Uniwersytet Gdański

Dokumentacja Backendu projektu “Awantura o kasę”

13.01.2025

1. Wykorzystane technologie

Aplikacja backendowa została zbudowana z użyciem nowoczesnych narzędzi, frameworków i bibliotek, które są aktualnie wykorzystywane w wielu projektach aplikacji webowych.

Język programowania

Aplikacja została napisana w języku C#, który jest wykorzystany do tworzenia aplikacji na platformie .NET. Zapewnia wsparcie dla rozwoju usług API, rozwiązań chmurowych, aplikacji multiplatformmowych czy też zwykłych aplikacji konsolowych.

Framework

1. .NET 8 Web API

Używany do budowy aplikacji webowych i usług RESTful API. Umożliwia łatwe tworzenie, testowanie i udostępnianie punktów końcowych w aplikacji.

Narzędzia deweloperskie

1. Visual Studio

Zintegrowane środowisko programistyczne wykorzystywane do pisania, testowania i debugowania aplikacji.

2. Docker i Docker Desktop

Docker jest platformą do konteneryzacji aplikacji, umożliwiającą tworzenie i uruchamianie aplikacji w izolowanych środowiskach zwanych kontenerami. Docker Desktop to aplikacja na systemy operacyjne Windows i macOS, która ułatwia zarządzanie kontenerami na lokalnych maszynach deweloperskich.

3. SQL Server Management Studio (SSMS) 20

Narzędzie do zarządzania bazami danych Microsoft SQL Server. Używane do administracji bazą danych oraz wykonywania zapytań SQL.

Biblioteki

1. SignalR

Używany do obsługi komunikacji w czasie rzeczywistym w aplikacji. Dzięki SignalR, aplikacja może realizować funkcje takie jak powiadomienia push, chat, aktualizacje na żywo, które wymagają natychmiastowego przekazania wiadomości czy też interakcji z użytkownikiem.

2. Entity Framework

Obiektowo-relacyjny mapper (ORM) dla platformy .NET, który umożliwia wygodne mapowanie obiektów C# na tabele w bazie danych, a także ułatwia operacje CRUD.

3. AutoMapper

Biblioteka służąca do mapowania obiektów pomiędzy różnymi typami. Używana głównie do automatycznego przekształcania danych, np. mapowanie obiektów DTO (Data Transfer Objects) na modele domenowe aplikacji.

4. Microsoft.AspNetCore.Authentication.JwtBearer

Biblioteka umożliwiająca integrację z systemem autoryzacji opartym na tokenach JWT (JSON Web Token). Używana do zabezpieczania API poprzez autentykację użytkowników.

5. Microsoft.AspNetCore.Identity.EntityFrameworkCore

Umożliwia łatwe zarządzanie użytkownikami i rolami w aplikacji przy użyciu Entity Framework Core. Jest częścią systemu ASP.NET Core Identity, który pozwala na autentykację i autoryzację użytkowników.

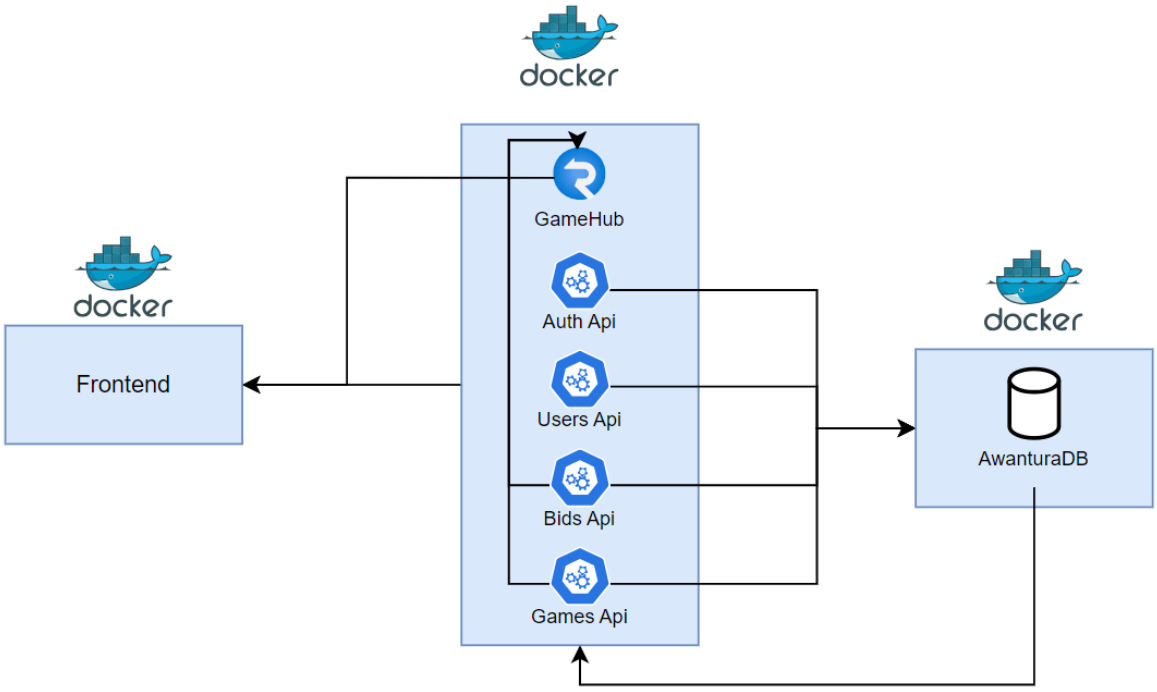
6. Microsoft.EntityFrameworkCore.Design oraz Microsoft.EntityFrameworkCore.Tools

Zawiera narzędzia i biblioteki wspierające rozwój aplikacji bazujących na Entity Framework Core, umożliwiające generowanie migracji, daje narzędzia do projektowania bazy danych oraz generuje kod bazy danych na podstawie kodu aplikacji.

7. Microsoft.IdentityModel.Tokens

Biblioteka wykorzystywana do obsługi tokenów JWT, w tym walidacji i deserializacji tokenów w celu uwierzytelniania i autoryzacji użytkowników.

2. Diagram architektury



Rys 1. Diagram architektury aplikacji.

Aplikacja została zaprojektowana z wykorzystaniem konteneryzacji, co pozwala na łatwe zarządzanie środowiskiem uruchomieniowym, skalowalność oraz izolację poszczególnych komponentów systemu. Wszystkie elementy aplikacji zostały zamknięte w kontenerach dockerowych.

Główna logika aplikacji została zaimplementowana w ramach jednego kontenera, w którym znajdują się wszystkie usługi API. Każda usługa pełni określoną rolę:

- GameHub
  - Wykorzystuje bibliotekę SignalR do komunikacji w czasie rzeczywistym między serwerem a frontendem.
  - Odpowiada za przesyłanie powiadomień o dołączeniu gracza do gry czy też czy gracz jest gotowy rozpocząć nową rundę .
- Auth API:
  - Obsługuje procesy autoryzacji i autentykacji użytkowników.
  - Wykorzystuje tokeny JWT do bezpiecznego uwierzytelniania i autoryzowania dostępu do zasobów.
- Users API:
  - Zajmuje się zarządzaniem danymi użytkowników, takimi jak tworzenie profili, aktualizacje danych, wyświetlanie listy użytkowników oraz usuwanie użytkowników.
- Bids API:

    - Odpowiada za procesy licytacyjne w grze.
    - Obsługuje składanie ofert, weryfikację zasad licytacji oraz aktualizacje związane z bieżącymi licytacjami.
- Games API
    - Służy do zarządzania rozgrywkami.
    - Umożliwia tworzenie nowych gier, dołączanie graczy oraz monitorowanie stanu rozgrywki, w tym zmian takich jak rozpoczęcie gry, przebieg rundy i zakończenie rozgrywki.

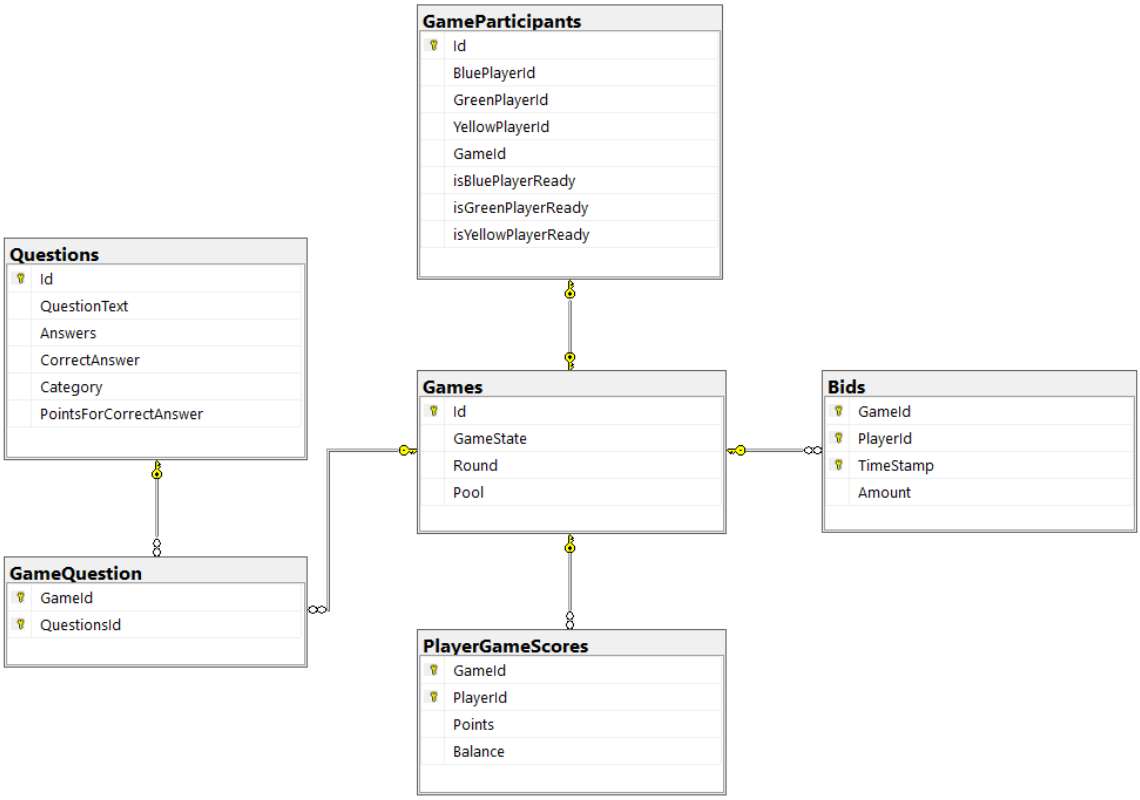
Aplikacja korzysta z dedykowanego kontenera, w którym działa serwer bazy danych – AwanturaDB.

- Baza danych przechowuje informacje o użytkownikach, rozgrywkach, licytacjach oraz innych kluczowych elementach systemu.
- API komunikuje się z bazą danych, aby uzyskiwać i aktualizować dane w czasie rzeczywistym

3. Baza danych

Na poniższym rysunku (Rys.2) zostały przedstawione tabele wykorzystywane do tworzenia rozgrywki.

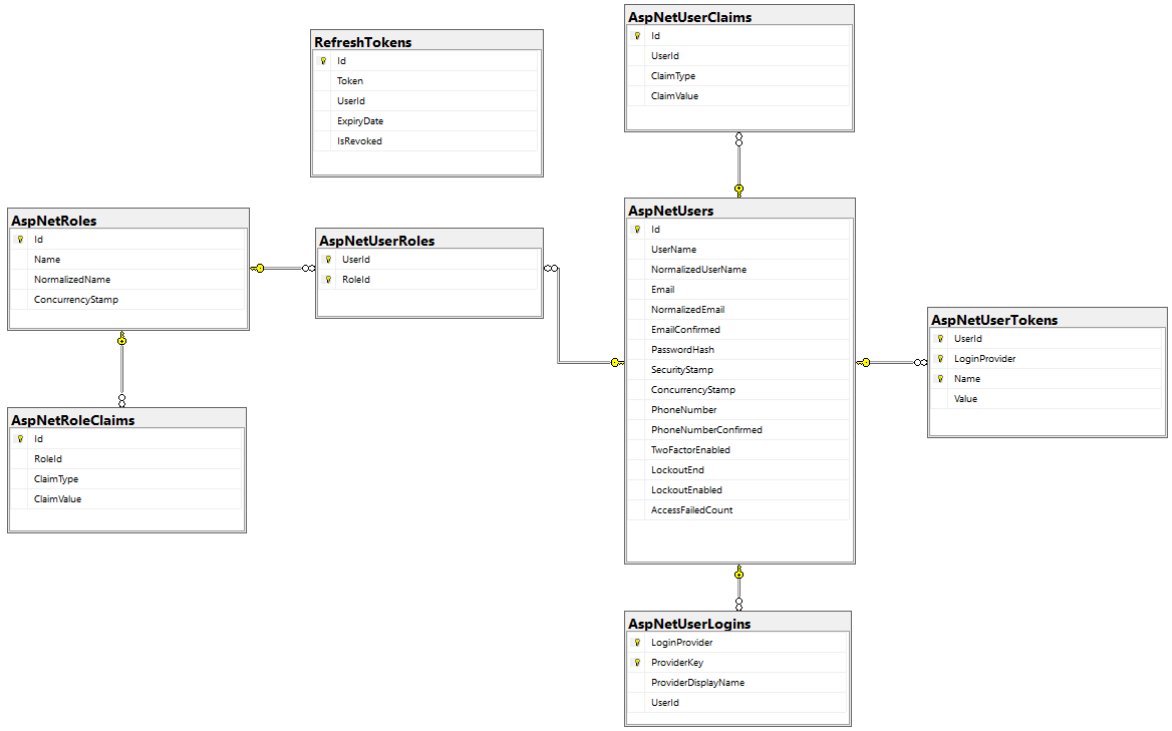
- Questions – tabela przechowująca pytania wraz z odpowiedziami, kategorią, punktami za dobrą odpowiedź oraz wskazuje poprzez indeks która odpowiedź jest prawidłowa.
- Games – tabela przechowuje najważniejsze informacje o stanie rozgrywki, jej id, numer rundy, aktualną pulę pieniędzy oraz numer rozgrywanej rundy.
- GameQuestions – tabela umożliwia stworzenie relacji wielu do wielu między pytaniami i rozgrywkami, dzięki czemu pytania mogą należeć do różnych rozgrywek.
- GameParticipants – tabela przechowuje informację o graczach w rozgrywce. Mamy trzech dostępnych graczy, blue, green i yellow. Również posiada informację czy gracz zgłosił swoją gotowość do gry.
- PlayerGameScores – tabela przechowuje informację o puli pieniędzy i numerze rundy dla konkretnej rozgrywki oraz gracza.
- Bids- tabela przechowuje informację potrzebne do licytacji tj. który gracz zaoferował największą kwotę w licytacji pytania.



Rys 2. Diagram relacji pomiędzy tabelami związanymi z rozgrywką.

Na poniższym rysunku (Rys.3) zostały przedstawione tabele wygenerowane przez bibliotekę Microsoft.AspNetCore.Identity.EntityFrameworkCore, które odpowiadają za:

- AspNetUserClaims – przechowywania rozszerzeń związanych z rolami,
- AspNetUsers - przechowywanie informacji o zarejestrowanych użytkownikach,
- AspNetUserTokens – przechowywanie tokenów,
- AspNetUserRoles - określenie, które role są przypisane do konkretnych użytkowników,
- AspNetRoles – przechowywanie ról, które mogą być przypisane użytkownikom,
- AspNetRoleClaims – przechowywanie roszczeń związanych z rolami,
- AspNetUserLogins – przechowywanie informacji o zewnętrznych dostawcach logowania przypisanych do konkretnych użytkowników.



Rys 3. Diagram relacji pomiędzy tabelami związanymi z autoryzacją i autentykacją użytkownika.

4. Opis funkcjonalności rozgrywki

Funkcjonalności związane z zarządzaniem rozgrywką w grze zostały zaimplementowane za pomocą kontrolera GamesController oraz repozytorium GameRepository. Te elementy odpowiadają za logikę związaną z tworzeniem gier, dołączaniem graczy, monitorowaniem stanu gry oraz interakcjami graczy w czasie rzeczywistym.

GamesController

Kontroler odpowiada za obsługę żądań HTTP związanych z zarządzaniem rozgrywkami. Wykorzystuje on repozytoria użytkowników i gier oraz mechanizm mapowania danych za pomocą AutoMapper. Główne akcje kontrolera to:

1. Tworzenie nowej gry (CreateNewGame)
    a. Umożliwia administratorom i graczom utworzenie nowej rozgrywki.
    b. Sprawdza tożsamość użytkownika na podstawie identyfikatora w tokenie uwierzytelniającym.
    c. Wywołuje metodę repozytorium odpowiedzialną za utworzenie gry i inicjalizację danych.

2. Dołączanie do gry (JoinGame)
    a. Pozwala graczom na dołączenie do istniejącej gry na podstawie jej identyfikatora.
    b. Obsługuje przypadki takie jak brak miejsca w grze lub sytuację, gdy gracz już uczestniczy w danej grze.
    c. Wysyła komunikat do grupy SignalR o nowym uczestniku rozgrywki oraz, jeśli wszyscy gracze są gotowi, rozpoczyna rozgrywkę.

3. Pobranie szczegółów gry (GetGame)
    a. Zwraca informacje o bieżącym stanie gry, takie jak numer rundy, kategorie pytań, stan uczestników oraz aktualny etap rozgrywki.
    b. Dane są zwracane w zależności od stanu gry, np. w stanie licytacji gracz widzi tylko kategorię pytań, a w stanie zadawania pytania – treść pytania i możliwe odpowiedzi.

4. Oznaczenie gracza jako gotowego (SetPlayerReady)
    a. Rejestruje gotowość gracza do rozpoczęcia gry.
    b. Po oznaczeniu wszystkich graczy jako gotowych, zmienia stan gry na licytację i dodaje środki do puli gry.
    c. Wysyła powiadomienia w czasie rzeczywistym do graczy za pomocą SignalR.

5. AnswerQuestion:
    a. Odpowiedzialna za obsługę odpowiedzi na pytanie zadane w trakcie gry przez gracza.
    b. Jeżeli gracz podał dobra odpowiedź zwrócona zostanie wartość true wraz z odpowiednim komunikatem.

GameRepository

Repozytorium odpowiada za bezpośrednią interakcję z bazą danych oraz zaimplementowaną logikę gry. Wykorzystuje ono kontekst Entity Framework oraz mechanizm komunikacji w czasie rzeczywistym za pomocą SignalR. Kluczowe metody repozytorium to:

1. CreateNewGame
    a. Tworzy nową grę z losowym zestawem pytań, inicjalizuje jej uczestników oraz stan początkowy.
    b. Każdy gracz rozpoczyna z początkowym saldem i punktami.

```
public async Task<Guid> CreateNewGame(Guid playerId)
{
  var gameId = Guid.NewGuid();
  var questions = await _context.Questions
    .OrderBy(q => Guid.NewGuid())
    .Take(7)
    .ToListAsync();

  var game = new Game
  {
    Id = gameId,
    GameState = GameState.NotStarted,
    Pool = 0,
    Round = 0,
    Questions = questions,
    PlayerScores = new List<PlayerGameScore>
    {
      new PlayerGameScore
      {
        GameId = gameId,
        PlayerId = playerId,
        Points = 0,
        Balance = 10000
      }
    },
    GameParticipants = new GameParticipants
    {
      Id = Guid.NewGuid(),
      BluePlayerId = playerId,
      isBluePlayerReady = false,
      GreenPlayerId = null,
      isGreenPlayerReady = false,
      YellowPlayerId = null,
      isYellowPlayerReady = false,
      GameId = gameId
    }
  };

  _context.Games.Add(game);
  await _context.SaveChangesAsync();
```

```
  return gameId;
}
```

2. AddPlayerToGame
    a. Dodaje nowego gracza do istniejącej gry, wypełniając wolne miejsce wśród uczestników.
    b. W przypadku dołączenia ostatniego gracza, uruchamia pierwszą rundę i wysyła odpowiednie powiadomienia SignalR o jej rozpoczęciu.

```
public async Task<CustomMessageResult> AddPlayerToGame(Guid gameId, PlayerDto
player)
{
  var game = await _context.Games
    .Include(g => g.GameParticipants)
    .Include(g => g.Questions)
    .FirstOrDefaultAsync(g => g.Id == gameId);

  if (game == null)
  {
    return new CustomMessageResult
    {
      Success = false,
      Message = "Game not found."
    };
  }

  var participants = game.GameParticipants;

  if (participants.BluePlayerId == player.Id || participants.GreenPlayerId == player.Id ||
participants.YellowPlayerId == player.Id)
  {
    return new CustomMessageResult
    {
      Success = true,
      Message = "Player is already in the game."
    };
  }

  if (participants.GreenPlayerId == null)
```

```
    participants.GreenPlayerId = player.Id;
else if (participants.YellowPlayerId == null)
    participants.YellowPlayerId = player.Id;
else
{
    return new CustomMessageResult
    {
        Success = false,
        Message = "No available slots in the game."
    };
}

var playerGameScore = new PlayerGameScore
{
    GameId = gameId,
    PlayerId = player.Id,
    Points = 0,
    Balance = 10000
};

_context.PlayerGameScores.Add(playerGameScore);

var isStartingGame = player.Id == participants.YellowPlayerId;
if (isStartingGame)
{
    game.Round++;
    game.GameState = GameState.CATEGORY_DRAW;
}

await _context.SaveChangesAsync();

PlayerJoinedDto playerJoinedEvent = new PlayerJoinedDto
{
    Id = player.Id,
    Username = player.UserName,
    PlayerColor = participants.GreenPlayerId == player.Id ? "green" : "yellow"
};

await SendEventToGameGroup(gameId, "PlayerJoined", playerJoinedEvent);

if (isStartingGame)
```

```
  {
    var currentQuestion = GetCurrentQuestion(game);
    RoundStartedDto roundStartedDto = new RoundStartedDto
    {
      RoundNumber = game.Round,
      Category = currentQuestion.Category
    };
    await SendEventToGameGroup(gameId, "RoundStarted", roundStartedDto);
  }

  return new CustomMessageResult
  {
    Success = true,
    Message = "Player added to the game successfully."
  };
}
```

3. GetGame
    a. Pobiera szczegółowe dane dotyczące konkretnej gry, uwzględniając stan gry, rundę, pytania oraz stan uczestników.
    b. W zależności od stanu gry, dostosowuje zakres zwracanych danych (np. w stanie „NotStarted” zwraca tylko informacje o uczestnikach).

```
public async Task<GameInfoDto> GetGame(Guid gameId, string playerId)
{
  var game = await _context.Games
    .Include(g => g.GameParticipants)
    .Include(g => g.Questions)
    .Include(g => g.PlayerScores)
    .FirstOrDefaultAsync(g => g.Id == gameId);

  if (game == null)
    return null;

  var participants = game.GameParticipants;
  var playerGuid = new Guid(playerId);
```

```
  if (participants.BluePlayerId != playerGuid && participants.GreenPlayerId !=
playerGuid && participants.YellowPlayerId != playerGuid)
    return null;

  var currentQuestion = GetCurrentQuestion(game);

  var gameBids = await GetGameBids(gameId);

  if (game.GameState == GameState.NotStarted)
  {
    return new GameInfoDto
    {
      Id = game.Id,
      Round = game.Round,
      GameState = game.GameState,
      Category = null,
      Question = null,
      Answers = null,
      GameParticipants = game.GameParticipants,
      Pool = game.Pool,
      PlayerGameScores = game.PlayerScores.ToList(),
      Bids = gameBids
    };
  }
  else if (game.GameState == GameState.CATEGORY_DRAW)
  {
    return new GameInfoDto
    {
      Id = game.Id,
      Round = game.Round,
      GameState = game.GameState,
      Category = currentQuestion.Category,
      Question = null,
      Answers = null,
      GameParticipants = game.GameParticipants,
      Pool = game.Pool,
      PlayerGameScores = game.PlayerScores.ToList(),
      Bids = gameBids
    };
  }
  else if (game.GameState == GameState.BIDDING)
```

```
  {
    return new GameInfoDto
    {
      Id = game.Id,
      Round = game.Round,
      GameState = game.GameState,
      Category = currentQuestion.Category,
      Question = null,
      Answers = null,
      GameParticipants = game.GameParticipants,
      Pool = game.Pool,
      PlayerGameScores = game.PlayerScores.ToList(),
      Bids = gameBids
    };
  }
  else if (game.GameState == GameState.QUESTION || game.GameState ==
GameState.FINISHED)
  {
    return new GameInfoDto
    {
      Id = game.Id,
      Round = game.Round,
      GameState = game.GameState,
      Category = currentQuestion.Category,
      Question = currentQuestion.QuestionText,
      Answers = currentQuestion.Answers.Split(";").ToList(),
      GameParticipants = game.GameParticipants,
      Pool = game.Pool,
      PlayerGameScores = game.PlayerScores.ToList(),
      Bids = gameBids
    };
  }

  throw new ArgumentException("Wrong Game State!");
}

public async Task<bool> SetPlayerReady(Guid gameId, Guid playerId)
{
  var game = await _context.Games
    .Include(g => g.GameParticipants)
    .Include(g => g.PlayerScores)
```

```
    .FirstOrDefaultAsync(g => g.Id == gameId);

  if (game == null)
    return false;

  if (game.GameParticipants.BluePlayerId == playerId)
    game.GameParticipants.isBluePlayerReady = true;
  else if (game.GameParticipants.GreenPlayerId == playerId)
    game.GameParticipants.isGreenPlayerReady = true;
  else if (game.GameParticipants.YellowPlayerId == playerId)
    game.GameParticipants.isYellowPlayerReady = true;
  else
    return false;

  var areAllPlayersReady = game.GameParticipants.isBluePlayerReady &&
game.GameParticipants.isGreenPlayerReady &&
game.GameParticipants.isYellowPlayerReady;
  var newRoundBids = new List<Bid>();
  if (areAllPlayersReady)
  {
    game.GameState = GameState.BIDDING;

    foreach (var playerScore in game.PlayerScores)
    {
      var amountToBid = playerScore.Balance >= 500 ? 500 : playerScore.Balance;
      playerScore.Balance -= amountToBid;
      game.Pool += amountToBid;
      var newBid = new Bid
      {
        GameId = gameId,
        PlayerId = playerScore.PlayerId,
        Amount = amountToBid,
        TimeStamp = DateTime.UtcNow
      };
      _context.Bids.Add(newBid);
      newRoundBids.Add(newBid);
    }
  }
  await _context.SaveChangesAsync();

  await SendEventToGameGroup(gameId, "PlayerReady", playerId);
```

```
  if (areAllPlayersReady)
  {
    var StartBiddingEventDto = new StartBiddingEventDto
    {
      Pool = game.Pool,
      PlayerGameScores = game.PlayerScores.ToList(),
      Bids = newRoundBids
    };
    await SendEventToGameGroup(gameId, "StartBidding", StartBiddingEventDto);
  }
  return true;
}
```

4. SetPlayerReady
    a. Ustawia stan gotowości gracza w rozgrywce.
    b. Po oznaczeniu wszystkich graczy jako gotowych, przełącza grę do stanu licytacji oraz aktualizuje pule gry, odejmując odpowiednią kwotę z salda każdego uczestnika.
    c. Powiadamia uczestników gry o gotowości za pomocą SignalR.

5. AnswersQuestions
    a. Pobiera stan gry i sprawdza czy gracz, który próbuje odpowiedzieć na pytanie jest zwycięzcą licytacji.
    b. Pobierane jest aktualne pytanie i sprawdzana jest opowiedź gracza.
    c. Jeżeli odpowiedź jest poprawna aktualizuję pulę gracza o pulę która jest aktualnie w grzę.
    d. Jeżeli zostało odpowiedziane na wszystkie 7 pytań to stan gry ustawiany jest na “FINISHED”.

```
public async Task<CustomMessageResult> AnswerQuestion(Guid gameId, Guid
playerId, int answerIndex)
{
  var game = await _context.Games
    .Include(g => g.PlayerScores)
    .Include(g => g.Questions)
    .Include(g => g.GameParticipants)
    .FirstOrDefaultAsync(g => g.Id == gameId);
  if (game == null)
  {
```

```
    return new CustomMessageResult()
    {
      Success = false,
      Message = "Game not found."
    };
  }
  if (game.GameState != GameState.QUESTION)
  {
    return new CustomMessageResult()
    {
      Success = false,
      Message = "You cannot answer the question in this game phase."
    };
  }
  var highestBid = await GetHighestBid(gameId);
  if (highestBid == null)
  {
    return new CustomMessageResult()
    {
      Success = false,
      Message = "No bids found for the game. Can't determine winner of bidding."
    };
  }
  if (highestBid.PlayerId != playerId)
  {
    return new CustomMessageResult()
    {
      Success = false,
      Message = "You cannot answer the question. You didn't win bidding."
    };
  }
  var currentQuestion = GetCurrentQuestion(game);
  if (currentQuestion == null)
  {
    return new CustomMessageResult()
    {
      Success = false,
      Message = "Couldn't find a question for current round of game. Probably error
occured when creating game."
    };
  }
```

```
  var isAnswerCorrect = currentQuestion != null && currentQuestion.CorrectAnswer ==
answerIndex;
  var playerScore = game.PlayerScores.Where(p => p.PlayerId ==
playerId).FirstOrDefault();
  if (playerScore == null)
  {
    return new CustomMessageResult()
    {
      Success = false,
      Message = "Couldn't find a playerScore for user account. Probably error occured
when creating game."
    };
  }
  if (isAnswerCorrect)
  {
    playerScore.Balance += game.Pool;
    game.Pool = 0;
  }
  game.GameState = game.Round == 7 ? GameState.FINISHED :
GameState.CATEGORY_DRAW;
  if (game.Round < 7){
    game.Round++;
  }
  CleanBidsForGame(gameId);
  ResetGameReadiness(game);
  await _context.SaveChangesAsync();
  var newQuestion = GetCurrentQuestion(game);
  var questionAnswerEventDto = new QuestionAnswerEventDto()
  {
    AnsweringPlayerId = playerId,
    NewPool = game.Pool,
    NewAccountBalance = playerScore.Balance,
    IsAnswerCorrect = isAnswerCorrect,
    NewQuestionCategory = newQuestion.Category
  };
  await SendEventToGameGroup(gameId, "QuestionAnswer",
questionAnswerEventDto);
  return new CustomMessageResult()
  {
    Success = true,
    Message = "Question answer processed correctly."
```

```
  };
}
```

Repozytorium wykorzystuje mechanizm SignalR do przesyłania informacji w czasie rzeczywistym do klientów (frontendu). Dzięki temu gracze są natychmiast informowani o zmianach w grze, takich jak:

- Dołączenie nowego uczestnika,
- Rozpoczęcie nowej rundy,
- Osiągnięcie stanu gotowości przez wszystkich graczy.

BidsController

1. MakeBid
    a. Metoda umożliwia graczowi złożenie oferty (bida) w trakcie fazy licytacji.
    b. Sprawdza autoryzację użytkownika.
    c. Przekazuje identyfikatory gry i gracza oraz kwotę oferty do repozytorium.
    d. Zwraca odpowiedź zależną od wyniku operacji.

2. EndBidding
    a. Metoda kończy fazę licytacji w grze.
    b. Sprawdza autoryzację użytkownika
    c. Przekazuje identyfikator gry do repozytorium.
    d. Zwraca odpowiedź zależną od wyniku operacji.

BidRepositry:

1. MakeBid
    a. Realizuje logikę składania ofert w trakcie fazy licytacji.
    b. Waliduje dane gracza, gry i kwoty oferty.
    c. Aktualizuje stan gry, saldo gracza i pulę gry.
    d. Wysyła zdarzenie SignalR informujące o nowej ofercie.

```
public async Task<CustomMessageResult> MakeBid(Guid gameId, Guid playerId, int bidAmount)

{

  if (!IsBidAmountValid(bidAmount))

    return CreateErrorResult("Bid amount must be divisible by 100.");
```

```
var game = await GetGame(gameId);
if (game == null)
   return CreateErrorResult("Game not found.");

if (!IsBiddingPhaseActive(game))
   return CreateErrorResult("Bidding is not active in this game.");

var playerScore = GetPlayerScore(game, playerId);
var currentUserBid = await GetCurrentUserBid(gameId, playerId);
if (currentUserBid == null) {
   return CreateErrorResult("User is not included in this bidding round.");
}
if (playerScore == null)
   return CreateErrorResult("Player not found in the game.");

if (!HasSufficientFunds(playerScore, currentUserBid, bidAmount))
   return CreateErrorResult("Insufficient funds to make this bid.");

var highestBid = await GetHighestBid(gameId);

if (IsBiddingTimePassed(highestBid))
{
   return CreateErrorResult("Too late! Time for bidding have passed.");
};

var highestBidAmount = 0;
if (highestBid != null)
   highestBidAmount = highestBid.Amount;
```

```
  if (!IsBidHigherThanCurrentHighest(bidAmount, highestBidAmount))
    return CreateErrorResult("Bid amount must be higher than the current highest bid.");

  var currentUserBidAmount = 0;
  if (currentUserBid != null)
  {
    currentUserBidAmount = currentUserBid.Amount;
    _context.Bids.Remove(currentUserBid);
  }

  var bid = CreateNewBid(gameId, playerId, bidAmount);
  _context.Bids.Add(bid);

  var bidIncrease = bidAmount - currentUserBidAmount;
  game.Pool += bidIncrease;
  playerScore.Balance -= bidIncrease;

  await _context.SaveChangesAsync();

  var signalRGroup = _hubContext.Clients.Group(gameId.ToString().ToLower());
  var bidDoneEventDto = new BidDoneEventDto
  {
    PlayerId = playerId,
    NewAccountBalance = playerScore.Balance,
    NewPool = game.Pool,
    Timestamp = bid.TimeStamp,
```

```
    Amount = bid.Amount
  };
  await signalRGroup.SendAsync("BidDone", bidDoneEventDto);
  return CreateSuccessResult($"Bid placed successfully: Player: {playerId}, Offer: {bidAmount}");
}
```

2. EndBidding
    a. Odpowiada za zakończenie fazy licytacji.
    b. Waliduje dane gry i czasu od ostatniej oferty.
    c. Określa zwycięzcę licytacji.
    d. Przechodzi do fazy pytań i wysyła zdarzenie SignalR informujące o zakończeniu licytacji.

```
public async Task<CustomMessageResult> EndBidding(Guid gameId)
{
  var game = await GetGameWithQuestions(gameId);
  if (game == null)
    return CreateErrorResult("Game not found.");

  if (!IsBiddingPhaseActive(game))
    return CreateErrorResult("Bidding phase is not active.");

  var lastBid = await GetLastBid(gameId);
  if (!IsBiddingTimePassed(lastBid))
    return CreateErrorResult("Bidding phase cannot be ended. Less than 10 seconds since the last bid.");

  var highestBid = await GetHighestBid(gameId);
```

```
    var highestBidder = game.PlayerScores.Where(p => p.PlayerId ==
highestBid.PlayerId).FirstOrDefault();
    if (highestBidder == null)
        return CreateErrorResult("Coudn't find the highest bidder.");

    game.GameState = GameState.QUESTION;
    await _context.SaveChangesAsync();

    var currentQuestion = game.Questions.ElementAtOrDefault(game.Round - 1);
    var biddingEndEventDto = new BiddingEndEventDto
    {
        QuestionText = currentQuestion.QuestionText,
        Answers = currentQuestion.Answers.Split(";").ToList()
    };
    var signalRGroup = _hubContext.Clients.Group(gameId.ToString().ToLower());
    await signalRGroup.SendAsync("BiddingEnd", biddingEndEventDto);

    return new CustomMessageResult
    {
        Success = true,
        Message = "Bidding phase ended. Proceeding to question phase.",
        Obj = CreateBidResult(highestBid)
    };
}
```

5. Autentykacja i autoryzacja

W celu zaimplementowania autoryzacji wykorzystany został mechanizm JWT, Identity oraz Role-based Authorization.

Rejestracja użytkownika:

- Użytkownik rejestruje się za pomocą metody Register w kontrolerze AuthController.
- Dane rejestracyjne (email, hasło, nazwa użytkownika) są weryfikowane pod kątem poprawności.
- Użytkownik jest tworzony w systemie z użyciem UserManager (metoda CreateAsync).
- Domyślnie przypisywana jest rola użytkownika Player (lub Admin w przypadku metody RegisterAdmin).

Logowanie użytkownika:

- Użytkownik przesyła swoje dane (email i hasło) do metody Login.
- System weryfikuje dane:
  - Sprawdza istnienie użytkownika w bazie (FindByEmailAsync).
  - Weryfikuje poprawność hasła (CheckPasswordAsync).
  - Pobiera role przypisane do użytkownika.
- Tworzone są dwa tokeny:
  - Token JWT (ważny przez 15 minut) – wykorzystywany do autoryzacji.
  - Refresh token (ważny 7 dni) – używany do odświeżania sesji bez konieczności ponownego logowania.
- Refresh token jest zapisywany w bazie danych i przesyłany do klienta jako ciasteczko.

Odświeżanie tokena:

- Metoda RefreshToken w kontrolerze AuthController
- Pobiera refresh token z ciasteczek.
- Weryfikuje jego ważność i obecność w bazie danych.
- Jeśli wszystko jest poprawne, generuje nowe tokeny JWT i refresh.

```
[HttpGet("RefreshToken")]

public async Task<IActionResult> RefreshToken()

{

  if (Request.Cookies.TryGetValue("jwt", out string jwt))

  {

    var refreshToken = await _tokenRepository.GetRefreshTokenAsync(jwt);

    if (refreshToken == null || refreshToken.ExpiryDate < DateTime.Now)

    {
```

```
            return Forbid();
        }
        var user = await _userRepository.GetUserById(refreshToken.UserId);

        var roles = await _userManager.GetRolesAsync(user);
        if (roles == null || roles.Count == 0)
            return BadRequest("No roles found for given user");

        var token = _tokenRepository.CreateJWTToken(user, roles.ToArray());
        var newRefreshToken = await _tokenRepository.CreateRefreshToken(user);
        Response.Cookies.Append("jwt", newRefreshToken, _cookieOptions);
        var response = new LoginResponseDto
        {
            Id = user.Id,
            JwtToken = token,
            UserName = user.UserName,
            Email = user.Email,
            Roles = roles.ToList()
        };
        return Ok(response);
    }

    return Forbid();
}
```

```
private JwtSecurityToken GenerateToken(List<Claim> claims, DateTime expires)
{
```

```
    var key = new SymmetricSecurityKey(Encoding.UTF8.GetBytes(_conf["JWT:Key"]));
    var credentials = new SigningCredentials(key, SecurityAlgorithms.HmacSha256);
    return new JwtSecurityToken(_conf["JWT:Issuer"], _conf["JWT:Audiance"], claims,
expires: expires, signingCredentials: credentials);
}

public string CreateJWTToken(IdentityUser user, string[] roles)
{
    var claims = new List<Claim>
    {
        new Claim(ClaimTypes.Email, user.Email),
        new Claim(ClaimTypes.NameIdentifier, user.Id)
    };
    foreach (var role in roles)
        claims.Add(new Claim(ClaimTypes.Role, role));

    var token = GenerateToken(claims, DateTime.Now.AddMinutes(15));

    return new JwtSecurityTokenHandler().WriteToken(token);
}

public async Task<string> CreateRefreshToken(IdentityUser user)
{
    var expiryDate = DateTime.Now.AddDays(7);
    var token = GenerateToken(new List<Claim>(), expiryDate);
    var tokenString = new JwtSecurityTokenHandler().WriteToken(token);
    await SaveRefreshTokenAsync(user.Id, tokenString, expiryDate);
    return tokenString;
```

```
}


private async Task SaveRefreshTokenAsync(string userId, string token, DateTime
expiryDate)
{
  var refreshToken = new RefreshToken
  {
    Token = token,
    UserId = userId,
    ExpiryDate = expiryDate,
    IsRevoked = false
  };


  _context.RefreshTokens.Add(refreshToken);
  await _context.SaveChangesAsync();
}
```

Wylogowanie użytkownika:

- Metoda Logout usuwa refresh token przypisany do użytkownika z bazy danych.

Konfiguracja autoryzacji:

- Mechanizm autoryzacji opiera się na JwtBearer:
  - Walidowane są: wystawca tokena (Issuer), odbiorca (Audience), czas ważności (Lifetime) oraz klucz podpisujący (SigningKey).
- Dla endpointów wymagających uwierzytelnienia lub autoryzacji, system automatycznie sprawdza poprawność tokena i jego zgodność z wymaganymi rolami.

6. Tworzenie kontenerów

Plik Dockerfile przedstawiony poniżej używa obrazu bazowego z runtime'em ASP.NET 8.0. Otwierany port 8080, aby aplikacja mogła komunikować się na tym porcie. Dodatkowo, ustalawiona jest zmienna środowiskowa ASPNETCORE_ENVIRONMENT na Development aby móc korzystać z GUI do API.

Użyty został obrazu SDK (Software Development Kit), który zawiera wszystkie narzędzia potrzebne do budowania aplikacji .NET. Tworzony jest katalog roboczy /src, a następnie kopiowane są wszystkie pliki projektu z lokalnej maszyny do kontenera. Uruchomione zostaje polecenie dotnet restore, aby przywrócić zależności NuGet (potrzebne paczki aplikacji), po czym aplikacja jest kompilowana za pomocą dotnet build, zapisując wynik w katalogu /app/build.

Kolejnym krokiem jest publikacja aplikacji. Obrazu z etapu kompilacji zostaje opublikowanay za pomocą polecenie dotnet publish dodatkowo przygotowując ją do uruchomienia w produkcji, zapisując pliki w katalogu /app/publish. Następnie przechodzi do finalnego obrazu, który bazuje na obrazie runtime .NET (z pierwszego etapu), kopiuję opublikowaną aplikację do tego obrazu. Na koniec ustawia punkt wejścia kontenera na dotnet Awantura.Api.dll, co oznacza, że po uruchomieniu kontenera aplikacja .NET zostanie wystartowana.

```
FROM mcr.microsoft.com/dotnet/aspnet:8.0 AS base
WORKDIR /app
EXPOSE 8080
ENV ASPNETCORE_ENVIRONMENT=Development

# Install the libgssapi_krb5 library
RUN apt-get update && \
    apt-get install -y libgssapi-krb5-2 && \
    rm -rf /var/lib/apt/lists/*

# Use the official Microsoft .NET Core SDK image to build the project files
FROM mcr.microsoft.com/dotnet/sdk:8.0 AS build
WORKDIR /src

# Copy the project files and restore dependencies
COPY . .
RUN dotnet restore "Awantura.Api/Awantura.Api.csproj"
```

```
# Build the application
RUN dotnet build "Awantura.Api/Awantura.Api.csproj" -c Release -o /app/build

# Publish the application
FROM build AS publish
RUN dotnet publish "Awantura.Api/Awantura.Api.csproj" -c Release -o /app/publish

# Build the final image
FROM base AS final
WORKDIR /app
COPY --from=publish /app/publish .

# Set the entrypoint to the correct path of the DLL
ENTRYPOINT ["dotnet", "Awantura.Api.dll"]
```

Skupiając się na opisie części backendowej, w pliku docker-compose.yml skonfigurowano dwie usługi: drivers-api oraz mssql. Usługa drivers-api korzysta z obrazu drivers-image i jest skonfigurowana do pracy z dwoma zmiennymi środowiskowymi: CONNECT_TO_DOCKER_DB oraz CLIENT_URL. Usługa ta działa w trybie tylko do odczytu (read_only: true), co zapewnia dodatkowe bezpieczeństwo. Aplikacja backendowa jest dostępna na porcie 8080.

Usługa mssql używa obrazu mcr.microsoft.com/mssql/server:2022-latest i wymaga ustawienia zmiennych środowiskowych ACCEPT_EULA oraz MSSQL_SA_PASSWORD. Baza danych jest dostępna na porcie 1433, a jej dane przechowywane są w woluminie sqlserver_data. Obie usługi są podłączone do wspólnej sieci common-network, umożliwiając ich komunikację.

```
services:
  frontend:
```

```
  restart: always
  build:
   dockerfile: Dockerfile.dev
   context: ./frontend
  volumes:
   - ./frontend/node_modules:/opt/app/node_modules
   - ./frontend:/opt/app
  environment:
   - WDS_SOCKET_PORT=3000
  ports:
   - "3000:3000"
  networks:
     - common-network

drivers-api:
  image: drivers-image
  environment:
   CONNECT_TO_DOCKER_DB: ${CONNECT_TO_DOCKER_DB}
   CLIENT_URL: ${CLIENT_URL}
  read_only: true
  build:
   context: ./backend
   dockerfile: Awantura.Api/Dockerfile
  ports:
   - "8080:8080"
  networks:
   - common-network
```

```
  mssql:
    image: mcr.microsoft.com/mssql/server:2022-latest
    environment:
      ACCEPT_EULA: "Y"
      MSSQL_SA_PASSWORD: ${MSSQL_SA_PASSWORD}
    ports:
      - "1433:1433"
    volumes:
      - sqlserver_data:/var/opt/mssql
    restart: always
    networks:
      - common-network


networks:
  common-network:
    driver: bridge
volumes:
  sqlserver_data:
```