Dokumentacja projektu: AutoWypożyczalniaFajna

1. Wstęp

Projekt "AutoWypożyczalniaFajna" to aplikacja internetowa, która umożliwia użytkownikom wypożyczenie samochodów w sposób prosty i efektywny. Projekt wykorzystuje framework ASP.NET Core i bazę danych SQL Server do przechowywania danych. Użytkownicy mogą przeglądać dostępne samochody, dokonywać rezerwacji oraz zarządzać swoimi danymi

Autor: Mateusz Ratajczak

Wykorzystane pakiety:

- **CoreAdmin**
- Wersja: 3.0.0
- **Microsoft.AspNetCore.Diagnostics.EntityFrameworkCore**
- Wersja: 8.0.10
- **Microsoft.AspNetCore.Identity.EntityFrameworkCore**
- Wersja: 8.0.10
- **Microsoft.AspNetCore.Identity.UI**
- Wersja: 8.0.10
- **Microsoft.EntityFrameworkCore.SqlServer**
- Wersja: 8.0.10
- **Microsoft.EntityFrameworkCore.Tools**
- Wersja: 8.0.11

2. Instrukcja uruchomienia

2.1 Wymagania wstępne

Aby uruchomić projekt, musisz mieć zainstalowane:

- .NET SDK 8.0 lub wyższy (wymagane do uruchomienia aplikacji)
- SQL Server (lub inny serwer bazy danych kompatybilny z EF Core)
- Visual Studio (lub inne IDE wspierające .NET Core)

2.2 Kroki do uruchomienia projektu

Otwórz projekt.

W pliku appsettings.json skonfiguruj połączenie do bazy danych SQL Server. *(opcjonalne)*

W konsoli menedżera pakietów użyj polecenia “update-database”.

Uruchom projekt.

3. Opis struktury projektu

Wykorzystano model architektoniczny: MVC

Projekt składa się z następujących głównych katalogów i plików:

3.1 Pliki konfiguracyjne

- **appsettings.json** – Główne ustawienia aplikacji (np. połączenie z bazą danych).
- **appsettings.Development.json** – Ustawienia specyficzne dla środowiska deweloperskiego.
- **launchSettings.json** – Ustawienia uruchomieniowe dla różnych środowisk (np. IIS Express, Kestrel).

3.2 Główne katalogi i pliki

- **AutoWypozyczalniaFajna.csproj** – Główny plik projektu, który zawiera informacje o zależnościach i konfiguracji projektu.
- **Program.cs** – Punkt wejścia aplikacji, konfiguracja serwera i aplikacji.
- **Properties/** – Katalog z dodatkowymi plikami konfiguracyjnymi projektu.
- **bin/** oraz **obj/** – Katalogi wygenerowane po kompilacji aplikacji.

3.3 Warstwy aplikacji

- **Controllers/** – Katalog zawierający kontrolery MVC odpowiedzialne za logikę aplikacji. Znajdziesz tu pliki, takie jak:
  - **HomeController.cs**
  - **MarkasController.cs**
  - **SamochodsController.cs**
  - **TypPaliwasController.cs**
  - **WypozyczeniesController.cs**
- **Models/** – Katalog z klasami modeli danych, np.:
  - **Marka.cs**
  - **Samochod.cs**
  - **TypPaliwa.cs**
  - **Wypozyczenie.cs**
- **Data/** – Katalog z plikami dotyczącymi dostępu do danych, m.in.:
  - **ApplicationDbContext.cs** – Kontekst bazy danych, zawierający definicje tabel.
  - **Migrations/** – Folder z migracjami bazy danych.

- **Views/** – Katalog z widokami Razor, które odpowiadają za interfejs użytkownika. Zawiera podkatalogi takie jak:
    - **Home/** – Widoki związane z główną stroną.
    - **Markas/**, **Samochods/**, **TypPaliwas/**, **Wypozyczenies/** – Widoki związane z danymi i formularzami dotyczącymi tych encji.
    - **Shared/** – Widoki wspólne dla aplikacji, jak layouty, formularze błędów.
- **wwwroot/** – Katalog z zasobami statycznymi (np. CSS, JavaScript, obrazy).
- **Validators/** – Katalog z klasami walidatorów, np. **GreaterThanAttribute.cs**, które służą do walidacji danych wejściowych w formularzach.

4. Opis systemu użytkowników

Projekt obsługuje różnych użytkowników, z różnymi poziomami dostępu. Poniżej znajduje się opis komponentów i funkcji dostępnych dla różnych typów użytkowników.

4.1 Użytkownicy systemu

- **Gość** – Użytkownik, który nie jest zalogowany. Ma dostęp do ogólnych informacji o samochodach, ale nie może rezerwować pojazdów.
- **Zalogowany użytkownik** – Może przeglądać dostępne samochody, wypożyczać pojazdy, zarządzać swoimi rezerwacjami.
- **Administrator** – Ma pełny dostęp do systemu, w tym możliwość zarządzania samochodami, markami, typami paliw oraz przeglądania danych użytkowników.

4.2 System logowania i rejestracji

- Użytkownicy mogą rejestrować się i logować za pomocą systemu tożsamości ASP.NET Core Identity.
- W systemie jest dostępna opcja resetowania hasła i zarządzania profilami użytkowników.
- Projekt posiada wbudowane konto Administratora:
  Email: admin@admin.com
  Hasło: P@ssw0rd

5. Opis modeli

1. Wypozyczenie

Model reprezentuje wypożyczenie samochodu przez użytkownika.

- **Id**: Unikalny identyfikator wypożyczenia.

- **SamochodId**: Klucz obcy wskazujący na samochód wypożyczony w ramach tego wypożyczenia.
- **Samochod**: Nawigacja do modelu Samochod, wskazuje na samochód, który jest wypożyczony.
- **DataWypozyczenia**: Data wypożyczenia samochodu, oznaczona jako wymagana.
- **DataZwrotu**: Data planowanego zwrotu samochodu, z walidacją sprawdzającą, czy jest późniejsza od daty wypożyczenia.
- **CenaCalkowita**: Całkowity koszt wypożyczenia, prawdopodobnie obliczany na podstawie ceny za dzień oraz długości okresu wypożyczenia.
- **UserId**: Identyfikator użytkownika (z systemu IdentityUser), który wypożyczył samochód.
- **User**: Nawigacja do modelu IdentityUser, reprezentująca użytkownika systemu.

2. TypPaliwa

Model reprezentuje typ paliwa, w którym samochód może pracować.

- **Id**: Unikalny identyfikator typu paliwa.
- **Paliwo**: Nazwa typu paliwa, z walidacją długości (maksymalnie 50 znaków).

3. Samochod

Model reprezentuje samochód dostępny w wypożyczalni.

- **Id**: Unikalny identyfikator samochodu.
- **MarkaId**: Klucz obcy wskazujący na markę samochodu, powiązany z modelem Marka.
- **Marka**: Nawigacja do modelu Marka, reprezentująca markę samochodu.
- **Model**: Nazwa modelu samochodu, z walidacją długości (maksymalnie 50 znaków).
- **TypPaliwaId**: Klucz obcy wskazujący na typ paliwa, powiązany z modelem TypPaliwa.
- **TypPaliwa**: Nawigacja do modelu TypPaliwa, określająca, jaki typ paliwa jest używany przez samochód.
- **NrRejestracyjny**: Numer rejestracyjny samochodu, z walidacją poprawności formatu (3-10 znaków alfanumerycznych).
- **CenaZaDzien**: Cena za wynajem samochodu na jeden dzień, z walidacją zakresu od 1 do 10 000 PLN.
- **IsAvailable**: Flaga dostępności samochodu, domyślnie ustawiona na true, oznacza, że samochód jest dostępny do wypożyczenia.

4. ErrorViewModel

Model, który jest używany do przedstawiania błędów w widokach, szczególnie przy wyświetlaniu stron błędów.

- **RequestId**: Identyfikator żądania, jeśli jest dostępny, w celu śledzenia problemów w aplikacji.
- **ShowRequestId**: Właściwość, która zwraca true, jeśli RequestId nie jest pusty, co oznacza, że można go wyświetlić w interfejsie.

5. Marka

Model reprezentuje markę samochodów dostępnych w wypożyczalni.

- **Id**: Unikalny identyfikator marki.
- **MarkaNazwa**: Nazwa marki samochodu, z walidacją długości (maksymalnie 50 znaków).

Podsumowanie

- **Wypozyczenie**: Obsługuje informacje o wypożyczeniu samochodu, powiązane z samochodem i użytkownikiem.
- **TypPaliwa**: Przechowuje typy paliw dostępne w systemie.
- **Samochod**: Reprezentuje samochody dostępne w wypożyczalni, z różnymi szczegółami, takimi jak marka, model, paliwo, numer rejestracyjny, cena za dzień i dostępność.
- **ErrorViewModel**: Używane do wyświetlania błędów i informacji o błędach w widokach.
- **Marka**: Przechowuje informacje o markach samochodów dostępnych w systemie.

1. Kontroler wypożyczeni:

- WypozyczeniesController jest kontrolerem, który odpowiada za logikę aplikacji dotyczącą wypożyczeń samochodów. Jest to kontroler z atrybutem [Authorize], co oznacza, że dostęp do jego akcji mają tylko zalogowani użytkownicy.
- Konstruktor przyjmuje obiekt ApplicationDbContext, który jest używany do komunikacji z bazą danych, pozwalając na wykonywanie operacji CRUD (tworzenie, odczyt, aktualizacja, usuwanie) na danych.

2. Akcje

Kontroler oferuje różne akcje (metody), które obsługują różne operacje związane z wypożyczeniami:

a. Index (GET):

- Wyświetla listę wypożyczeń przypisanych do aktualnie zalogowanego użytkownika.

- Pobiera wszystkie wypożyczenia, sprawdzając, czy data zwrotu już minęła, i jeśli tak, aktualizuje dostępność samochodu w bazie danych.

b. Create (GET):

- Umożliwia stworzenie nowego wypożyczenia.
- Przekazuje użytkownikowi listę dostępnych samochodów, które nie są aktualnie wypożyczone, a także ceny tych samochodów.
- Dodatkowo w ViewData przekazuje dane do wyświetlenia w formularzu, jak lista samochodów i użytkowników.

c. Create (POST):

- Zapisuje nowe wypożyczenie do bazy danych.
- Przypisuje cenę całkowitą na podstawie liczby dni wypożyczenia i ceny dziennej samochodu.
- Zmienia status dostępności samochodu na false (niedostępny).

d. Edit (GET):

- Umożliwia edycję istniejącego wypożyczenia.
- Sprawdza, czy użytkownik ma prawo edytować dane wypożyczenie (czy należy do niego).
- Wyświetla formularz z danymi wypożyczenia, umożliwiając wybór samochodu i użytkownika.

e. Edit (POST):

- Aktualizuje dane wypożyczenia w bazie danych.
- Jeśli zmienia się data zwrotu, aktualizuje także całkowitą cenę wypożyczenia na podstawie nowych danych.

f. Details (GET):

- Wyświetla szczegóły danego wypożyczenia.
- Przekazuje dane wypożyczenia, w tym powiązane informacje o samochodzie i użytkowniku.
- Sprawdza, czy wypożyczenie należy do aktualnie zalogowanego użytkownika, jeśli nie — blokuje dostęp.

g. Details (POST):

- Obsługuje możliwość edytowania szczegółów wypożyczenia. Umożliwia m.in. zmianę daty zwrotu, co wiąże się z przeliczeniem ceny całkowitej.

h. Delete (GET):

- Wyświetla stronę, na której użytkownik może usunąć dane wypożyczenie.
- Przed usunięciem sprawdzany jest właściciel wypożyczenia, aby tylko użytkownik, który je stworzył, miał prawo do jego usunięcia.

i. Delete (POST):

- Usuwa wypożyczenie z bazy danych, a także przywraca dostępność samochodu do puli dostępnych pojazdów.
- Usunięcie wypożyczenia wiąże się z aktualizacją statusu dostępności samochodu w bazie danych.

3. Sprawdzanie uprawnień użytkownika

W każdej akcji, która ma na celu manipulację danymi (np. edytowanie lub usuwanie wypożyczenia), kontroler sprawdza, czy wypożyczenie należy do aktualnie zalogowanego użytkownika. Jeśli nie, dostęp do danej akcji jest zabroniony (Forbid()).

4. Obsługa błędów

Kontroler obsługuje sytuacje, w których operacje na danych mogą się nie powieść (np. przy próbie edycji lub usunięcia wypożyczenia, które już zostało zmienione przez innego użytkownika). W takich przypadkach wyświetlane są odpowiednie komunikaty błędów.

5. Widoki

Kontroler używa ViewData i ViewBag do przekazywania danych do widoków, które są odpowiedzialne za renderowanie formularzy i wyświetlanie danych na stronach.

6. Wykorzystanie technologii

- **Entity Framework (EF)**: Do komunikacji z bazą danych i wykonywania operacji CRUD.
- **ASP.NET Core MVC**: Struktura aplikacji pozwalająca na tworzenie aplikacji webowych z modelami, widokami i kontrolerami.

W kodzie przedstawiony jest kontroler TypPaliwasController odpowiedzialny za zarządzanie typami paliw w systemie. Kontroler działa w kontekście aplikacji opartej na ASP.NET Core,

która wykorzystuje wzorzec MVC oraz Entity Framework do operacji na bazie danych. Jest on dostępny tylko dla użytkowników z rolą "Administrator" dzięki atrybutowi [Authorize(Roles = "Administrator")]. Oto szczegółowy opis funkcji kontrolera:

1. Kontroler paliw:

- W konstruktorze kontrolera wstrzykiwany jest kontekst aplikacji ApplicationDbContext, który umożliwia interakcję z bazą danych.

2. Akcja Index:

- **Opis**: Wyświetla listę wszystkich dostępnych typów paliw z bazy danych.
- **Metoda**: public async Task<IActionResult> Index()
- **Działanie**: Pobiera wszystkie rekordy z tabeli TypPaliwa i przekazuje je do widoku.

3. Akcja Details:

- **Opis**: Wyświetla szczegóły wybranego typu paliwa.
- **Metoda**: public async Task<IActionResult> Details(int? id)
- **Działanie**: Pobiera jeden rekord typu paliwa na podstawie jego id. Jeśli rekord nie istnieje, zwraca błąd 404.

4. Akcja Create (GET):

- **Opis**: Przedstawia formularz do tworzenia nowego typu paliwa.
- **Metoda**: public IActionResult Create()
- **Działanie**: Renderuje widok formularza, w którym użytkownik może wprowadzić nazwę nowego typu paliwa.

5. Akcja Create (POST):

- **Opis**: Obsługuje zapis nowego typu paliwa do bazy danych.
- **Metoda**: public async Task<IActionResult> Create([Bind("Id,Paliwo")] TypPaliwa typPaliwa)
- **Działanie**: Sprawdza poprawność danych w formularzu. Jeśli są one poprawne, zapisuje nowy rekord w tabeli TypPaliwa. Następnie przekierowuje do akcji Index.

6. Akcja Edit (GET):

- **Opis**: Wyświetla formularz edycji dla wybranego typu paliwa.
- **Metoda**: public async Task<IActionResult> Edit(int? id)
- **Działanie**: Pobiera wybrany rekord typu paliwa na podstawie id i renderuje formularz z jego danymi.

7. Akcja Edit (POST):

- **Opis**: Zapisuje zmiany w istniejącym typie paliwa.
- **Metoda**: public async Task<IActionResult> Edit(int id, [Bind("Id,Paliwo")] TypPaliwa typPaliwa)
- **Działanie**: Sprawdza, czy dane w formularzu są poprawne i czy rekord o danym id istnieje. Zaktualizowane dane są zapisywane w bazie, a następnie użytkownik zostaje przekierowany do widoku Index.

8. Akcja Delete (GET):

- **Opis**: Wyświetla formularz potwierdzenia usunięcia typu paliwa.
- **Metoda**: public async Task<IActionResult> Delete(int? id)
- **Działanie**: Pobiera jeden rekord typu paliwa na podstawie jego id i renderuje widok, który pozwala na potwierdzenie chęci usunięcia.

9. Akcja Delete (POST):

- **Opis**: Usuwa wybrany rekord typu paliwa z bazy danych.
- **Metoda**: public async Task<IActionResult> DeleteConfirmed(int id)
- **Działanie**: Usuwa rekord typu paliwa o wskazanym id i zapisuje zmiany w bazie danych. Po usunięciu użytkownik zostaje przekierowany do widoku Index.

10. Metoda pomocnicza TypPaliwaExists:

- **Opis**: Sprawdza, czy w bazie danych istnieje rekord typu paliwa o podanym id.
- **Metoda**: private bool TypPaliwaExists(int id)
- **Działanie**: Używa metody Any z Entity Framework do sprawdzenia, czy dany rekord istnieje w tabeli TypPaliwa.

Wymogi:

- Kontroler wykorzystuje rolę Administrator do zabezpieczenia dostępu do tych operacji, co oznacza, że tylko użytkownicy z tą rolą mogą tworzyć, edytować, usuwać i przeglądać typy paliw.

Ten kod przedstawia kontroler SamochodsController, który jest odpowiedzialny za zarządzanie samochodami w aplikacji AutoWypozyczalniaFajna. Podobnie jak poprzedni kontroler, ten również wykorzystuje wzorzec MVC oraz Entity Framework. Poniżej znajduje się szczegółowy opis funkcji tego kontrolera:

1. Kontroler Samochodów:

- W konstruktorze kontrolera wstrzykiwany jest kontekst aplikacji ApplicationDbContext, który umożliwia interakcję z bazą danych i wykonywanie operacji CRUD na tabeli Samochod.

2. Akcja Index:

- **Opis**: Wyświetla listę wszystkich samochodów, uwzględniając dane o marce i typie paliwa.
- **Metoda**: public async Task<IActionResult> Index()
- **Działanie**: Pobiera dane samochodów z bazy, w tym związane z nimi informacje o marce (Marka) oraz typie paliwa (TypPaliwa), a następnie przekazuje je do widoku.

3. Akcja Details:

- **Opis**: Wyświetla szczegóły wybranego samochodu.
- **Metoda**: public async Task<IActionResult> Details(int? id)
- **Działanie**: Sprawdza, czy id jest przekazane, a następnie pobiera szczegóły wybranego samochodu, w tym powiązaną markę i typ paliwa. Jeśli samochód o danym id nie istnieje, zwraca błąd 404.

4. Akcja Create (GET):

- **Opis**: Renderuje formularz do tworzenia nowego samochodu.
- **Metoda**: public IActionResult Create()
- **Działanie**: Udostępnia formularz, w którym użytkownik może dodać nowy samochód. Zawiera listy rozwijane do wyboru marki oraz typu paliwa.

5. Akcja Create (POST):

- **Opis**: Tworzy nowy samochód w bazie danych.
- **Metoda**: public async Task<IActionResult> Create([Bind("Id,MarkaId,Model,TypPaliwaId,NrRejestracyjny,CenaZaDzien")] Samochod samochod)
- **Działanie**: Jeśli dane są poprawne, zapisuje nowy samochód w bazie i przekierowuje do akcji Index. W przypadku błędu formularza, ponownie renderuje widok z danymi formularza i listami rozwijanymi.

6. Akcja Edit (GET):

- **Opis**: Wyświetla formularz edycji istniejącego samochodu.
- **Metoda**: public async Task<IActionResult> Edit(int? id)

- **Działanie**: Pobiera istniejący samochód na podstawie przekazanego id, a następnie renderuje formularz edycji. Zawiera aktualne dane oraz listy rozwijane dla marki i typu paliwa.

7. Akcja Edit (POST):

- **Opis**: Zapisuje zmiany w edytowanym samochodzie.
- **Metoda**: public async Task<IActionResult> Edit(int id, [Bind("Id,MarkaId,Model,TypPaliwaId,NrRejestracyjny,CenaZaDzien")] Samochod samochod)
- **Działanie**: Sprawdza, czy dane w formularzu są poprawne i czy id w formularzu odpowiada temu w bazie. Jeśli wszystko jest poprawne, zapisuje zmiany w bazie danych i przekierowuje do widoku Index. Jeśli wystąpi błąd, ponownie renderuje formularz edycji.

8. Akcja Delete (GET):

- **Opis**: Wyświetla formularz do potwierdzenia usunięcia samochodu.
- **Metoda**: public async Task<IActionResult> Delete(int? id)
- **Działanie**: Sprawdza, czy id jest przekazane, a następnie wyświetla szczegóły samochodu do usunięcia. Jeśli samochód o danym id nie istnieje, zwraca błąd 404.

9. Akcja Delete (POST):

- **Opis**: Usuwa wybrany samochód z bazy danych.
- **Metoda**: public async Task<IActionResult> DeleteConfirmed(int id)
- **Działanie**: Usuwa samochód o danym id z bazy i zapisuje zmiany. Po usunięciu samochodu przekierowuje do widoku Index.

10. Metoda pomocnicza SamochodExists:

- **Opis**: Sprawdza, czy samochód o danym id istnieje w bazie danych.
- **Metoda**: private bool SamochodExists(int id)
- **Działanie**: Używa metody Any z Entity Framework, aby sprawdzić, czy dany rekord samochodu istnieje w tabeli Samochod.

Zabezpieczenia:

- **Role Administratora**: Wszystkie akcje, które umożliwiają tworzenie, edytowanie i usuwanie samochodów, są chronione przez atrybut [Authorize(Roles = "Administrator")]. Oznacza to, że tylko użytkownicy posiadający rolę Administrator mogą wykonywać te operacje.
- **Walidacja danych**: Każda z akcji Create i Edit sprawdza poprawność danych za pomocą ModelState.IsValid, co zapobiega zapisaniu niepoprawnych danych.

Podsumowanie:

Kontroler SamochodsController pozwala na pełne zarządzanie samochodami w systemie. Administratorzy mogą dodawać, edytować, usuwać i wyświetlać szczegóły samochodów. Kontroler zapewnia interakcję z tabelami Marka i TypPaliwa, umożliwiając łatwą obsługę danych powiązanych z samochodami.

Kontroler MarkasController jest odpowiedzialny za zarządzanie markami samochodów w systemie AutoWypozyczalniaFajna. Działa w sposób podobny do poprzednich kontrolerów, implementując wzorzec MVC z pełną funkcjonalnością CRUD (tworzenie, odczyt, edytowanie i usuwanie). Oto szczegółowe omówienie jego funkcji:

1. Kontroler Marek:

- W konstruktorze kontrolera wstrzykiwany jest kontekst aplikacji ApplicationDbContext, który pozwala na interakcję z bazą danych.

2. Akcja Index:

- **Opis**: Wyświetla listę wszystkich marek samochodów.
- **Metoda**: public async Task<IActionResult> Index()
- **Działanie**: Pobiera wszystkie marki z bazy danych i przekazuje je do widoku w celu wyświetlenia.

3. Akcja Details:

- **Opis**: Wyświetla szczegóły wybranej marki.
- **Metoda**: public async Task<IActionResult> Details(int? id)
- **Działanie**: Sprawdza, czy id zostało przekazane. Jeśli nie, zwraca błąd. Następnie pobiera markę o danym id i wyświetla jej szczegóły. Jeśli marka nie istnieje, zwraca błąd 404.

4. Akcja Create (GET):

- **Opis**: Renderuje formularz do tworzenia nowej marki.
- **Metoda**: public IActionResult Create()
- **Działanie**: Renderuje pusty formularz, w którym użytkownik może wprowadzić nazwę nowej marki.

5. Akcja Create (POST):

- **Opis**: Tworzy nową markę i zapisuje ją w bazie danych.
- **Metoda**: public async Task<IActionResult> Create([Bind("Id,MarkaNazwa")] Marka marka)

- **Działanie**: Jeśli dane w formularzu są poprawne, dodaje nową markę do bazy i zapisuje zmiany. Następnie przekierowuje do widoku Index, który wyświetli zaktualizowaną listę marek.

6. Akcja Edit (GET):

- **Opis**: Wyświetla formularz edycji istniejącej marki.
- **Metoda**: public async Task<IActionResult> Edit(int? id)
- **Działanie**: Pobiera markę na podstawie przekazanego id i wyświetla formularz z jej aktualnymi danymi. Jeśli marka o danym id nie istnieje, zwraca błąd 404.

7. Akcja Edit (POST):

- **Opis**: Zapisuje zmiany w edytowanej marce.
- **Metoda**: public async Task<IActionResult> Edit(int id, [Bind("Id,MarkaNazwa")] Marka marka)
- **Działanie**: Sprawdza, czy dane w formularzu są poprawne oraz czy id w formularzu odpowiada temu w bazie. Jeśli wszystko jest poprawnie, zapisuje zmiany i przekierowuje do widoku Index. Jeśli wystąpi błąd, ponownie renderuje formularz edycji.

8. Akcja Delete (GET):

- **Opis**: Wyświetla formularz potwierdzenia usunięcia marki.
- **Metoda**: public async Task<IActionResult> Delete(int? id)
- **Działanie**: Pobiera markę do usunięcia na podstawie przekazanego id. Wyświetla formularz z danymi marki, aby użytkownik mógł potwierdzić jej usunięcie. Jeśli marka o danym id nie istnieje, zwraca błąd 404.

9. Akcja Delete (POST):

- **Opis**: Usuwa wybraną markę z bazy danych.
- **Metoda**: public async Task<IActionResult> DeleteConfirmed(int id)
- **Działanie**: Usuwa markę o danym id z bazy danych i zapisuje zmiany. Po usunięciu przekierowuje do widoku Index.

10. Metoda pomocnicza MarkaExists:

- **Opis**: Sprawdza, czy marka o danym id istnieje w bazie danych.
- **Metoda**: private bool MarkaExists(int id)
- **Działanie**: Używa metody Any z Entity Framework, aby sprawdzić, czy marka o danym id istnieje w tabeli Marka.

Zabezpieczenia:

- **Role Administratora**: Cały kontroler jest zabezpieczony atrybutem [Authorize(Roles = "Administrator")], co oznacza, że tylko użytkownicy z rolą Administrator mogą wykonać operacje tworzenia, edytowania, usuwania marek.
- **Walidacja danych**: Każda z akcji Create i Edit sprawdza poprawność danych za pomocą ModelState.IsValid, co zapobiega zapisaniu niepoprawnych danych.

Podsumowanie:

Kontroler MarkasController umożliwia administratorom zarządzanie markami samochodów. Administratorzy mogą tworzyć, edytować, usuwać i przeglądać szczegóły marek. Całość jest zabezpieczona przed nieautoryzowanym dostępem przez mechanizm ról w ASP.NET Core, który wymusza rolę Administrator do wykonywania operacji administracyjnych.

1. Home Kontroler:

- Konstruktor kontrolera przyjmuje obiekt ILogger<HomeController>, który jest używany do logowania zdarzeń i błędów w aplikacji.
- ILogger umożliwia rejestrowanie logów w aplikacji, co jest pomocne w diagnostyce i monitorowaniu.

2. Akcja Index:

- **Opis**: Wyświetla stronę główną aplikacji.
- **Metoda**: public IActionResult Index()
- **Działanie**: Zwraca widok, który odpowiada za stronę główną aplikacji. Strona główna może zawierać np. powitanie lub ogólne informacje o aplikacji.

3. Akcja Privacy:

- **Opis**: Wyświetla stronę polityki prywatności.
- **Metoda**: public IActionResult Privacy()
- **Działanie**: Zwraca widok, który wyświetla politykę prywatności aplikacji, tj. zasady przetwarzania danych użytkowników, politykę cookies itd.

4. Akcja Error:

- **Opis**: Wyświetla stronę błędu w przypadku wystąpienia jakiejkolwiek awarii aplikacji.
- **Metoda**: public IActionResult Error()

- **Działanie**: Metoda ta jest zabezpieczona przez atrybut [ResponseCache], który wyłącza buforowanie odpowiedzi dla strony błędu (aby zawsze wyświetlała aktualny błąd). Tworzy obiekt ErrorViewModel, który przekazuje informacje o bieżącym identyfikatorze żądania (np. pomocne w logowaniu błędów lub śledzeniu problemów w aplikacji).

Atrybut [ResponseCache]:

- **Duration = 0**: Wyłącza buforowanie odpowiedzi.
- **Location = ResponseCacheLocation.None**: Oznacza brak buforowania.
- **NoStore = true**: Nie przechowuje żadnej informacji w pamięci podręcznej.

5. Model ErrorViewModel:

- Zawiera właściwość RequestId, która jest używana do śledzenia identyfikatora żądania. Jest to przydatne, gdy chcesz powiązać błąd z konkretnym żądaniem, aby łatwiej było go zdiagnozować.

Podsumowanie:

Kontroler HomeController jest jednym z podstawowych kontrolerów aplikacji ASP.NET Core, który obsługuje akcje związane z prezentacją strony głównej, polityki prywatności oraz obsługą błędów. Ma on za zadanie zapewnić użytkownikowi podstawową nawigację po stronie aplikacji, a także umożliwić wyświetlanie odpowiednich informacji w przypadku błędów, które mogą wystąpić podczas korzystania z aplikacji.

Kontener ApplicationDbContext jest kluczowym elementem w aplikacji ASP.NET Core, który zarządza dostępem do bazy danych, zapewniając mechanizm komunikacji pomiędzy aplikacją a systemem bazodanowym. Jest to rozszerzenie klasy IdentityDbContext, która jest częścią frameworka ASP.NET Core Identity i umożliwia zarządzanie tożsamościami użytkowników (np. logowanie, rejestracja, role, itp.).

Opis klasy ApplicationDbContext:

1. **Dziedziczenie po IdentityDbContext**:
    a. Klasa ApplicationDbContext dziedziczy po IdentityDbContext, co pozwala na integrację z systemem tożsamości użytkowników (Identity). Dzięki temu aplikacja może wykorzystywać funkcjonalności takie jak logowanie, rejestracja, zarządzanie użytkownikami oraz przypisywanie ról użytkownikom.
    b. IdentityDbContext automatycznie zapewnia zestaw tabel w bazie danych, które przechowują dane o użytkownikach, ich rolach, hasłach itp.
2. **Konstruktor klasy**:
    a. Konstruktor przyjmuje parametry typu DbContextOptions<ApplicationDbContext>, które zawierają ustawienia

konfiguracji dla bazy danych. Wywołanie base(options) przekazuje te ustawienia do klasy bazowej (IdentityDbContext), umożliwiając jej skonfigurowanie połączenia z bazą danych.

3. **DbSet dla encji**:
    a. DbSet<Marka> Marka: Reprezentuje zbiór danych przechowywanych w tabeli Marka. Tabela ta zawiera informacje o markach samochodów w systemie.
    b. DbSet<TypPaliwa> TypPaliwa: Reprezentuje zbiór danych przechowywanych w tabeli TypPaliwa, który zawiera informacje o typach paliw dostępnych w systemie.
    c. DbSet<Samochod> Samochod: Reprezentuje zbiór danych przechowywanych w tabeli Samochod, który przechowuje informacje o samochodach dostępnych w wypożyczalni.
    d. DbSet<Wypozyczenie> Wypozyczenie: Reprezentuje zbiór danych przechowywanych w tabeli Wypozyczenie, która zawiera informacje o wypożyczeniach samochodów (np. data wypożyczenia, klient, cena).
4. **Metoda OnModelCreating**:
    a. Metoda ta jest nadpisana w celu skonfigurowania dodatkowych szczegółów dotyczących bazy danych. Wywołanie base.OnModelCreating(modelBuilder) jest ważne, ponieważ zapewnia prawidłową konfigurację tabel systemu tożsamości (takich jak AspNetUsers, AspNetRoles, AspNetUserRoles, itp.).
    b. W tej metodzie można dodać dodatkową konfigurację dla poszczególnych tabel i właściwości encji, np. ustalając typy danych dla pól.
5. **Konfiguracja dla encji Samochod i Wypozyczenie**:
    a. **Samochod**:
        i. Dla właściwości CenaZaDzien zastosowano konfigurację .HasColumnType("decimal(18,2)"), co oznacza, że cena za dzień wynajmu będzie przechowywana jako liczba dziesiętna o precyzji 18 cyfr i 2 miejscach po przecinku (np. 100.50).
    b. **Wypozyczenie**:
        i. Dla właściwości CenaCalkowita zastosowano podobną konfigurację .HasColumnType("decimal(18,2)"), co gwarantuje, że całkowita cena wypożyczenia będzie przechowywana z precyzją do 2 miejsc po przecinku.
        ii. Ponieważ ta właściwość pojawia się dwukrotnie w konfiguracji (co jest najprawdopodobniej niezamierzonym duplikatem), można ją uprościć do jednej definicji.
6. **Możliwość rozszerzenia**:
    a. Można tu dodać inne niestandardowe konfiguracje modelu, takie jak ustalanie relacji między tabelami, definicje indeksów, czy też szczegółowe ustawienie ograniczeń dla danych (np. unikalność, wymagane pola, długość tekstów, itp.).

Podsumowanie:

Klasa ApplicationDbContext zarządza dostępem do bazy danych i obsługuje encje związane z aplikacją, takie jak Marka, TypPaliwa, Samochod i Wypozyczenie. Dzięki dziedziczeniu po

IdentityDbContext, integruje system tożsamości użytkowników, umożliwiając zarządzanie użytkownikami i ich rolami. Dodatkowo, w metodzie OnModelCreating przeprowadzona jest konfiguracja precyzyjnego typu danych dla pól, które przechowują wartości liczbowe, takie jak ceny. To zapewnia lepszą kontrolę nad strukturą bazy danych oraz optymalizację przechowywanych danych.

System Użytkowników w Projekcie AutoWypozyczalniaFajna

Projekt **AutoWypozyczalniaFajna** wykorzystuje system zarządzania tożsamościami oparty na technologii **ASP.NET Core Identity**, który zapewnia rejestrację, logowanie, zarządzanie sesjami użytkowników oraz autoryzację. System użytkowników jest odpowiedzialny za obsługę ról, takich jak Administrator i użytkownicy standardowi, a także umożliwia zarządzanie danymi użytkowników.

Kluczowe komponenty systemu użytkowników:

1. **IdentityUser**:
    a. **IdentityUser** jest domyślną klasą dostarczoną przez **ASP.NET Core Identity**, która reprezentuje użytkownika w systemie. Zawiera informacje takie jak nazwa użytkownika, adres e-mail, hasło oraz inne metadane związane z użytkownikiem (np. identyfikator).
2. **SignInManager<IdentityUser>**:
    a. **SignInManager** odpowiada za zarządzanie logowaniem i sesjami użytkowników. Dzięki niemu możliwe jest:
        i. Logowanie użytkowników.
        ii. Zatrzymywanie sesji po wylogowaniu.
        iii. Sprawdzanie, czy użytkownik jest zalogowany (metoda IsSignedIn).
3. **UserManager<IdentityUser>**:
    a. **UserManager** zarządza użytkownikami w systemie, zapewniając funkcje takie jak:
        i. Tworzenie, edytowanie i usuwanie użytkowników.
        ii. Zarządzanie hasłami użytkowników.
        iii. Zarządzanie rolami, co pozwala przypisywać użytkownikom odpowiednie uprawnienia (np. Administrator, Użytkownik).
        iv. Sprawdzanie tożsamości i weryfikacja użytkownika.

Interfejs użytkownika (HTML):

W sekcji nawigacyjnej (navbar) znajduje się kod odpowiedzialny za dynamiczne wyświetlanie linków nawigacyjnych w zależności od stanu zalogowania użytkownika:

1. **Jeśli użytkownik jest zalogowany**:
    a. Wyświetlana jest opcja powitania użytkownika oraz link do zarządzania kontem: Hello @User.Identity?.Name!.

    b. Użytkownik ma również możliwość wylogowania się za pomocą formularza i przycisku **Logout**.
2. **Jeśli użytkownik nie jest zalogowany**:
    a. Wyświetlane są opcje rejestracji (link do strony rejestracji) oraz logowania (link do strony logowania).

Opis funkcji:

1. **Rejestracja (/Account/Register)**:
    a. Użytkownicy mogą stworzyć nowe konto, wprowadzając swoje dane (np. adres e-mail, hasło).
    b. Po pomyślnej rejestracji użytkownik może zalogować się przy użyciu swoich danych.
2. **Logowanie (/Account/Login)**:
    a. Zalogowani użytkownicy mogą uzyskać dostęp do aplikacji, podając swoje dane logowania.
    b. Po pomyślnym logowaniu, użytkownik zostaje przekierowany do strony głównej lub wcześniej odwiedzonej strony.
3. **Zarządzanie kontem (/Account/Manage/Index)**:
    a. Użytkownicy, którzy są zalogowani, mają dostęp do sekcji zarządzania swoim kontem, gdzie mogą zmieniać dane, takie jak adres e-mail, hasło itp.
4. **Wylogowanie**:
    a. Użytkownicy mogą wylogować się z aplikacji. Po wylogowaniu, użytkownik zostaje przekierowany z powrotem na stronę główną.

Role użytkowników:

- **Administrator**:
    - Administratorzy mają pełne uprawnienia do zarządzania samochodami, markami, typami paliwa, wypożyczeniami oraz użytkownikami w aplikacji.
    - Administratorzy mogą tworzyć, edytować, usuwać samochody, marki i typy paliwa.
- **Użytkownicy standardowi**:
    - Użytkownicy mają możliwość przeglądania dostępnych samochodów do wypożyczenia i składania wypożyczeń, ale nie mają dostępu do funkcji zarządzania danymi samochodów, markami itp.

Panel administratora
Dla panelu Administratora używamy pakietu core-admin. Tworzy on ładny panel administratora, do którego dostęp trzeba jednak zabezpieczyć.
Więcej informacji:
https://github.com/edandersen/core-admin