Politechnika Rzeszowska

Wydział Elektrotechniki i Informatyki

**Aplikacje internetowe**

Laboratorium

Aplikacja do tworzenia list zadań  
z wykorzystaniem .NET CORE MVC

Miś Mateusz

I EF/AA-ZU

152175

Rzeszów 2021

[**Opis aplikacji**](#_b4ckwo3qlvz6) **3**

[**Model danych**](#_7svidu3noy6q) **4**

[**Warstwa kontrolera**](#_9bxu518vkqze) **8**

[**Warstwa widoku**](#_8eaef2zhiptq) **11**

[**Autoryzacja użytkowników oraz zabezpieczenia aplikacji**](#_az7nznsc666e) **14**

[**Deployment aplikacji**](#_lld5no6rycet) **19**

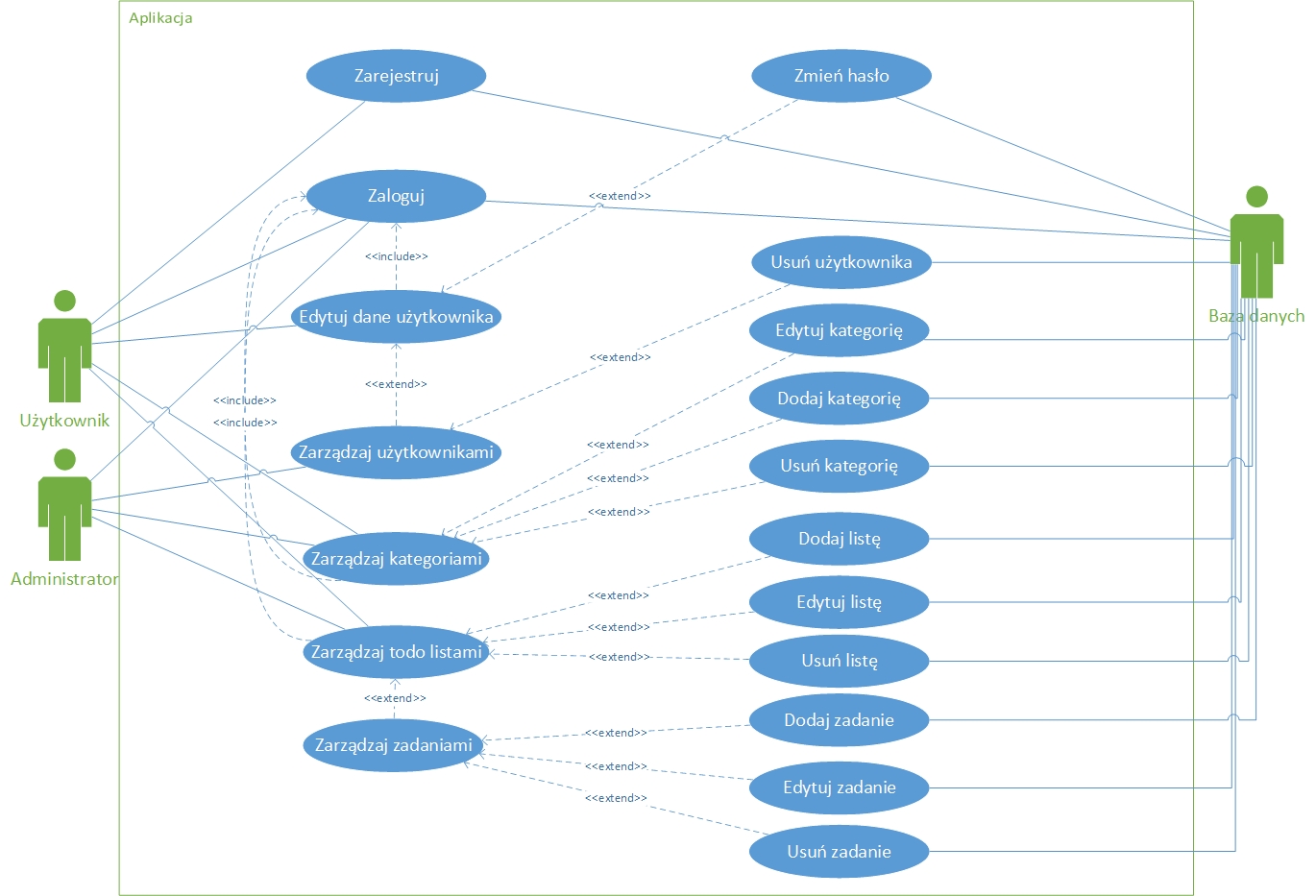
[**Bibliografia**](#_4z9cfp4u3uon) **21**

# Opis aplikacji

Niniejsza aplikacja została wykonana w ramach laboratorium z przedmiotu Aplikacje Internetowe na Politechnice Rzeszowskiej. Umożliwia ona tworzenie prostych list zadań do wykonania tzw. Todo list. Użytkownicy mogą utworzyć w niej własne konta. Po zalogowaniu mają możliwość zarządzania swoim profilem, tworzenia wspomnianych wcześniej list oraz kategorii, do których one należą.

Aplikacja oparta została o technologię .NET CORE MVC i napisana w języku C#. Do przechowywania informacji aplikacja wykorzystuje bazę danych PostgreSQL udostępnioną przez platformę Heroku. Do udostępnienia aplikacji w Internecie wykorzystano Dockera w celu przygotowania obrazu aplikacji, który następnie uruchomiany został w skonteneryzowanym środowisku zapewnionym przez platformę Heroku.

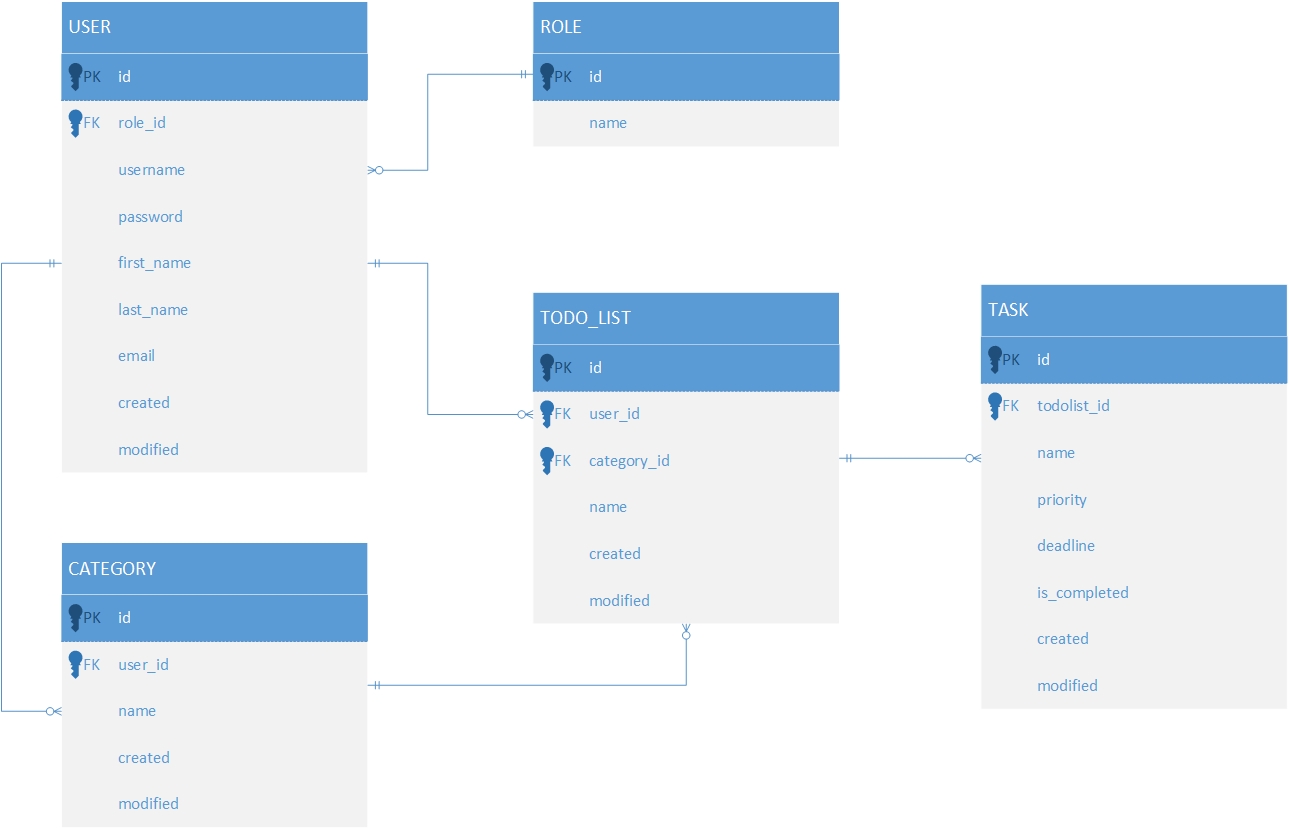
Pełny zakres funkcjonalności aplikacji zawarto na diagramie przypadków użycia z rysunku 1.1. Zaobserwować na nim można, że w systemie istniej dwa rodzaje użytkowników. Podstawowy typ użytkownika, po zalogowaniu posiada dostęp do szeregu funkcjonalności. Pierwszą z nich jest modyfikacja wprowadzonych przez niego danych. Jej rozszerzeniem jest możliwość zmiany hasła. Drugą grupę funkcjonalności stanowi zarządzanie kategoriami. Użytkownicy mogą je dodawać, edytować oraz usuwać. Analogiczny zestaw funkcjonalności dotyczy pozostałych grup tj. zarządzania listami zadań oraz samymi zadaniami. Administratorzy poza wspomnianymi już funkcjonalnościami mają również dostęp do panelu zarządzania użytkownikami. Z jego poziomu są w stanie edytować dane innych użytkowników czy ich usuwać.



Rys. 1.1 Diagram przypadków użycia dla stworzonej aplikacji

# Model danych

Pierwszym etapem tworzenia aplikacji było zaprojektowanie modelu danych, na których wykonywane będą operacje. Dane przechowywane są w relacyjnej bazie danych, która przechowuje je w postaci powiązanych ze sobą tabel. Schemat bazy danych przygotowanej w ramach niniejszej aplikacji przedstawiono w postaci diagramu ERD z rysunku 2.1.



Rys. 2.1 Diagram ERD bazy danych aplikacji

Z diagramu z rysunku 2.1 wynika, że baza danych stworzonej aplikacji składa się z pięciu tabel. Główną tabelą, lub tą która posiada najwięcej zależności, jest ta przechowująca dane o użytkownikach. Posiada ona, a tym samym i każdy z użytkowników powiązanie z odpowiednią (pojedynczą) rolą w systemie, zestawem kategorii oraz list zadań, stworzonych przez użytkownika. Każda z list musi znajdować się w konkretnej kategorii i może zawierać zestaw zadań.

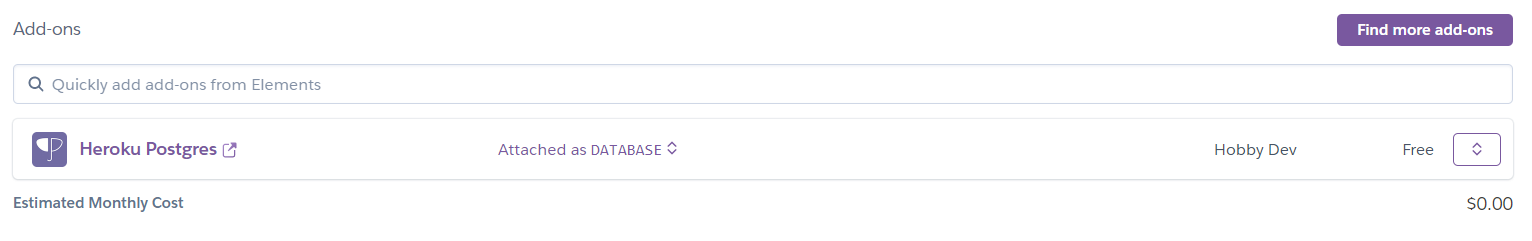
Posiadając schemat modelu przystąpiono do jego implementacji. Zastosowano podejście code first, co oznacza, że w pierwszej kolejności stworzono klasy odpowiadające encjom bazy danych, a dopiero na ich podstawie stworzono migrację, która inicjalizowała samą bazę. Proces implementacji przeprowadzono w oparciu o [1]. Przykładową implementację klasy przedstawiono na listingu 2.1.

Listing 2.1 Implementacja klasy *Category*

|  |
| --- |
| 1. [Table("category")] 2. **public** **class** Category 3. { 4. [Key] 5. [Column("id")] 6. **public** **int** Id { **get**; **set**; } 8. [Column("name")] 9. [Required] 10. [StringLength(50, ErrorMessage = "{0} length must be between {2} and {1}", MinimumLength = 3)] 11. **public** **string** Name { **get**; **set**; } 13. [Column("created")] 14. **public** DateTime Created { **get**; **set**; } 16. [Column("modified")] 17. **public** DateTime Modified { **get**; **set**; } 19. *//* 20. *// Foreign keys* 22. [ForeignKey("user\_id")] 23. [Column("user\_id")] 24. **public** **int** UserId { **get**; **set**; } 25. } |

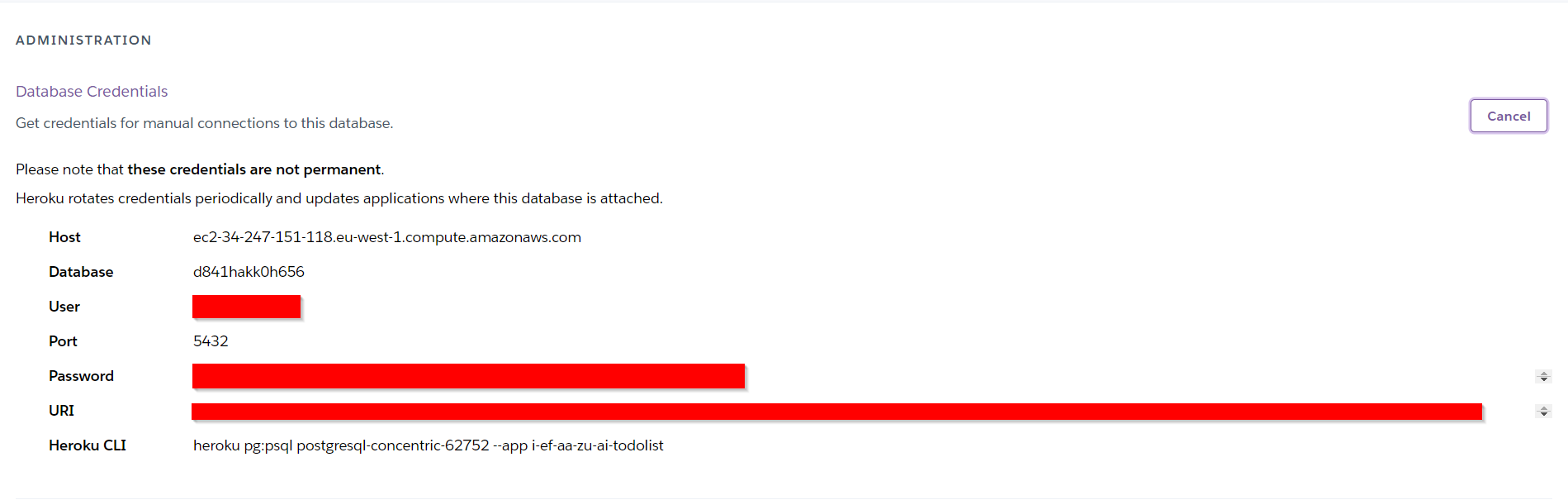
Definicję klasy poprzedzono adnotacją *Table*, wskazującą na nazwę tabeli w bazie danych. Podobnie każda z właściwości, stanowiąca odwzorowanie kolumn tabel została poprzedzona odpowiednią adnotacją. Klucze główne identyfikowane są poprzez adnotację *Key*, natomiast klucze obce *ForeignKey*. Analogicznie przygotowano wszystkie klasy modelowe.

Na aktualnym etapie przygotowany został model, jednakże nadal nie istnieje baza danych w której mógłby on zostać odwzorowany. W ramach niniejszej aplikacji została przygotowana na platformie Heroku. Do jej utworzenia niezbędne jest konto na platformie oraz utworzony projekt dla aplikacji. Będzie on wykorzystywany zarówno do deploymentu gotowej aplikacji jak i hostowania bazy danych. Proces deploymentu zostanie opisany w późniejszych rozdziałach. W celu utworzenia bazy danych wystarczy jedynie przejść do zakładki *Resources* projektu, a następnie w widocznej tam wyszukiwarce wpisać i dodać *Heroku Postgres*.



Rys. 2.2 Utworzenie bazy danych na platformie Heroku

Po utworzeniu bazy danych możliwe jest wyświetlenie informacji potrzebnych do połączenia się z nią. W tym celu należy przejść do ustawień bazy, a następnie kliknąć w View Database Credentials.



Rys 2.3 Dostęp do informacji koniecznych do połączenia z bazą danych

Posiadając zarówno model jak i instancję bazy danych należało ustanowić między nimi połączenie tj. tak zwany kontekst. Jego implementację przedstawiono na listingu 2.2. Rozszerza on klasę DbContext implementującą wszystkie niezbędnie mechanizmy wykonywania operacji na bazie danych. Właściwości kontekstu wskazują na wszystkie klasy modelu.

Listing 2.2 Kontekst połączenia z bazą danych

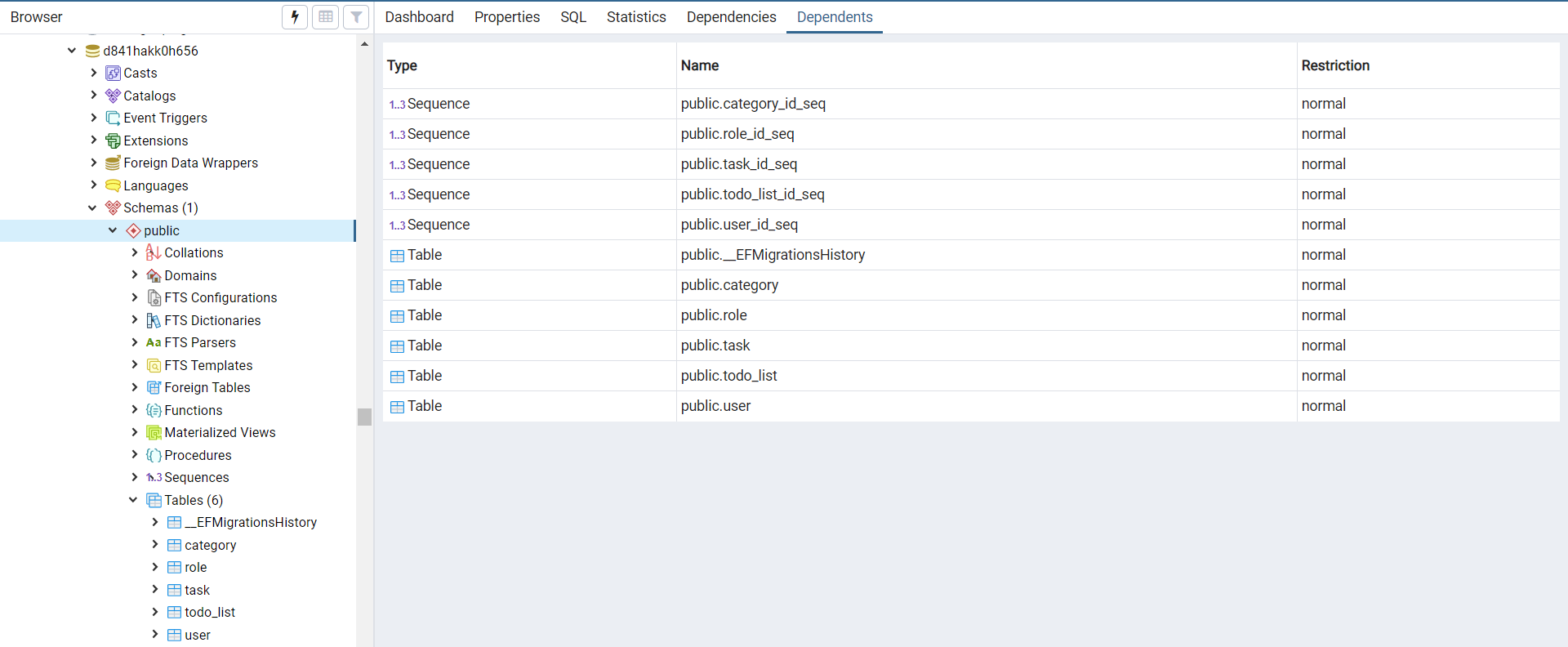
|  |
| --- |
| 1. **public** **class** TodoListDbContext : DbContext 2. { 3. **public** TodoListDbContext (DbContextOptions<TodoListDbContext> options) 4. : **base**(options) 5. { } 7. **public** DbSet<User> Users { **get**; **set**; } 8. **public** DbSet<Role> Roles { **get**; **set**; } 9. **public** DbSet<Category> Categories { **get**; **set**; } 10. **public** DbSet<TodoList> TodoLists { **get**; **set**; } 11. **public** DbSet<TodoTask> Tasks { **get**; **set**; } 12. } |

Utworzony kontekst należało następnie podpiąć w pliku z konfiguracją aplikacji. Przekazywany jest w niej również Connection string tj. Odpowiednio skonfigurowany łańcuch znaków zawierający dane konieczne do nawiązania połączenia takie jak: host i port bazy danych, jej nazwa oraz hasło. W przypadku bazy danych umieszczonej na Heroku istotne są również dodatkowe parametry wymagające ustanowienia połączenia szyfrowanego z bazą danych.

Posiadając połączenie z bazą danych, można wygenerować migrację ją inicjalizującą. W tym celu należało skorzystać z konsoli menedżera pakietów. W niej należy uruchomić dwa polecenia:

* Add-Migration <nazwa migracji> - tworzy plik migracji w katalogu Migrations, na jego podstawie będą wykonywane operacje tworzące / modyfikujące schemę bazy danych,
* Update-Database - wykonuje instrukcje zawarte w plikach migracyjnych.

W tym momencie baza danych została zainicjalizowana tj. stworzone zostały w niej tabele na podstawie klas modelowych zawartych w kontekście. Utworzone tabele przedstawiono na zrzucie ekranu z rysunku 2.4.



Rys. 2.4 Utworzone tabele w bazie danych; widok z narzędzia PgAdmin

Baza danych została utworzona jednakże nadal nie znajdują się w niej żadne dane. Do wypełnienia jej wartościami wykorzystano mechanizm seedowania. Przygotowano specjalną klasę *SeedData*, która sprawdza czy poszczególne tabele posiadają już rekordy i jeśli są one puste wypełnia je przygotowanym zestawem danych tj. obiektami odpowiednich klas modelowych. Tak przygotowaną klasę należy wykorzystać przy starcie aplikacji do inicjalizacji danych w głównej klasie aplikacji.

# Warstwa kontrolera

Do wykonywania operacji na modelu służy warstwa kontrolera. Jej zadaniem jest jest również pobieranie danych z bazy, mapowanie ich do odpowiedniej struktury, a następnie udostępnienie ich do warstwy widoku. Kontrolery obsługują również wszelkie akcje wykonywane przez użytkowników w aplikacji. Proces ich implementacji przedstawiony zostanie na przykładzie kontrolera do obsługi todo list.

Listing 3.1 Fragment klasy TodoListContorller

|  |
| --- |
| 1. **public** **class** TodoListsController : Controller 2. { 3. **private** **readonly** TodoListDbContext context; 5. **public** TodoListsController(TodoListDbContext context) 6. { 7. **this**.context = context; 8. } 10. [SignedIn] 11. **public** IActionResult Index() 12. { 13. **var** identity = HttpContext.Items["Identity"]; 14. **int** currUserId = identity != **null** ? (**int**) identity : -1; 15. **var** lists = context.TodoLists 16. .Include(t => t.Category) 17. .**Where**(t => t.UserId == currUserId); 18. **return** View(lists); 19. } 21. [SignedIn] 22. **public** IActionResult Edit(**int**? id) 23. { 24. *// ...* 25. } 27. [HttpPost] 28. [SignedIn] 29. **public** IActionResult Edit(**int**? id, [Bind("Name,CategoryId")] TodoList model) 30. { 31. **if** (ModelState.IsValid) 32. { 33. *// ...* 34. context.SaveChanges(); 35. **return** RedirectToAction(nameof(Index)); 36. } 37. **return** View(model); 38. } 39. } |

Kontrolery zawierają definicje endpointów, do których odwołują się widoki. Każdy z kontrolerów odpowiada konkretnemu modelowi bądź wydzielonej funkcjonalności aplikacji np. Autoryzacji użytkowników. Główną (podstawową) metodą kontrolera jest *Index*. Jego zadaniem jest pobranie z bazy danych potrzebnych danych, a następnie udostępnienie ich do widoku głównego, związanego z danym modelem n.p. widoku do zarządzania listami zadań. W liniach 15 - 17 listingu 3.1 pokazano sposób w jaki przy pomocy kontekstu bazy danych można filtrować istotne dane podobnie jakby to miała miejsce przy tworzeniu zapytania SQL na bazie danych. Kontekst bazy danych dołączany jest to każdego z kontrolerów jako prywatne pole tylko do odczytu inicjalizowane w konstruktorze klasy. Każdy z kontrolerów rozszerza dostępną we frameworku, specjalnie do tego przygotowaną klasę *Controller*.

Endpointy kontrolerów podzielić można na dwie kategorie. Pierwszą z nich stanowią metody obsługujące requesty HTTP typu GET służące do udostępniania danych do poszczególnych widoków. Drugą kategorię stanowią zaś metody obsługujące requesty typu POST. Przyjmują one jako parametr ciało requestu, które jest ściśle związane z danymi wyświetlanymi czy modyfikowanymi przez użytkownika w widoku. Do odebrania danych z widoku służy adnotacja Bind przedstawiona w linii 29 listingu 3.1.

Między formularzami dostępnymi w widokach a metodami kontrolera istnieje ścisłe powiązanie. Po otrzymaniu danych zebranych przy pomocy adnotacji Bind, kontroler dokonuje ich walidacji. W liniach 31 - 37 listingu 3.1 przedstawiono typowy proces przetwarzania danych w metodzie typu POST. Dane są odbierane od widoku, następnie sprawdzana jest ich poprawność w linii 31. Jeżeli przeszły one proces walidacji zapisywane są do bazy danych, a użytkownik przekierowany do innego widoku, w przeciwnym wypadku dane takie są odrzucane, a kontroler zwraca otrzymane dane do widoku wraz z informacjami o błędach walidacji.

Walidacja w .NET CORE MVC przeprowadzana jest przez kontroler na podstawie adnotacji zamieszczonych w modelu. Ich implementację wykonano w oparciu o [6]. Przykładową walidację przedstawiono na listingu 3.2 na podstawie modelu przeznaczonego do rejestracji nowych użytkowników w aplikacji.

Listing 3.2 Fragment modelu do rejestracji użytkowników wraz z adnotacjami do walidacji

|  |
| --- |
| 1. **public** **class** SignUpViewModel 2. { 3. [Required] 4. [StringLength(20, ErrorMessage = "{0} length must be between {2} and {1}", MinimumLength = 3)] 5. [RegularExpression(@"^[0-9a-zA-Z''-'\s]{1,40}$", ErrorMessage = "Special characters are not allowed")] 6. **public** **string** Username { **get**; **set**; } 8. [Required] 9. [DataType(DataType.EmailAddress)] 10. **public** **string** Email { **get**; **set**; } 12. [Required] 13. [DataType(DataType.Password)] 14. [StringLength(100, ErrorMessage = "{0} length must be between {2} and {1}", MinimumLength = 3)] 15. **public** **string** Password { **get**; **set**; } 17. [Required] 18. [Display(Name = "Repeat password")] 19. [DataType(DataType.Password)] 20. [Compare("Password")] 21. [StringLength(100, ErrorMessage = "{0} length must be between {2} and {1}", MinimumLength = 3)] 22. **public** **string** PasswordRep { **get**; **set**; } 23. } |

Framework .NET CORE MVC udostępnia wiele adnotacji do walidacji danych wprowadzanych przez użytkownika. Pierwszą z nich (linia 3 listingu 3.2) jest *Required*, mówiąca, że dane pole jest wymagane. *StringLength* (linia 4 listing 3.2) określa minimalną oraz maksymalną długość prowadzonego łańcucha znaków. *RegularExpression* (linia 5 listing 3.2) pozwala na stworzenie własnego walidatora określonego wyrażeniem regularnym. W tym konkretnym przypadku zapewnia, że we wprowadzonym tekście nie mogą znajdować się znaki specjalne. *DataType* (linia 13 listingu 3.2) pozwala na określenie typu wprowadzanych danych i zapewnia dla nich odpowiednią walidację np. typ *EmailAdress* zapewnia odpowiednią strukturę adresu mailowego, natomiast *Password* sprawia, że wprowadzane dane nie będą jawnie widoczne w formularzu. Ostatnią z użytych adnotacji jest *Compare* (linia 20 listingu 3.2), zapewnia ona, że dwa określone pola muszą mieć tą samą wartość, co przydaje się np. w przypadku konieczności powtórzenia hasła.

Nie zawsze dane dostępne w modelu są wystarczające do pełnego funkcjonowania widoku. Niezbędne jest wtedy przekazanie dodatkowych informacji bez modyfikowania klas modelowych. To tego celu służy specjalna zmienna *ViewData,* w której można umieszczać dane dowolnego typu pod różnymi identyfikatorami. Szczególnie jest to przydatne w przypadku gdy w formularzu dostępne są rozwijane listy wyboru. Dostępne w nich pozycje nie znajdują się w samym modelu dlatego też należy przygotować je w kontrolerze, a następnie umieścić w *ViewData*. Przykład użycia przedstawiono na listingu 3.3.

Listing 3.3 Przykład użycia zmiennej ViewData

|  |
| --- |
| 1. ViewData["Categories"] = context.Categories 2. .**Where**(c => c.UserId == currUserId) 3. .ToList(); |

# Warstwa widoku

Warstwa widoku stanowi graficzny interfejs użytkownika. Składają się na niego tabele, listy i inne elementy służące do prezentacji danych, jak również formularze służące do ich wprowadzania czy modyfikacji. W frameworku .NET CORE MVC do tworzenia widoków używany jest format *Razor*. Stanowi on połączenie języka znaczników HTML oraz programowania C#. Oznacza to, że poszczególne elementy widoków budowane są przy pomocy znaczników HTML, które mogą być wzbogacone o logikę napisaną w języku C#. Pozwala to na używanie pętli czy instrukcji warunkowych. Mechanizmy języka C# wykorzystywane są również do uzyskiwania dostępu do danych udostępnionych przez kontroler.

Widok główny aplikacji znajduje się w pliku *\_Layout*. W nim umieszczone są stałe elementy widoku takie jak pasek nawigacyjny czy stopka strony. Wywoływana jest w nim również metoda renderująca pod widoki przygotowywane w oddzielnych plikach.

Widoki, podobnie jak kontrolery można podzielić na dwie kategorie. Pierwszą stanowią widoki służące do prezentacji danych, drugą do ich modyfikacji tzw formularze. Przywy widok z pierwszej kategorii zostanie przedstawiony na podstawie widoku listy zadań. Fragment jej kodu przedstawiono na listingu 4.1.

Listing 4.1 Fragment widoku do zarządzania listą zadań

|  |
| --- |
| 1. <div> 2. <hr /> 3. <dl **class**="row"> 4. <dt **class**="col-sm-2"> 5. Category 6. </dt> 7. <dd **class**="col-sm-10"> 8. @Html.DisplayFor(model => model.Category.Name) 9. </dd> 10. <dt **class**="col-sm-2"> 11. @Html.DisplayNameFor(model => model.Created) 12. </dt> 13. <dd **class**="col-sm-10"> 14. @Html.DisplayFor(model => model.Created) 15. </dd> 16. <dt **class**="col-sm-2"> 17. @Html.DisplayNameFor(model => model.Modified) 18. </dt> 19. <dd **class**="col-sm-10"> 20. @Html.DisplayFor(model => model.Modified) 21. </dd> 22. </dl> 23. <hr /> 24. </div> 26. <div **class**="d-flex flex-row-reverse"> 27. <a asp-action="Index" 28. **class**="btn btn-primary ml-2">Back</a> 29. <a asp-controller="Tasks" 30. asp-action="Edit" 31. asp-route-TodoListId="@Model.Id" 32. **class**="btn btn-success">New task</a> 33. </div> 35. <div **class**="mt-3"> 36. <table **class**="table"> 37. <thead> 38. <tr> 39. <th></th> 40. <th>Name</th> 41. <th>Deadline</th> 42. <th>Priority</th> 43. <th></th> 44. </tr> 45. </thead> 46. <tbody> 47. @**foreach** (TodoTask task **in** Model.Tasks) 48. { 49. <tr **class**="@(task.IsCompleted ? "table-success" : "")"> 50. <td> 51. <form asp-controller="Tasks" asp-action="Complete" asp-route-id="@task.Id"> 52. <input asp-**for**="@task.IsCompleted" **class**="form-control todo-task-checkbox"> 53. </form> 54. </td> 55. <td>@task.Name</td> 56. <td>@task.Deadline</td> 57. <td> 58. <span **class**="badge badge-primary badge-pill">@task.Priority</span> 59. </td> 60. <td> 61. <a asp-controller="Tasks" 62. asp-action="Edit" 63. asp-route-id="@task.Id">Edit</a> | 64. <a asp-controller="Tasks" 65. asp-action="Delete" 66. asp-route-id="@task.Id">Delete</a> 67. </td> 68. </tr> 69. } 70. </tbody> 71. </table> 72. </div> |

W liniach 3 - 22 listingu 4.1 zawarta jest lista wyświetlająca dane dotyczące listy zadań. Wykorzystuje ona specjalne obiekty oraz metody udostępniane przez framework w celu wyświetlenia nazw oraz wartości poszczególnych właściwości modelu. Niżej w liniach 26 - 33 znajdują się przyciski wywołujące akcje kontrolerów przekierowujące do formularza do dodawania zadań do listy czy przekierowującego z powrotem do nadrzędnego widoku. Z kolei w liniach 36 - 71 znajduje się tabela wyświetlająca dodane do listy zadania. Wykorzystuje on pętlę foreach przechodzącą przez zadania zawarte w modelu listy. W poszczególnych wierszach tabeli wyświetlane są dane zadań oraz przyciski wywołujące akcje kontrolerów.

Nie wszystkie akcje wykonywane przez użytkownika na widoku mogą bezpośrednio wywoływać metody kontrolera. Czasami konieczne jest użycie pośredniczącego skryptu JavaScript [4]. Taka też sytuacja ma miejsce w przypadku oznaczania zadań jako wykonanych poprzez zaznaczenie pola wyboru na liście. W przeznaczonym do tego pliku site.js umieszczono skrypt tworzący listener na każdym z pól wyboru nasłuchujący wykonanie akcji ich zaznaczenia lub odznaczenia. Gdy taka sytuacja wystąpi skrypt zatwierdza formularz otaczający checkbox tym samym wywołując akcję kontrolera. Kod JavaScript przedstawiono na listingu 4.2.

Listing 4.2 Kod JavaScript nasłuchujący akcji wykonanych na polach wyboru

|  |
| --- |
| 1. $(document).ready(**function** () { 3. $('.todo-task-checkbox').on('click', **function** (e) { 4. markCompleted(e.target); 5. }); 7. }); 9. **function** markCompleted (checkbox) { 10. **var** form = checkbox.closest('form'); 11. form.submit(); 12. } |

Widoki tworzone przy pomocy jedynie standardowych elementów dostępnych w języku HTML nie są zbyt przyjemne wizualnie. Dlatego też framework .NET CORE MVC posiada wbudowane wsparcie dla biblioteki Bootstrap. Stanowi ona bibliotekę elementów graficznego interfejsu użytkownika zbudowanych na podstawie języka HTML oraz arkuszy stylów CSS. Możliwe jest więc korzystanie z gotowych rozwiązań w postaci przykładowych kodów lub klas nadających standardowym widokom nieco więcej elegancji [3].

# Autoryzacja użytkowników oraz zabezpieczenia aplikacji

Ponieważ aplikacja zapewnia możliwość tworzenia list zadań dla każdego z użytkowników z osobna, konieczna była implementacja systemu rejestracji oraz logowania do aplikacji. Formularze oraz ich obsługa zaimplementowane zostały w sposób analogiczny do opisanych w rozdziałach poświęconych warstwie widoku oraz kontrolera. Różnica polega jednak na bezpiecznym przechowywaniu wprowadzonego przez użytkownika hasła. Nie może być ono przechowywane w bazie danych w postaci jawnego tekstu, gdyż stanowiłoby to potencjalną lukę do wycieku takich informacji. Muszą one więc zostać zaszyfrowane. W tym celu wykorzystano bibliotekę *BCrypt*. Pozwala ona na wygenerowanie unikalnego hasha na podstawie wprowadzonego przez użytkownika hasła. Zaszyfrowany w ten sposób tekst może być bezpiecznie przechowywany w bazie danych. W czasie logowania hasło wprowadzone przez użytkownika jest porównywane przy użyciu biblioteki BCrypt z hashem pobranym z bazy danych. Po pomyślnej operacji porównania użytkownik zostaje zalogowany. Podstawowe informacje o nim tj. imię, nazwisko, rola w systemie zostają umieszczone w sesji.

Aby umożliwić aplikacji na korzystanie z sesji konieczne jest dołączenie do projektu dwóch pakietów: Microsoft.Extensions.Caching.Memory oraz Microsoft.AspNetCore.Session, a następnie wskazanie aplikacji, że ma korzystać z sesji w pliku konfiguracyjnym *Startup.cs* co przedstawiono na listingu 5.1 w liniach 9 - 10 [5].

Listing 5.1 Fragment pliku konfiguracyjnego Sratup.cs

|  |
| --- |
| 1. **public** **void** ConfigureServices(IServiceCollection services) 2. { 3. services.AddControllersWithViews(); 5. services.AddDbContext<TodoListDbContext>(options => 6. options.UseNpgsql(Configuration.GetConnectionString("DbConnection")) 7. ); 9. services.AddDistributedMemoryCache(); 10. services.AddSession(); 11. } |

Posiadając dane użytkownika w sesji należało przygotować mechanizm, który będzie je udostępniać do widoków, a zarazem do kontrolerów przy każdej akcji wykonanej w aplikacji. W tym celu zastosowano mechanizm zwany middleware. Pozwala on na przygotowanie logiki która wykonywana jest przy każdym żądaniu wysyłanym w aplikacji [8].

Pierwszym krokiem w jego implementacji jest przygotowanie samej logiki. Zawarto ją w specjalnie do tego przygotowanej klasie *IdentityMiddleware*. Jej kod zawarto na listingu 5.2. Jej zadaniem jest sprawdzenie czy dane użytkownika znajdują się w sesji. Jeżeli ich tam nie ma, znaczy to że użytkownik nie jest zalogowany i kontekście ustawiana jest odpowiednia flaga (listing 5.2 linie 12 - 14). W przeciwnym wypadku uzyskane dane użytkownika umieszczane są w kontekście aplikacji (listing 5.2 linie 15 - 23).

Listing 5.2 Kod middleware do obsługi danych użytkownika

|  |
| --- |
| 1. **public class IdentityMiddleware** 2. **{** 3. **private readonly RequestDelegate next;** 5. **public** IdentityMiddleware(RequestDelegate next) 6. **{** 7. **this.next = next;** 8. **}** 10. **public** **async** Task InvokeAsync(HttpContext context) 11. **{** 12. **byte[] bytes;** 13. **bool loggedIn = context.Session.TryGetValue("Identity", out bytes);** 14. **context.Items.Add("IsSignedIn", loggedIn);** 15. **if** (loggedIn) 16. **{** 17. **string identity = Encoding.UTF8.GetString(bytes);** 18. **string[] elements = identity.Split(";");** 19. **context.Items.Add("Identity", Int32.Parse(elements[0]));** 20. context.Items.**Add**("FirstName", elements[1]); 21. **context.Items.Add("LastName", elements[2]);** 22. **context.Items.Add("IsAdmin", Int32.Parse(elements[3]).Equals(RolesEnum.ADMIN\_ID));** 23. **}** 25. **await** next(context); 26. **}** 27. **}** |

Kolejnym krokiem jest opakowanie przygotowanej wcześniej middleware w klasę udostępniającą do interfejsu odpowiedzialnego za budowanie aplikacji. Umożliwi to na wskazanie aplikacji konieczności używania przygotowanego middleware z poziomu pliku konfiguracyjnego. Kod tej klasy zawarto na listingu 5.3.

Listing 5.2 Kod klasy *IdentityMiddlewareExtension*

|  |
| --- |
| 1. **public static class IdentityMiddlewareExtension** 2. **{** 3. **public static IApplicationBuilder UseIdentityMiddleware(** 4. **this IApplicationBuilder builder** 5. ) 6. **{** 7. **return builder.UseMiddleware<IdentityMiddleware>();** 8. **}** 9. **}** |

Ostatnim krokiem jest umieszczenie wywołania metody przygotowanej wcześniej klasy w pliku konfiguracyjnym aplikacji. Przedstawiono to na listingu 5.4.

Listing 5.4 Fragment pliku konfiguracyjnego aplikacji *Startup.cs*

|  |
| --- |
| 1. **public void Configure(IApplicationBuilder app, IWebHostEnvironment env)** 2. **{** 3. ***// ...*** 5. **app**.UseIdentityMiddleware(); 7. ***// ...*** 8. **}** |

Posiadając dane użytkownika w kontekście aplikacji, należało je wykorzystać do zabezpieczenia poszczególnych elementów systemu. Zostało to zrealizowane na dwa sposoby: ukrycie poszczególnych elementów widoku oraz przygotowanie filtrów zabezpieczających dostęp do metod kontrolerów.

Zabezpieczenie poszczególnych elementów widoku sprowadza się do wykorzystania instrukcji warunkowej języka C# w kodzie HTML. Dzięki temu zależnie od ustawionych flag tj. np. czy użytkownik jest zalogowany (listing 5.5 linia 2) lub posiada uprawnienia administratora (listing 5.5 lina 6), użytkownikowi wyświetlane są lub nie elementy widoku.

Listing 5.5 Fragment głównego widoku aplikacji

|  |
| --- |
| 1. **<body>** 2. **@if ((bool) ViewContext.HttpContext.Items["IsSignedIn"])** 3. **{** 4. **<header>** 5. @\* ... \*@ 6. @**if ((bool) ViewContext.HttpContext.Items["IsAdmin"])** 7. **{** 8. **<li class="nav-item">** 9. **<a asp-area=""** 10. asp-controller="Users" 11. **asp-action="Index"** 12. **class="nav-link text-dark">Users</a>** 13. **</li>** 14. **}** 15. @\* ... \*@ 16. **</header>** 17. **}** 18. **@\* ... \*@** 19. **</body>** |

Zabezpieczenie metod kontrolerów wymaga nieco więcej pracy. W tym celu należało przygotować filtry, które przed każdym żądaniem wysłanym na określoną metodę sprawdzają określone warunki np. Czy użytkownik jest zalogowany, lub odwołuje się do przypisanych mu zasobów. Implementacja filtrów omówiona zostanie na przykładzie klasy *AdminAttribute*, sprawdzającej czy użytkownik posiada uprawnienia administratora. Jej kod zawarto na listingu 5.6.

Listing 5.6 Kod filtra *Admin*, zawartego w klasie *AdminAttribute*

|  |
| --- |
| 1. **public class AdminAttribute : ResultFilterAttribute** 2. **{** 3. **public AdminAttribute()** 4. **{** 5. } 7. **public override void OnResultExecuting(ResultExecutingContext context)** 8. **{** 9. **var admin = context.HttpContext.Items["IsAdmin"];** 10. **bool** isAdmin = admin != **null** ? (**bool**) admin : **false**; 11. **if (isAdmin)** 12. **{** 13. **base.OnResultExecuting(context);** 14. **}** 15. **else** 16. **{** 17. **context.HttpContext.Response.Redirect("/Home");** 18. **}** 19. **}** 20. **}** |

Zwrócić należy uwagę na fakt, że klasy implementujące filtry muszą posiadać na końcu nazwy *Attribute*, oraz rozszerzać klasę *ResultFilterAttribute*. Następnie należy nadpisać metodę *OnResultExecuting*, umieszczając w niej logikę filtru. Jako parametr przyjmuje ona kontekst aplikacji. W filtrze z listingu 5.6 w liniach 9 - 11 sprawdzane jest czy użytkownik posiada uprawnienia administratora. Jeżeli warunek jest spełniony, żądanie jest przetwarzane. W przeciwnym wypadku następuje przekierowanie na stronę startową [7].

Filtry stosowane są jako adnotacje umieszczane nad metodami kontrolerów. Pojedyncza metoda może posiadać kilka filtrów. Wykonywane są one wtedy jeden po drugim. Każdy z warunków użytych filtrów musi zostać spełniony aby żądanie zostało obsłużone. Wykorzystanie adnotacji filtrów zaprezentowano na listingu 5.7 w liniach 3, 9 oraz 16.

Listing 5.7 Zastosowanie adnotacji filtrów na metodach kontrolera na przykładzie fragmentu klasy UserController

|  |
| --- |
| 1. **public class UsersController : Controller** 2. **{** 3. **[Admin]** 4. **public async Task<IActionResult> Index()** 5. { 6. ***// ...*** 7. **}** 9. **[CurrentUser]** 10. **public** IActionResult Details(**int** id) 11. **{** 12. ***// ...*** 13. **}** 15. [HttpPost] 16. **[CurrentUser]** 17. **public IActionResult Edit(int? id, [Bind("Username,FirstName,LastName,Email")] User model)** 18. **{** 19. ***// ...*** 20. } 21. } |

W aplikacji przygotowano następujące filtry:

* sprawdzający, czy użytkownik posiada uprawnienia administratora,
* sprawdzający, czy użytkownik jest zalogowany,
* sprawdzający, czy użytkownik jest wylogowany,
* sprawdzający, czy użytkownik żąda dostępu do własnych danych.

# Deployment aplikacji

Ostatnim etapem tworzenia aplikacji jest jej deployment. Jest to proces umieszczenia jej na odpowiednim środowisku uruchomieniowym, aby stała się ona dostępna dla użytkowników Internetu. W niniejszej aplikacji wykorzystano w tym celu platformę Heroku. Ponieważ jednak nie spiera ona bezpośrednio aplikacji opartych na frameworku .NET CORE, wykorzystano dodatkowo Dockera tj. proces konteneryzacji. Przed rozpoczęciem pracy należało przygotować niezbędne narzędzia. Pierwszym z nich jest sam Docker, dzięki któremu aplikacja zostanie zbudowana w postaci obrazu. Drugim zaś jest Heroku CLI, które umożliwi na udostępnienie zbudowanego obrazu na platformę Heroku [2].

Kolejnym etapem po przygotowaniu niezbędnych narzędzi jest stworzenie pliku Dockerfile zawierającego ciąg instrukcji dla Dockera w jaki sposób ma on zbudować obraz aplikacji. Jego zawartość przedstawiono na listingu 6.1.

Listing 6.1 Zawartość pliku Dockerfile

|  |
| --- |
| 1. **FROM mcr.microsoft.com/dotnet/sdk:5.0 AS build-env** 2. **WORKDIR /app** 4. **COPY \*.csproj ./** 5. RUN dotnet restore 7. COPY . . 8. RUN dotnet publish **-c Release -o out** 10. **FROM mcr.microsoft.com/**dotnet**/**aspnet:5.0 11. **WORKDIR /app** 12. **COPY --from=build-env /app/out .** 14. **CMD ASPNETCORE\_URLS=http://\*:$PORT dotnet todolist.dll** |

Idąc po kolei liniami kodu z listingu 6.1 omówiony zostanie proces budowania aplikacji przez Dockera. W pierwszej linii następuje załadowanie SDK koniecznego do procesu budowania aplikacji opartych na .NET CORE. Następnie w linii 2 ustawiany jest katalog roboczy na *app*, po czym kopiowany jest plik projektu aplikacji (linia 4), na podstawie którego pobierane są wszelkie wymagane pakiety (linia 5). Po skopiowaniu pozostałych plików aplikacji (linia 7) następuje jej zbudowanie (linia 8). To stanowiło pierwszy etap tj. budowanie aplikacji. Drugim etapem jest określenie sposobu w jaki ma być ona uruchamiana. W linii 10 załadowane zostaje środowisko uruchomieniowe dla aplikacji .NET CORE. Ponownie katalog roboczy ustawiany jest na *app* (linia 11) po czym skopiowane są pliki utworzone w czasie procesu budowania (linia 12). W ostatniej linii ustawiany jest dla aplikacji port, na którym ma zostać uruchomiona. Będzie on przekazywany przez platformę Heroku.

Na tym etapie proces deploymentu jest prawie zakończony ponieważ projekt na platformie HEroku dla aplikacji został stworzony w czasie tworzenia bazy danych. Pozostaje więc jedynie wykonać kilka komend udostępniających aplikację na platformę Heroku. Dla ułatwienia i możliwości wykonywania tych czynności wiele razy dobrze jest przygotować skrypt batch. Jego zawartość przedstawiono na listingu 6.2.

Listing 6.2 Skrypt udostępniający zbudowaną aplikację na platformie Heroku

|  |
| --- |
| 1. **call heroku login** 3. **call heroku container:login** 5. **call docker build** -t todolist . 7. call heroku container:push **-a i-ef-aa-zu-ai-todolist web** 9. **call heroku container:release -a i-ef-aa-zu-ai-todolist web** 11. **PAUSE** |

Podobnie jak w przypadku pliku Dockerfile, na podstawie skryptu z listingu 6.2 omówiony zostanie proces udostępniania aplikacji na platformie Heroku. W linii 1 następuje logowanie do platformy Heroku. Po wykonaniu tej komendy otwarta zostaje przeglądarka z formularzem do logowania. Skrypt zatrzymuje się wtedy oczekując na zalogowanie. Następnie następuje zalogowanie do rejestru kontenerów Heroku (linia 3). Proces ten wykonuje się automatycznie na podstawie danych wprowadzonych wcześniej przy logowaniu do platformy. W linii 5 przy użyciu Dockera budowana jest aplikacja z wykorzystaniem przygotowanego wcześniej pliku Dockerfile. Przygotowany obraz aplikacji jest uploadowany na platformę Heroku (linia 7), a następnie ustawiany jako aktualna wersja, która ma zostać uruchomiona (linia 9).

W ten sposób w kilku prostych krokach można udostępnić napisaną aplikację w Internecie. Będzie ona dostępna pod adresem udostępnionym przez Heroku. W przypadku niniejszej aplikacji jest to adres dostępny pod linkiem 6.1.

Link 6.1 Adres udostępniony aplikacji przez platformę Heroku

<https://i-ef-aa-zu-ai-todolist.herokuapp.com/>

W celach testowych stworzone zostały dwa konta użytkowników:

* zwykłego użytkownika: login: user; hasło: user,
* administratora: login: admin; hasło: admin.

# Bibliografia

[1] Anderson R. 2021 Get started with ASP.NET Core MVC <https://docs.microsoft.com/en-us/aspnet/core/tutorials/first-mvc-app/start-mvc>

[2] Robilliard A. 2020 Deploying .NET Core to Heroku <https://dev.to/alrobilliard/deploying-net-core-to-heroku-1lfe>

[3] <https://getbootstrap.com/docs/4.0/components>

[4] Barbettini N. 2018 Complete items with a checkbox, The Little ASP.NET Core Book <https://nbarbettini.gitbooks.io/little-asp-net-core-book/content/chapters/add-more-features/complete-with-checkbox.html>

[5] Lock A. 2016 An introduction to Session storage in ASP.NET Core <https://andrewlock.net/an-introduction-to-session-storage-in-asp-net-core/>

[6] Larkin K. 2019 Model validation in ASP.NET Core MVC and Razor Pages <https://docs.microsoft.com/en-us/aspnet/core/mvc/models/validation>

[7] Larkin K., Anderson R., Dykstra T., Smith S. 2020 Filters in ASP.NET Core <https://docs.microsoft.com/en-us/aspnet/core/mvc/controllers/filters>

[8] Anderson R., Smith S. 2020 Write custom ASP.NET Core middleware <https://docs.microsoft.com/en-us/aspnet/core/fundamentals/middleware/write>