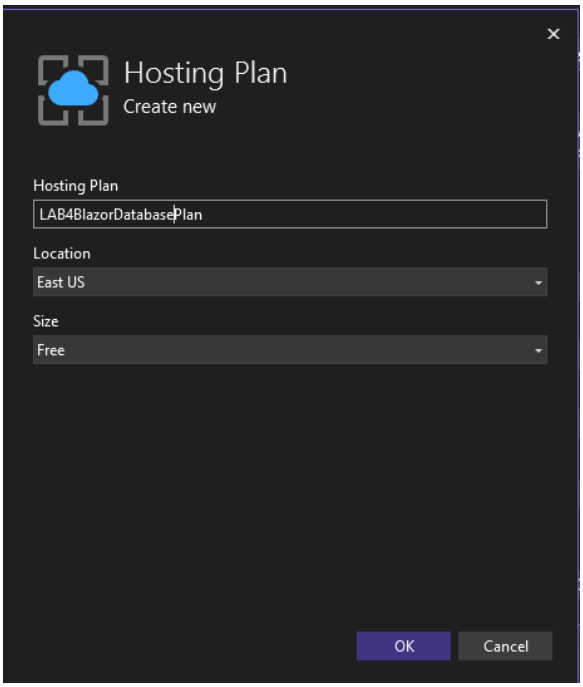
- stworzeniem nowego planu hostingu, w którym należy nadać nazwę, wybrać lokalizację `East US` oraz wybrać rozmiar `Free`.

Wymagane kroki pokazano na Rysunku 5, który przedstawia okno, mające się wyświetlić po wyborze opcji `Create new` w zakładce `App Service` oraz na Rysunku 4, który przedstawia tworzenie planu hostingu.

Po wpisaniu wszystkich wymaganych opcji należy wybrać opcję `Create`, odczekać chwilę potrzebną na utworzenie serwisu a następnie zakończyć tworzenie `Finish` i zamknąć okno. Po utworzeniu nowego elementu `App Service`, należy na stronie `Publish` wybrać opcję `Publish`. Pod stworzonym linkiem powinna być dostępna nasza witryna. Aby opublikować zmiany dokonane w aplikacji już po jej opublikowaniu, należy ponownie wybrać opcję `Publish`.

3.3.2 Hostowanie bazy danych w chmurze

Po poprawnym przejściu opisanych wcześniej kroków, prawdopodobnie okaże się, że logowanie ani podejrzenie stworzonych rekordów z obiektami kultury (filmami, grami, itp.) nie jest możliwe. Wynika to z faktu, iż do poprawnego działania aplikacji konieczna jest jeszcze konfiguracja bazy danych, która wcześniej była uruchomiona lokalnie. Chcąc podejrzeć szczegóły mało mówiącego błędu wyświetlającego się w przeglądarce należy podmienić zmienną środowiskową `ASPNETCORE_ENVIRONMENT` na wartość `Development`. Można to zrobić z poziomu portalu Azure (`https://portal.azure.com/`), tak jak pokazano na Rysunku 6. Można użyć także konsoli `PowerShell`, która również znajduje się w portalu Azure. Porównanie wyświetlanych błędów po zmianie wartości zmiennej zobaczyć można na Rysunku 7. Na szczegółowym errorze widać, iż brakuje połączenia z bazą danych.



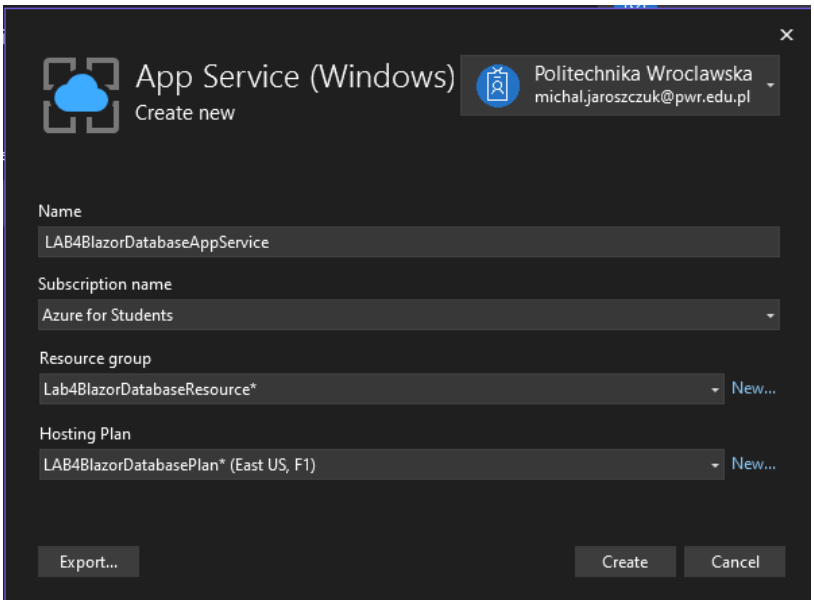
Rysunek 4: Tworzenie nowego planu hostingu

Aby utworzyć bazę danych w chmurze należy dodać nową zależność `Azure SQL Database`, której konfiguracja pokazana jest na Rysunku 8. Samą zależność dodajemy z poziomu strony `Publish`. Zależność do lokalnej bazy danych może zostać usunięta. Następnie należy nadać nazwę bazie danych, pozostawić grupę zasobów taką jaka została przypisana wcześniej przy hostowaniu aplikacji webowej, oraz utworzyć nowy serwer bazy poprzez akcję `New...` (Rysunek 9). Przy tworzeniu serwera SQL, konieczne będzie podanie loginu i hasła, będących później podstawą do utworzenia connection stringa z bazą danych (Rysunek 10).

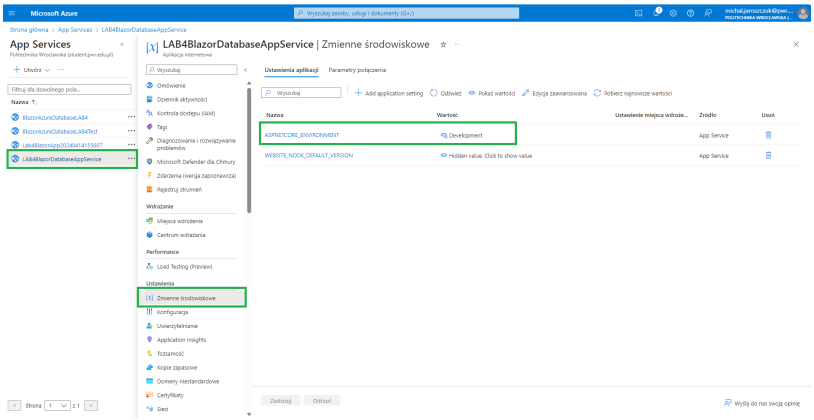
Po utworzeniu bazy danych, należy podać zdefiniowane wcześniej login i hasło oraz nadać nazwę dla connection stringa. Wygeneruje się on automatycznie w tym samym oknie i będzie potrzebny do podania go w programie celem powiązania z bazą danych (Rysunek 11). Następnie należy zakończyć tworzenie bazy. Skopiowany connection string należy teraz przekazać w programie. Można to zrobić na dwa sposoby - albo z użyciem buildera i definicją connection stringa w pliku `appsettings.json` albo podając go bezpośrednio w sposób pokazany na Rysunku 12 w pliku `Program.cs`.

Można teraz podjąć próbę opublikowania aplikacji. Okaże się, że próba dostępu do bazy danych znów skończy się niepowodzeniem. Jednak dzięki wcześniejszej modyfikacji zmiennej odpowiedzialnej za podgląd błędu możemy łatwo zauważyć, iż brakuje migracji bazy danych. Można ją wykonać albo klikając przycisk dostępny w treści błędu na stronie albo zaznaczając checkbox `Apply this migration on publish` dostępny po wejściu w `More actions` → `Edit` → `Entity Framework Migrations` i pokazany na Rysunku 13.

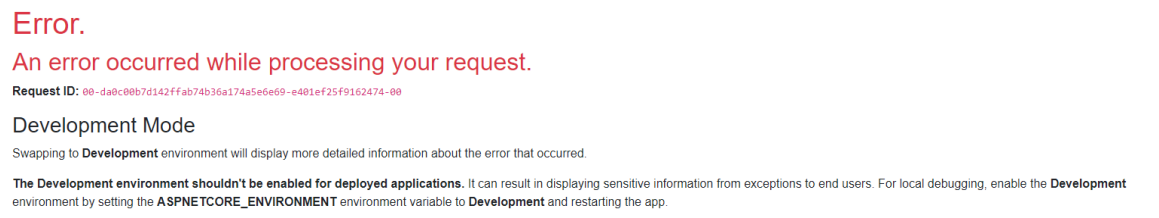
Warunkiem zaliczenia zadania, jest zaprezentowanie prowadzącemu Zadania 2, uruchomionego z poziomu platformy `Azure`, a nie lokalnie z działającą bazą danych w chmurze. Po prezentacji zadania, warto usunąć/dezaktywować serwisy z portalu `Azure` aby niepotrzebnie nie tracić środków na hostowanie aplikacji po zaliczeniu laboratorium ;-).



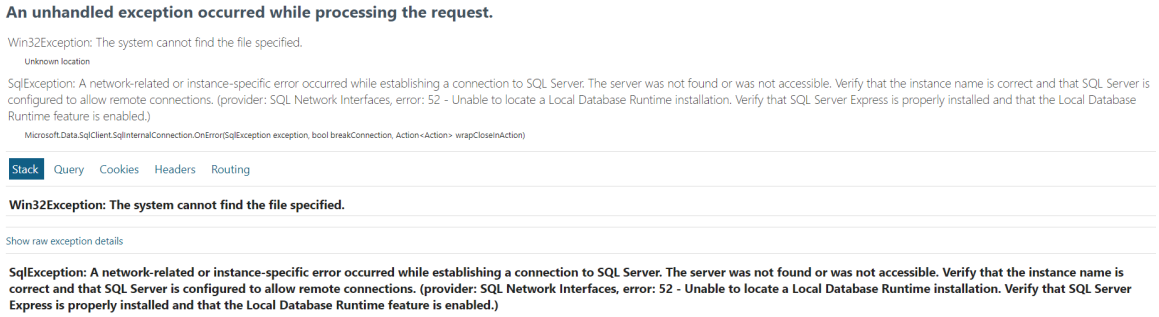
Rysunek 5: Tworzenie nowego serwisu aplikacji



Rysunek 6: Tworzenie nowego serwisu aplikacji

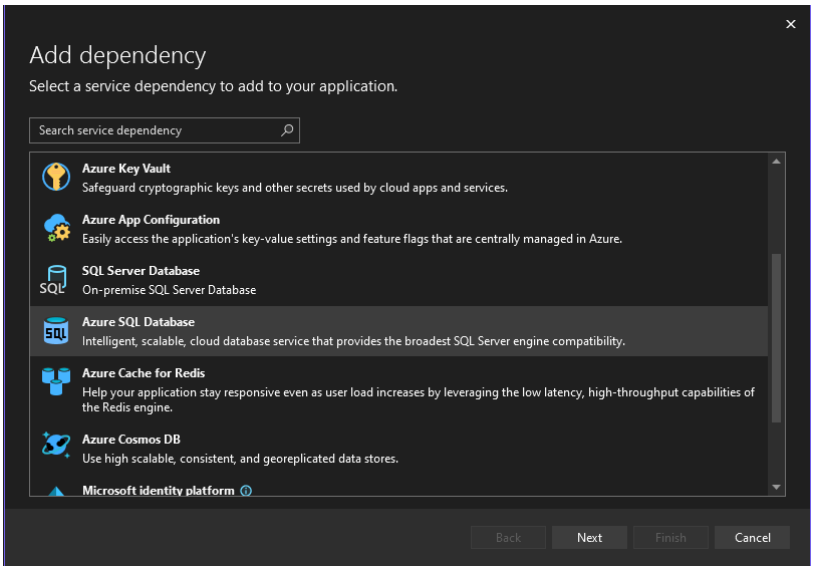


(a) Ogólny stacktrace błędu

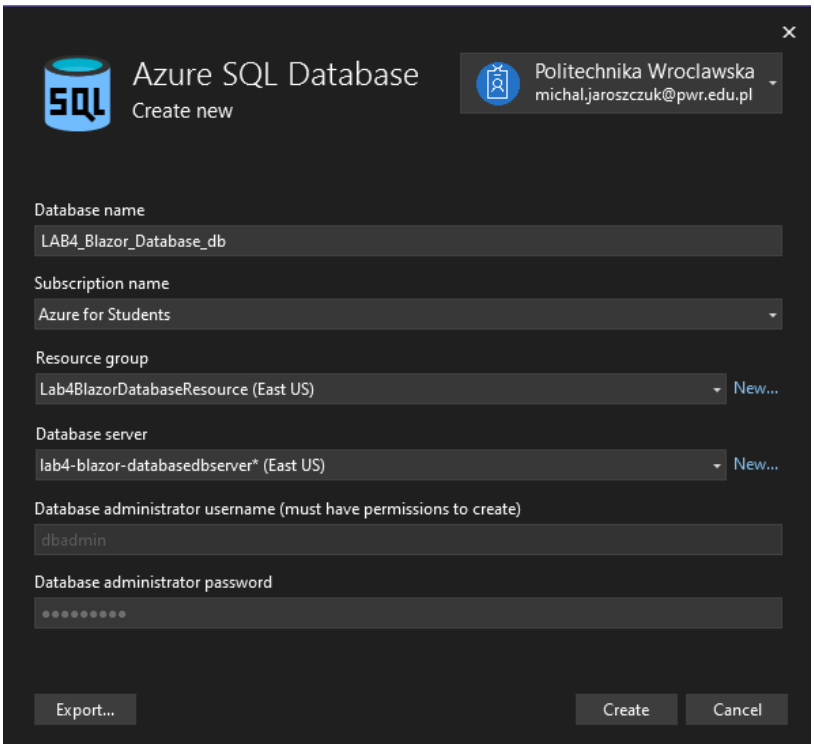


(b) Szczegółowy stacktrace błędu

Rysunek 7: Pokazanie różnicy pomiędzy wyświetlaniem błędów przy zmianie wartości zmiennej `ASPNETCORE_ENVIRONMENT`

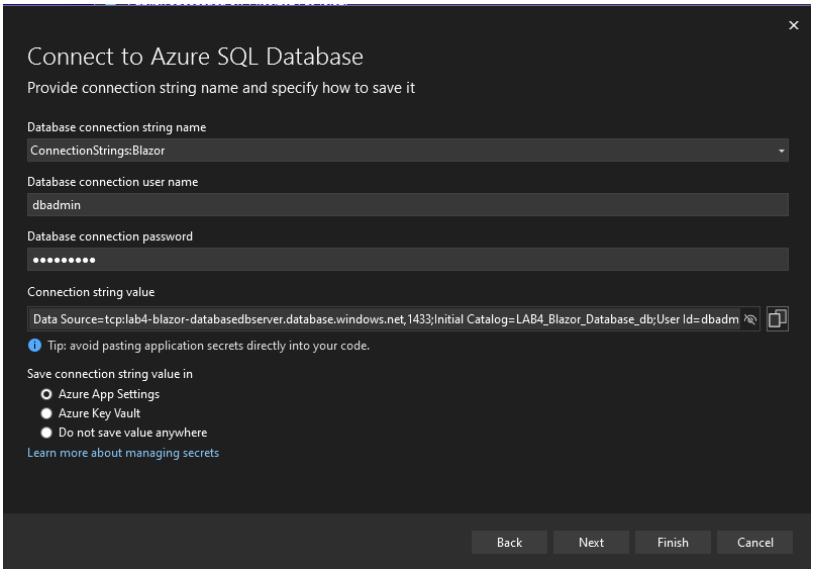


Rysunek 8: Dodanie dependency do bazy danych w chmurze



Rysunek 9: Tworzenie zdalnej bazy danych

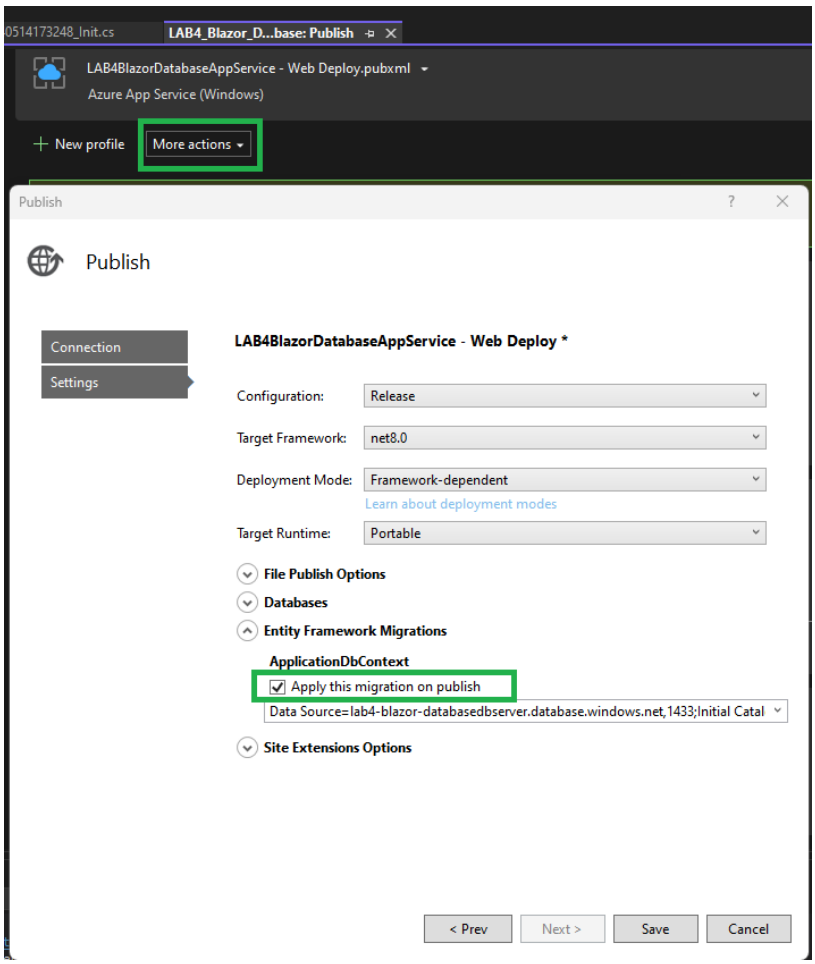
Rysunek 10: Definiowanie parametrów serwera bazy danych



Rysunek 11: Definiowanie połączenia z bazą danych i connection stringa

```
//var connectionString = builder.Configuration.GetConnectionString("DefaultConnection") ?? throw new
  InvalidOperationException("Connection string 'DefaultConnection' not found.");
var connectionString = "Data Source=tcp:lab4 - blazor - databasedbserver.database.windows.net,1433; Initial Catalog =
  LAB4_Blazor_Database_db; User Id = dbadmin@lab4-blazor-databasedbserver; Password = 1234QWer!";
```

Rysunek 12: Przekazanie connection stringa do programu



Rysunek 13: Parametr umożliwiający wykonanie ostatniej migracji bazy