TECHNICAL UNIVERSITY OF CLUJ-NAPOCA ROMANIA

Sigma Football Manager

Team members

Calomfirescu Radu-Adrian: *Software tester, Team Leader*
Farcas Andrei: *Team Leader, Database developer*
Hadarau Cristian-Ioan: *Database developer, Software developer*
Iacobut Florin-Cosmin: *Software developer, Software tester*
Morar Antonio-Axel: *Software developer, Software tester*

Coordinating teacher:
Dr. Ing. Teodora Sanislav

Contents

1 Introduction

This project is a reflection of our two main passions, where we have tried our best to bring together football and programming in a web application that simulates at a very low level the experience of a football manager.
The project "Sigma Football Manager" was built as a semester project and was our first experience with ADO .NET Core technologies. Consequently, we have seen it as a learning process and even though we have faced some difficult challenges, we managed to overcome them, we became better at every step and we have developed a web application that we could not have been more proud of.
The project has been developed using an MVC (Model View Controller) architecture and was created as a ADO.NET Core Web Application project using Visual Studio. It has two main parts, one for the clients and one for the administrator. In order to build all the functionalities we have set as objectives at the beginning, we have used knowledge about C#, SQL, Database Design, HTML & CSS and Java Script.
A presentation about all the steps we have taken in building this application will be provided in the following section. However, only the most important parts of the project will be presented in detail.

2 Application design

2.1 Use cases

The primary objective of this documentation is to guide you through the process of creating and understanding Sigma Football Manager. Our main actors are the users and the administrator, each of those having its own application built to fulfil his role. I have further split the Diagram into 4 big functionalities: the team management, the player shop, the administration and the game-play section.

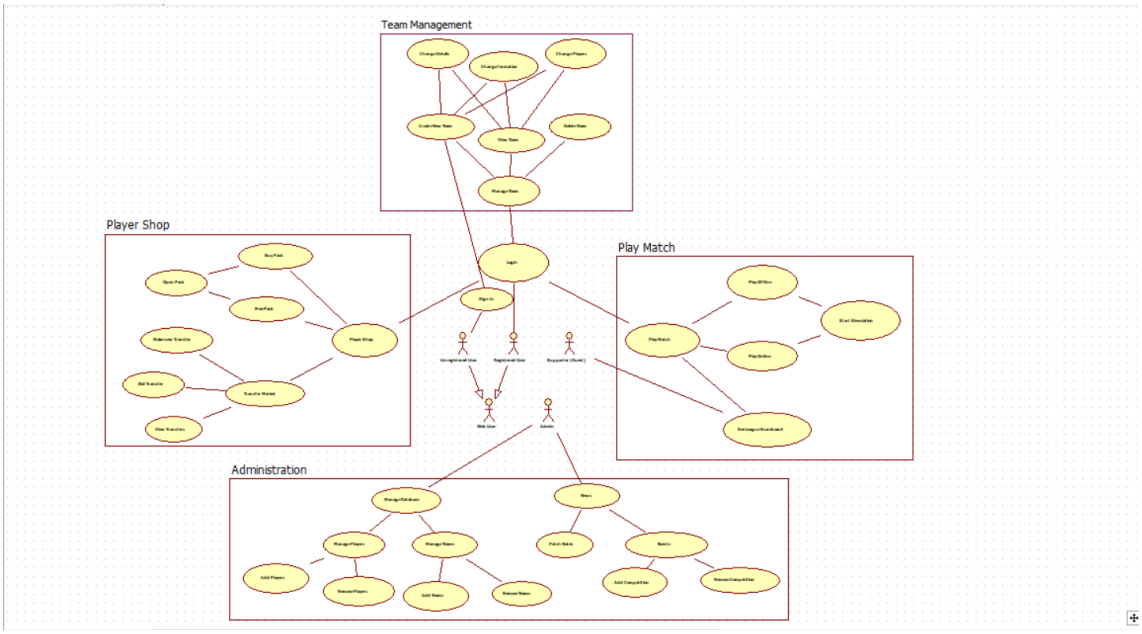

Figure 1: Use Case Diagram

3 Database design

3.1 Database Overview

The Football Database is a complete and well-organized database for storing and managing football-related information. With tables dedicated to players, squads, team contracts, transfers, and users, the database provides an organized structure to support various functionalities of a football manager application. It allows efficient tracking of player details, squad compositions, contract information, and transfer activities. The database's relationships enable smooth data management and contribute to the overall functionality of the application.

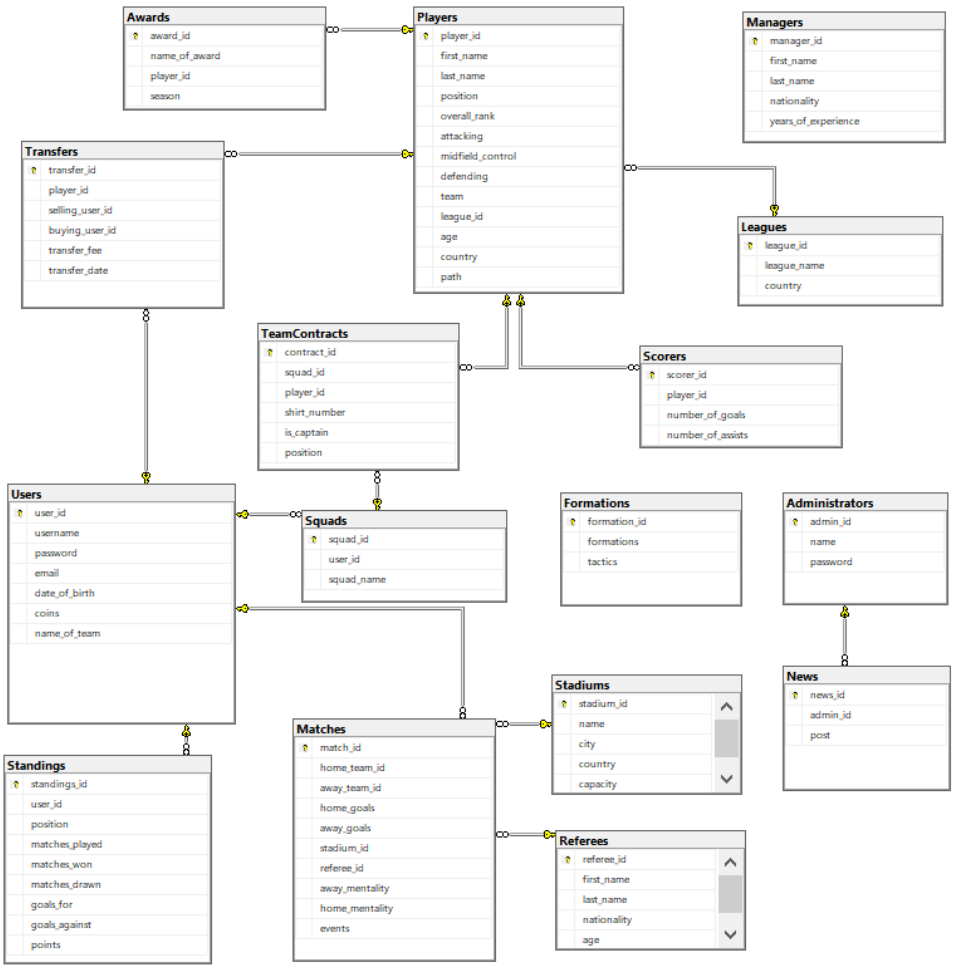

Figure 2: Database Diagram

3.2 Tables Descriptions

1. **Players:** This table serves as a repository for accurate and real football player information. It is populated manually to ensure the accuracy and authenticity of the data. The table includes columns such as player_id, name, overall rank, attacking attributes, midfield attributes, defending attributes, team, league, age, and country. By structuring the data in this manner, we guarantee that the player information reflects real-world players, providing a reliable foundation for a realistic football management experience within the game.
2. **Squads:** The Squads table stores user-created football teams in the game. It includes squad_id, user_id, and squad_name columns. The squad_id uniquely identifies each squad, user_id represents the associated user, and squad_name holds the user's chosen name for the team. This allows users to create and manage their own squads, enhancing their gaming enjoyment.
3. **Team Contracts**: The Team Contracts table acts as a essential link between squads, users, and players, ensuring a normalized structure within the game database. It stores contract information, including contract_id, squad_id, player_id, shirt number, captaincy status, and player position. The "position" column follows a logical scheme: positions 1 to 11 represent players in the starting lineup, positions 12 to 18 designate substitutes, position 19 is for non-first team players, and position 20 indicates players listed in the transfer market.
4. **Transfers**: The Transfers table keeps track of player transfers between clubs in the game. It stores essential information including transfer_id, player_id, selling club_id, buying club_id, transfer fee, and transfer date. This table captures the dynamic nature of player movements, allowing for accurate recording and monitoring of transfers. By storing details such as the clubs involved, transfer fees, and dates, the Transfers table provides a comprehensive record of player transactions, adding realism and depth to the gaming experience.
5. **Users**: The Users table is a vital component of the database, storing key user details such as user_id, username, password, email, date_of_birth, coins, and team name. It plays a central role in managing

user accounts and enabling personalized game customization.
6. **Standings**: The Standings table provides an overview of player rankings in the game. It includes columns like standings_id, user_id, position, matches_played, matches_won, matches_drawn, goals_for, goals_against, and points. This table showcases the user's position, matches played, wins, draws, goals scored, goals conceded, and accumulated points. It serves as a reference for assessing player performance and determining their ranking within the game.
7. **Matches**: The Matches table contains essential information about the game fixtures. It includes columns like match_id, home_team_id, away_team_id(both linked to the user_id) home_goals, away_goals, stadium_id, referee_id, away_mentality, home_mentality, and events. This table provides key insights into each match, such as the participating teams, goals scored, stadium and referee details, and a record of significant events during the game. It serves as a valuable resource for analyzing match outcomes and monitoring team progress in the game.
In conclusion, the Football Database encompasses a range of tables designed to support the functionality of the football management application. While we have highlighted the key tables such as Players, Squads, TeamContracts, Transfers, Users and Matches it's important to note that there are other supplementary tables that contribute to the overall structure. These additional tables can be explored further in the accompanying database diagram. Together, these tables form a comprehensive foundation for a realistic gaming experience.

4 Administrator implementation

4.1 Administrator structure

The Administrator application is much simpler than the Client application and offers a simple menu that allows the user to select one of three options: "Players", "User" and "News".

Figure 3: Navigation bar of the Administrator application

The purpose of each section is to offer information and easily accessible option to modify data related to users, players and also add news that will be then be available for the clients.

4.2 Players Management

The players section plays an essential role in our project. By regularly updating the player table, administrators ensure users have the most accurate and relevant player information for making good team-building and strategic decisions. Firstly, when the "Players" section is accessed, you will find a table created using HTML that displays all the players. This table presents important player information such as names, positions, overall rank, and other attributes. It provides a user-friendly interface which was styled using CSS for administrators to manage and oversee the player database effectively. By utilizing this table, administrators can easily view and organize player data, facilitating efficient player management.

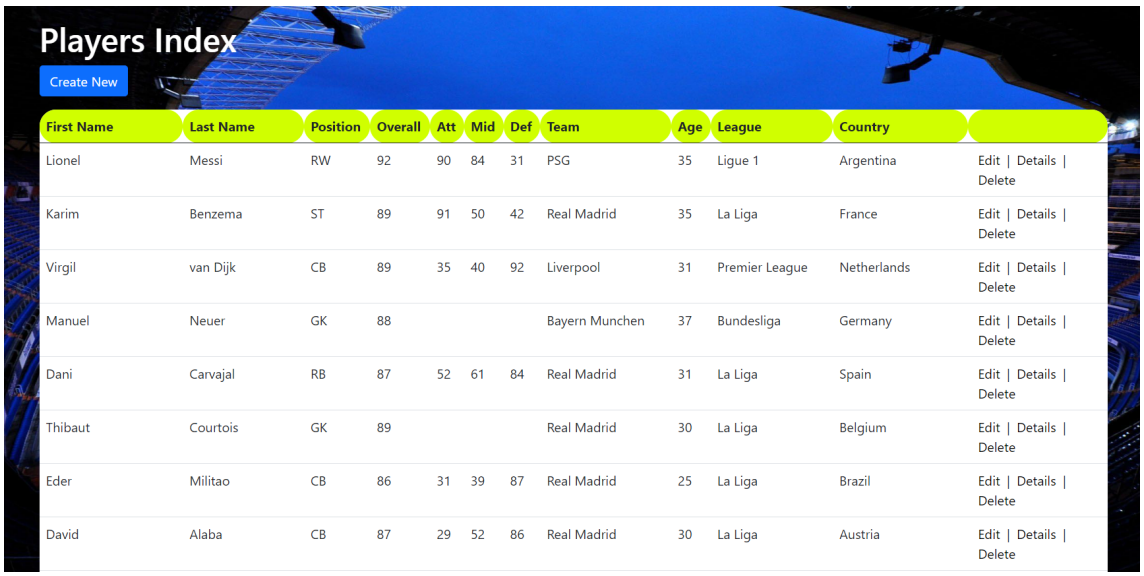

Figure 4: Players Table

The "Create New" button allows administrators to add players to the database. They can input player information such as name, position, overall rank, and other attributes. Once added, these players become part of the database, ready to be used in the client application.
In the table, administrators have access to three essential functionalities for modifying the player database: editing, deleting, and viewing details. These functionalities are represented by buttons in the table that trigger specific actions. Here is the code snippet for these buttons:

Index.cshtml - Players

```
<td>
    <a class="butonLink" asp-action="Edit" asp-route-id="@item.PlayerId">Edit</a> |
    <a class="butonLink" asp-action="Details" asp-route-id="@item.PlayerId">Details</a> |
    <a class="butonLink" asp-action="Delete" asp-route-id="@item.PlayerId">Delete</a>
</td>
```

For instance, by clicking the "Edit" button, administrators can make changes to a player's details using a dedicated view page. Similarly, the "Details" button allows administrators to access comprehensive player information, while the "Delete" button removes the player from the database.

Details.cshtml - Dedicated view for player's details

```
@model FootballManager_v0._1.Models.Player
@{
    ViewData["Title"] = "Details";
}
<h1>Details</h1>
<div>
    <h4>Player</h4>
    <hr />
    <dl class="row">
        <dt class = "col-sm-2">
            @Html.DisplayNameFor(model => model.FirstName)
        </dt>
        <dd class = "col-sm-10">
            @Html.DisplayFor(model => model.FirstName)
        </dd>
        <dt class = "col-sm-2">
            @Html.DisplayNameFor(model => model.LastName)
        </dt>
        <dd class = "col-sm-10">
            @Html.DisplayFor(model => model.LastName)
        </dd>
        <dt class = "col-sm-2">
            @Html.DisplayNameFor(model => model.Position)
```

```
        </dt>
        <dd class = "col-sm-10">
            @Html.DisplayFor(model => model.Position)
        </dd>
        <dt class = "col-sm-2">
            @Html.DisplayNameFor(model => model.OverallRank)
        </dt>
        <dd class = "col-sm-10">
            @Html.DisplayFor(model => model.OverallRank)
        </dd>
        <dt class = "col-sm-2">
            @Html.DisplayNameFor(model => model.Attacking)
        </dt>
        <dd class = "col-sm-10">
            @Html.DisplayFor(model => model.Attacking)
        </dd>
        <dt class = "col-sm-2">
            @Html.DisplayNameFor(model => model.MidfieldControl)
        </dt>
        <dd class = "col-sm-10">
            @Html.DisplayFor(model => model.MidfieldControl)
        </dd>
        <dt class = "col-sm-2">
            @Html.DisplayNameFor(model => model.Defending)
        </dt>
        <dd class = "col-sm-10">
            @Html.DisplayFor(model => model.Defending)
        </dd>
        <dt class = "col-sm-2">
            @Html.DisplayNameFor(model => model.Team)
        </dt>
        <dd class = "col-sm-10">
            @Html.DisplayFor(model => model.Team)
        </dd>
        <dt class = "col-sm-2">
            @Html.DisplayNameFor(model => model.Age)
        </dt>
        <dd class = "col-sm-10">
            @Html.DisplayFor(model => model.Age)
        </dd>
        <dt class = "col-sm-2">
            @Html.DisplayNameFor(model => model.League)
        </dt>
        <dd class = "col-sm-10">
            @Html.DisplayFor(model => model.League.LeagueId)
        </dd>
    </dl>
</div>
<div>
    <a class="btn btn-primary" asp-action="Edit" asp-route-id="@Model?.PlayerId">Edit</a> |
    <a class="btn btn-primary" asp-action="Index">Back to List</a>
</div>
```

These buttons offer a straightforward and user-friendly approach for administrators to manage the player database efficiently. When a button is clicked, it calls specific functions from the controller, allowing administrators to effortlessly edit, view, and delete player information. This structured approach follows the Model-View-Controller (MVC) architecture, ensuring efficient data management and an intuitive user experience. Within this framework, controllers handle the background operations, while the view pages serve as the interface for administrators to interact with the system. This clear division of tasks simplifies the management process and enhances overall usability throughout the application.

PlayersController - Delete functions

```
public async Task<IActionResult> Delete(int? id)
{
    if (id == null || _context.Players == null)
    {
        return NotFound();
    }

    var player = await _context.Players
        .Include(p => p.League)
        .FirstOrDefaultAsync(m => m.PlayerId == id);
    if (player == null)
    {
        return NotFound();
    }

    return View(player);
}

[HttpPost, ActionName("Delete")]
[ValidateAntiForgeryToken]
public async Task<IActionResult> DeleteConfirmed(int id)
{
    if (_context.Players == null)
    {
        return Problem("Entity set 'FootballDatabaseContext.Players'  is null.");
    }
    var player = await _context.Players.FindAsync(id);
    if (player != null)
    {
        _context.Players.Remove(player);
    }

    await _context.SaveChangesAsync();
    return RedirectToAction(nameof(Index));
}
```

4.3 Users and News Management

The administrator's role extends beyond managing the player database. They also oversee the user and news sections, which operate on the same logical structure. With user management, the administrator can add, edit, or remove user accounts, ensuring the system remains up to date. Similarly, in the news section, the administrator can create, edit, and delete news articles or announcements. This consistent approach across players, users, and news simplifies the administrator's tasks. The controllers handle the behind-the-scenes operations, while dedicated views provide an interface for efficient management of these sections.

5 Client implementation

5.1 Client structure

The main part of the project can be split into seven parts, each with its own unique functionalities and important purpose in creating the Sigma Football Manager experience. The first contact of a user with the application is an authentication system that allows him to create a new account or log in to his existing one. Once authenticated, the user is presented with the main menu, where he can choose to access one of the other big functionalities. He can choose to edit his squad and manage how his team will line up in the squad menu, he can visit the transfer market menu where he can buy or sell specific players to or from other users, he can visit the store, where he can buy a pack that has random players of a specific quality, or he can visualize standings or announcements made by the administrators. Last but certainly not least, after preparing a competitive team, the client application offers the users the

possibility to test his team against other managers in an online simulation, and accumulate points for the leader board standings and coins to further improve his team. All this functionalities will be described in detail together with the most important parts of the implementation, in the following subsections.

5.2 Access and authentication process

One of the most important functionalities of the application, the authentication process is handled by the **AccessController**. It encompasses several key methods that collectively enable users to securely access their accounts and utilize the application's features. The **AccessController** serves as the central hub for managing user authentication, from initial sign up to team creation.

- The **Sign up** method allows new users to create an account by providing the required registration information. It ensures that user credentials are securely stored and establishes a unique account for each user.
- The **Login** method validates user credentials and grants access to the application's features upon successful authentication. It employs robust security measures to protect user accounts from unauthorized access.
- The **Team Creation** functionality ensures that each user can create only one team. It verifies if the user has already created a team and appropriately handles the creation process, ensuring a seamless user experience.

By integrating these functionalities, the **AccessController** streamlines the authentication process, providing users with a secure and hassle-free access to their accounts.

5.2.1 Sign up

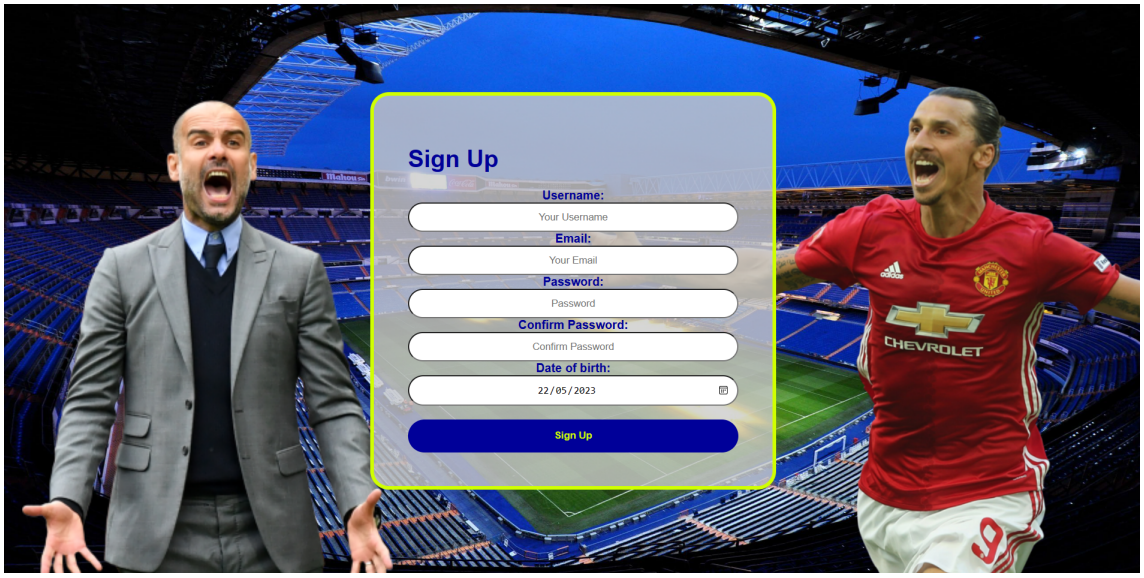

Figure 5: Sign up page

The view presents a sign up form where users can enter their username, email, password, and date of birth. The form is submitted to the "*Signup*" action of the "*Access*" controller. Any validation errors or custom messages are displayed below the form.

Signup View

```
@model ClientApplication.Models.Signup;
@{
    Layout = null;
    ViewData["Title"] = "Register";
}

<!DOCTYPE html>
```

```
<html>
    <head>
        <meta name="viewport" content="width=device-width" />
        <title>Football Manager - Register</title>
        <link rel="stylesheet" href="~/css/site.css" />
    </head>
    <body>

        <form asp-controller="Access" asp-action="Signup" method="post">
            <div class="centerContainer" style=" margin-top: 8%;">
                <h1>Sign Up</h1>

            <label style="text-align:center;"><b>Username:</b></label>
                <input type="text" placeholder="Your Username" asp-for="Username"
                ↪  class="textContainer" required/>
                <label style="text-align:center;"><b>Email:</b></label>
            <input type="email" placeholder="Your Email" asp-for="Email" class="textContainer"
            ↪  required />
                <label style="text-align:center;"><b>Password:</b></label>
            <input type="password" placeholder="Password" asp-for="Password" class="textContainer"
            ↪  required />
                <label style="text-align:center;"><b>Confirm Password:</b></label>
            <input type="password" placeholder="Confirm Password" asp-for="ConfirmPassword"
            ↪  class="textContainer" required />
                <label style="text-align:center;"><b>Date of birth:</b></label>
                <input type="date" value="@ViewBag.Today" min="1900-01-01" max="@ViewBag.Today"
                ↪  asp-for="DateOfBirth" class="textContainer" />
                <br />
                <button class="accessButton" type="submit">Sign Up</button>

                @if (ViewData["ValidateMessage"] != null)
                {
                    <br />
                    <br />
                    <label>@ViewData["ValidateMessage"]</label>
                }
            </div>
        </form>

    <img style="position: absolute;right: 0%;top: 10%;z-index: -1; max-width: auto; height: 65%;"
    ↪  src="~/css/legends/ibra.png"/>
    <img style="position: absolute;right: 57%;top: 9%;z-index: -1; max-width: auto; height: 65%;"
    ↪  src="~/css/legends/pep.png" />
    </body>
</html>
```

First, we have the **Signup** GET method. Its purpose is to handle the sign up functionality and render the associated view. The method checks whether the user is already authenticated. If the user is authenticated, the method redirects them to the "*Index*" action of the "*Home*" controller using the **RedirectToAction** method. If the user is not authenticated, the method continues executing by returning the view associated with the "*Signup*" action. The view will be responsible for displaying the sign up form to the user.

AccessController - Signup() GET method

```
// Redirects the user to the home page if they are already authenticated; otherwise, displays the
↪  signup view
public IActionResult Signup()
    {
        ClaimsPrincipal claimUser = HttpContext.User;
        if (claimUser.Identity.IsAuthenticated)
        {
            return RedirectToAction("Index", "Home");
        }
```

```
        var today = DateTime.Today.ToString("yyyy-MM-dd");
        ViewBag.Today = today;
        return View();
    }
```

Upon clicking the Submit button, an HTTP POST request is made which handles the form submission for the sign up functionality. It validates the provided username, email, and password, creates a new user instance, saves it to the database, generates claims for the user's identity, signs the user in, and redirects them to the "*Create*" action of the "*Access*" controller to choose a team name. It sets the default name for user's team and gives them 1000 coins. It also handles various error scenarios, such as existing username or email, password mismatch, or missing input.

AccessController - Signup() POST method

```
[HttpPost]
public async Task<IActionResult> Signup(Signup modelSignup)
{
    // Check if a user with the provided username already exists
    var user = _context.Users.FirstOrDefault(u => u.Username == modelSignup.Username);
    if (user == null && modelSignup.Username != null)
    {
        // Check if the email is already in use
        var mail = _context.Users.FirstOrDefault(u => u.Email == modelSignup.Email);
        if (mail == null)
        {
            // Check if the password and confirm password match
            if (modelSignup.Password == modelSignup.ConfirmPassword && modelSignup.Password !=
↪   null)
            {
                // Create a new user instance
                User utilizator = new User();
                // Set the properties of the new user
                utilizator.Username = modelSignup.Username;
                utilizator.Password = modelSignup.Password;
                utilizator.Email = modelSignup.Email;
                utilizator.DateOfBirth = modelSignup.DateOfBirth;
                // Generate a new user ID starting from the last ID present in the table
                int maxId = _context.Users.Max(u => u.UserId);
                utilizator.UserId = maxId + 1;
                // Set default values for not null fields
                utilizator.NameOfTeam = "defaultName";
                utilizator.Coins = 1000;
                // Add the user to the context and save changes to the database
                _context.Users.Add(utilizator);
                await _context.SaveChangesAsync();
                // Create claims for the user
                List<Claim> claims = new List<Claim>()
                {
                    new Claim(ClaimTypes.NameIdentifier, modelSignup.Username),
                    new Claim("OtherProperties", "Example Role")
                };
                // Create an identity with the claims
                ClaimsIdentity claimsIdentity = new ClaimsIdentity(claims,
↪   CookieAuthenticationDefaults.AuthenticationScheme);
                // Configure authentication properties
                AuthenticationProperties properties = new AuthenticationProperties()
                {
                    AllowRefresh = true,
                    IsPersistent = false
                };
                // Sign in the user
                await HttpContext.SignInAsync(CookieAuthenticationDefaults.AuthenticationScheme,
↪   new ClaimsPrincipal(claimsIdentity), properties);
```

```
47                  // Redirect to the Create action of the Access controller (user chooses name of
    ↪ team)
48                  return RedirectToAction("Create", "Access");
49              }
50              else
51              {
52                  ViewData["ValidateMessage"] = "Passwords do not match or were not entered.";
53              }
54          }
55          else
56          {
57              ViewData["ValidateMessage"] = "Email already in use.";
58          }
59      }
60      else
61      {
62          ViewData["ValidateMessage"] = "User " + modelSignup.Username + " already exists. Please
    ↪ choose a different username!";
63      }
64      return View();
65  }
```

5.2.2 Login

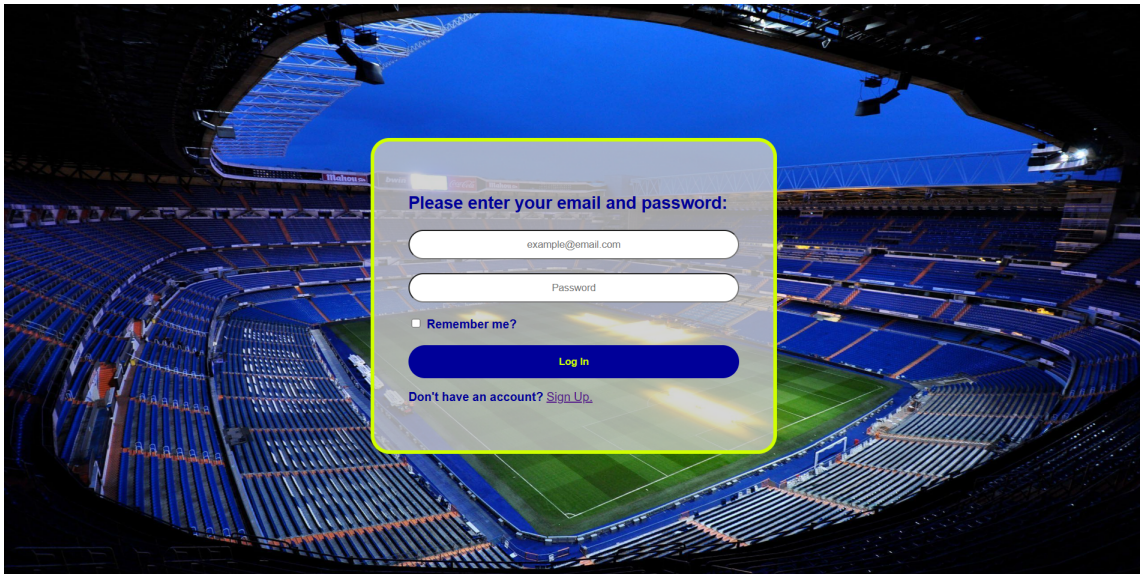

Figure 6: Login page

The **Login** view is responsible for displaying the login form to the user. It includes input fields for the email and password, along with an option to remember the user's login. The form is submitted to the "*Login*" action of the "*Access*" controller using the HTTP POST method. If there are any error messages stored in the ViewData dictionary, they are displayed below the form. Additionally, there is a link to the sign up page for users who don't have an account yet.

Login View

```
1  @model ClientApplication.Models.Login;
2
3  @{
4      Layout = null;
5      ViewData["Title"] = "Log In";
6  }
7
8  <!DOCTYPE html>
```

```
<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>Football Manager - Login</title>
    <link rel="stylesheet" href="~/css/site.css" />
</head>
<body>
    <form asp-controller="Access" asp-action="Login" method="post">
        <div class="centerContainer">
            <h3 class="instruction">Please enter your email and password: <br /></h3>
            <input class ="textContainer" type="email" asp-for="Email" name="email"
            ↪  placeholder="example@email.com"/>
            <br />
            <input class="textContainer" type="password" asp-for="Password" name="password"
            ↪  placeholder="Password" />
            <br />

            <label>
                <input type="checkbox" asp-for="RememberMe" />
                <b>Remember me?</b>
            </label>

            <br />
            <button class="accessButton" type="submit">Log In</button>

            <p>
                <label><b>Don't have an account?</b></label>
                <a href="@Url.Action("Signup", "Access")">Sign Up.</a>
            </p>


            @if (ViewData["ValidateMessage"] != null)
            {
                <br />
                <br />
                <label>@ViewData["ValidateMessage"]</label>
            }

        </div>
    </form>
</body>
</html>
```

The **Login** method is responsible for rendering the login view to the user. It first checks if the user is already authenticated by examining the '*HttpContext.User*' object. If the user is authenticated, it redirects them to the home page by invoking the "*Index*" action of the "*Home*" controller using the **RedirectToAction** method. Otherwise, it returns the login view, allowing the user to enter their credentials and submit the login form.

AccessController - Login() GET method

```
// Redirects the user to the home page if they are already authenticated; otherwise, displays the
↪  login view
public IActionResult Login()
{
    ClaimsPrincipal claimUser = HttpContext.User;
    if (claimUser.Identity.IsAuthenticated)
    {
        return RedirectToAction("Index", "Home");
    }
    return View();
}
```

The **Login** method is an HTTP POST action that handles the form submission for user login. It retrieves the user from the database based on the provided email and verifies if the entered password matches the user's password. If the login credentials are valid, it creates claims for the user's identity, signs them in by creating a cookie-based authentication ticket, and redirects them to the home page or the "*Create*" action of the "*Access*" if the user doesn't have a team. In case of invalid credentials or a user not found, appropriate error messages are set in the view.

AccessController - Login() POST method

```
// Handles the login form submission
[HttpPost]
public async Task<IActionResult> Login(Login modelLogin)
{
    // Gets the user from the database based on the provided email
    var user = _context.Users.FirstOrDefault(u => u.Email == modelLogin.Email);
    if (user != null)
    {
        // Checks if the entered password matches the user's password
        if (modelLogin.Password == user.Password)
        {
            List<Claim> claims = new List<Claim>()
            {
                new Claim(ClaimTypes.NameIdentifier, user.Username),
                new Claim("OtherProperties", "Example Role")
            };
            // Creates an identity with the claims
            ClaimsIdentity claimsIdentity = new ClaimsIdentity(claims,
↪   CookieAuthenticationDefaults.AuthenticationScheme);
            // Configure authentication properties
            AuthenticationProperties properties = new AuthenticationProperties()
            {
                AllowRefresh = true,
                IsPersistent = modelLogin.RememberMe
            };
            // Signs in the user by creating a cookie-based authentication ticket
            await HttpContext.SignInAsync(CookieAuthenticationDefaults.AuthenticationScheme, new
↪   ClaimsPrincipal(claimsIdentity), properties);
            if (user.NameOfTeam == "defaultName")
            {
                return RedirectToAction("Create", "Access");
            }
            return RedirectToAction("Index", "Home");
        }
        else
        {
            ViewData["ValidateMessage"] = "The password entered is wrong.";
        }
    }
    else
    {
        ViewData["ValidateMessage"] = "User not found.";
    }
    return View();
}
```

5.2.3 Team Creation

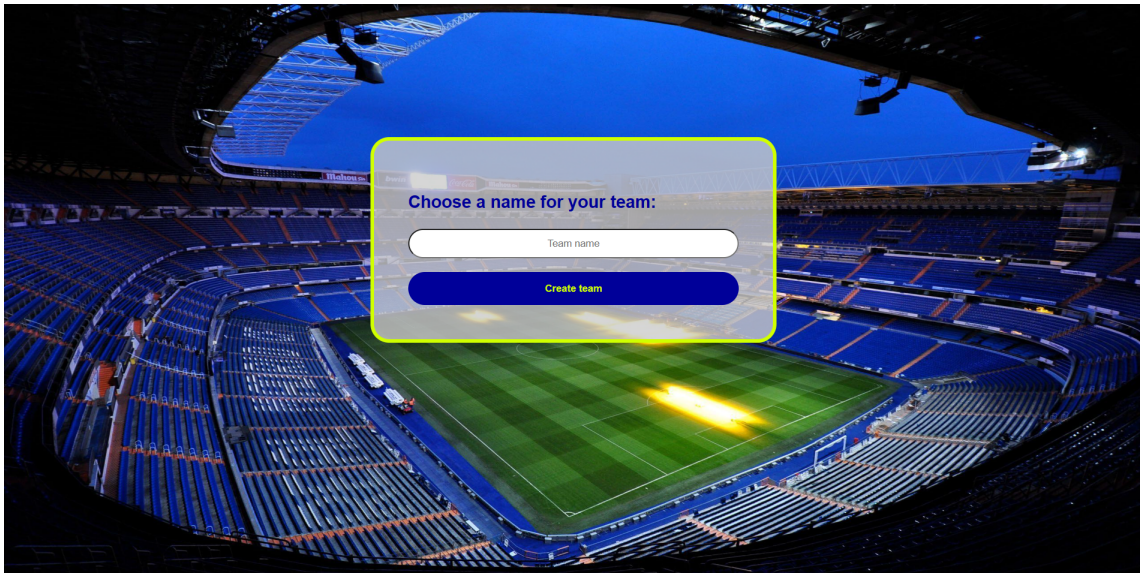

Figure 7: Team creation page

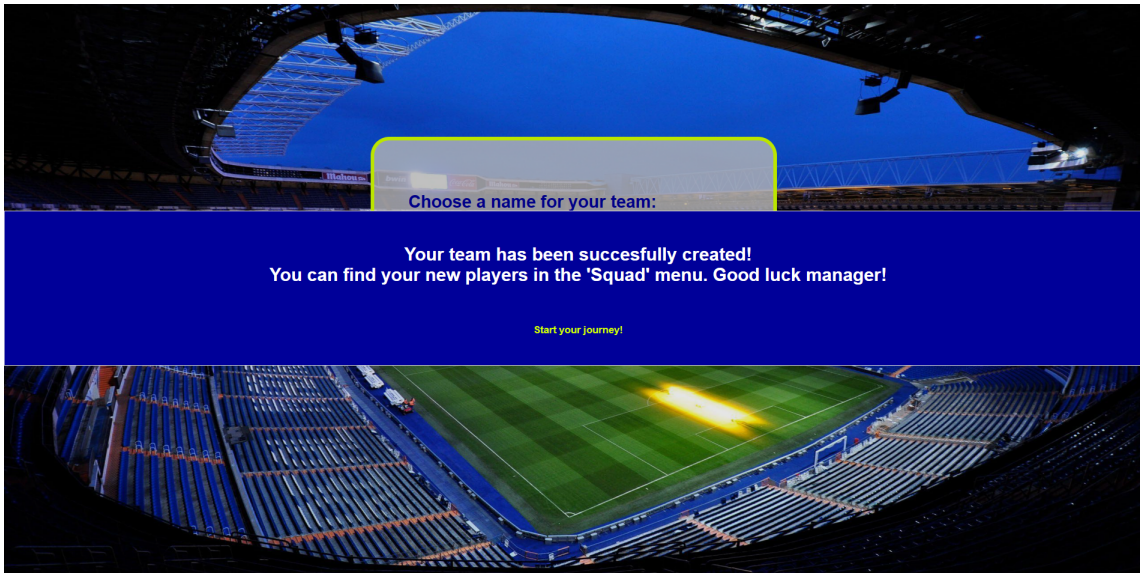

Figure 8: Team created successfully

The **Create** view is responsible for rendering a form where the user can choose a name for their team, which can't be null. It uses the Squad model to bind the team name input. The view includes a button labeled "Create team" that, when clicked, opens a dialog box displaying a success message. The success message informs the user that their team has been successfully created and encourages them to start their journey. The dialog box includes a button labeled "Start your journey!" that, when clicked, closes the dialog and submits the form, causing a redirect to the main page. The view also includes conditional rendering of a validation message if it is present in the ViewData.

Team Creation View

```
@model ClientApplication.Models.Squad

@{
    Layout = null;
    ViewData["Title"] = "Create team";
}

```

```
<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>Football Manager - Name your team</title>
    <link rel="stylesheet" href="~/css/site.css" />
</head>

<body>
    <form asp-controller="Access" asp-action="Create" method="post">
        <div class="centerContainer">
            <h3 class="instruction"><b>Choose a name for your team:</b></h3>
            <input class="textContainer" type="text" asp-for="SquadName" name="SquadName"
            ↪  placeholder="Team name" maxlength="50" required/>
            <br />
            <button class="accessButton" data-open-modal>Create team</button>

            @if (ViewData["ValidateMessage"] != null)
            {
                <br />
                <br />
                <label>@ViewData["ValidateMessage"]</label>
            }
        </div>
    </form>

    <dialog data-modal style="
                                padding: 24px;
                                border: 1px solid #dfdfdf;
                                background-color: #000099;
                                width: 100%;
                                text-align: center;">
        <h2 style="color:white">
            Your team has been succesfully created! <br />You can find your new players in the
            ↪  'Squad' menu. Good luck manager!
        </h2>
        <br />
        <button class="accessButton" data-close-modal> Start your journey! </button>
    </dialog>

    <script>
        const createTeamButton = document.querySelector('[data-open-modal]')
        const closeButton = document.querySelector('[data-close-modal]')
        const dialog = document.querySelector('[data-modal]')

        // Opens a dialog with a nice message
        createTeamButton.addEventListener('click', () => {
            event.preventDefault(); // prevent the default form submission behavior
            dialog.showModal()
        })

        // Closes the dialog and submits the form causing redirect to the main page
        closeButton.addEventListener('click', () => {
           dialog.close()
           document.querySelector('form').submit()
        })
    </script>

</body>
</html>
```

The **Create** method is an action that renders the create view for the user's team. It first retrieves the current user's username from the '*HttpContext*' and then fetches the corresponding user from the database based on the username. It checks if the user has already created a team by verifying if

the '**NameOfTeam**' property is not equal to "*defaultName*". If the user has already created a team, it redirects them to the home page by invoking the "*Index*" action of the "*Home*" controller using the **RedirectToAction** method, thus preventing them from creating a team twice. Otherwise, it returns the **Create** view, allowing the user to create their team.

AccessController - Create() GET method

```
public IActionResult Create()
{
    // Get the current user
    var username = HomeController.GetUserName(HttpContext);
    var user = _context.Users.FirstOrDefault(u => u.Username == username);
    // Check if the user has already created a team
    if (user.NameOfTeam != "defaultName")
    {
        return RedirectToAction("Index", "Home");
    }
    return View();
}
```

The **Create** method handles the form submission when creating a team. It first retrieves the current user from the database based on the username obtained from the '**HttpContext**'. If the user has already created a team (the '**NameOfTeam**' property is not "*defaultName*"), it redirects to the home page, preventing the user from submitting the form twice. Otherwise, it checks if the provided team name in the modelSquad parameter is not "*defaultName*". If a valid team name is provided, it creates a new Squad instance, sets its properties (SquadId, UserId, and SquadName), adds it to the database, and associates it with the user.
Then, it selects players from the database randomly based on their overall rank and position to form the team. The number of legendary players, rare players, and common players is determined based on the quality of the selected goalkeeper, with a total of 18 players. The players are added to the Squad by creating TeamContract instances for each player and assigning them a position, with the goalkeeper being the first. The TeamContract objects are also added to the database. Finally, it redirects to the home page with a success message if the team creation is successful. If a team name is not provided, a validation message is set in the ViewData and the view is returned to display the error message. An example of a newly created team is provided below.

AccessController - Create() POST method

```
[HttpPost]
public async Task<IActionResult> Create(Squad modelSquad)
{
    // Get the current user
    var username = HomeController.GetUserName(HttpContext);
    var user = _context.Users.FirstOrDefault(u => u.Username == username);
    // Check if the user has already created a team
    if (user.NameOfTeam != "defaultName")
    {
        return RedirectToAction("Index", "Home");
    }
    else
    {
        if (modelSquad.SquadName != "defaultName")
        {
            // Set the SquadId to be the same as the current user's UserId, set the UserID and the
↪   SquadName
            Squad echipa = new Squad();
            echipa.SquadId = user.UserId;
            echipa.UserId = user.UserId;
            echipa.SquadName = modelSquad.SquadName;
            // Add the Squad to the Database and to the user's Squads, change nameOfTeam field
↪   inside Users
            _context.Squads.Add(echipa);
            user.NameOfTeam = modelSquad.SquadName;
            user.Squads.Add(echipa);
            // Select all players (both outfield players and goalkeepers)
            var allPlayers = await _context.Players.ToListAsync();
```

```
            // Shuffle the list of players randomly
            var random = new Random();
            var shuffledPlayers = allPlayers.OrderBy(p => random.Next()).ToList();
            // Select the goalkeeper and add the goalkeeper to the team and to the database on the
↪ 1st position
            var goalkeeper = shuffledPlayers.FirstOrDefault(p => p.Position.TrimEnd() == "GK");
            // Team needs to have 18 players upon creation: 1 legendary player, 4 rare players, and
↪ 13 common players
            // We choose the other players based on the quality of the goalkeeper
            var legendaryPlayersCount = 1;
            var rarePlayersCount = 4;
            var commonPlayersCount = 13;
            if (goalkeeper != null)
            {
                if (goalkeeper.OverallRank >= 85)
                {
                    // If the goalkeeper is Legendary, we skip selecting a Legendary player
                    legendaryPlayersCount--;
                }
                else if (goalkeeper.OverallRank >= 80 && goalkeeper.OverallRank < 85)
                {
                    // If the goalkeeper is Rare, we choose one less Rare player
                    rarePlayersCount--;
                }
                else
                {
                    // If the goalkeeper is Common, we choose one less Common player
                    commonPlayersCount--;
                }
            }
            // Select additional players based on the modified counts
            var legendaryPlayers = shuffledPlayers.Where(p => p.OverallRank >= 85 &&
↪ p.Position.TrimEnd() != "GK").Take(legendaryPlayersCount).ToList();
            var rarePlayers = shuffledPlayers.Where(p => p.OverallRank >= 80 && p.OverallRank < 85
↪ && p.Position.TrimEnd() != "GK").Take(rarePlayersCount).ToList();
            var commonPlayers = shuffledPlayers.Where(p => p.OverallRank < 80 &&
↪ p.Position.TrimEnd() != "GK").Take(commonPlayersCount).ToList();
            // Combine the selected players into a single list
            var selectedPlayers =
↪ legendaryPlayers.Concat(rarePlayers).Concat(commonPlayers).ToList();
            // Add the selected players to the team and database
            int position = 1;
            int maxId = _context.TeamContracts.Max(u => u.ContractId);
            var goalkeeperContract = new TeamContract
            {
                ContractId = ++maxId,
                PlayerId = goalkeeper.PlayerId,
                SquadId = echipa.SquadId,
                ShirtNumber = null,
                IsCaptain = false,
                Position = position++
            };
            echipa.TeamContracts.Add(goalkeeperContract);
            _context.TeamContracts.Add(goalkeeperContract);
            await _context.SaveChangesAsync();
            foreach (var player in selectedPlayers)
            {
                var contract = new TeamContract
                {
                    ContractId = ++maxId,
                    PlayerId = player.PlayerId,
                    SquadId = echipa.SquadId,
                    ShirtNumber = null,
```

```
                    IsCaptain = false,
                    Position = position++
                };
                echipa.TeamContracts.Add(contract);
                _context.TeamContracts.Add(contract);
                await _context.SaveChangesAsync();
            }
            // Redirect to the index page with a success message
            return RedirectToAction("Index", "Home");
        }
        else
        {
            ViewData["ValidateMessage"] = "You must choose a name!";
        }
    }
    return View();
}
```

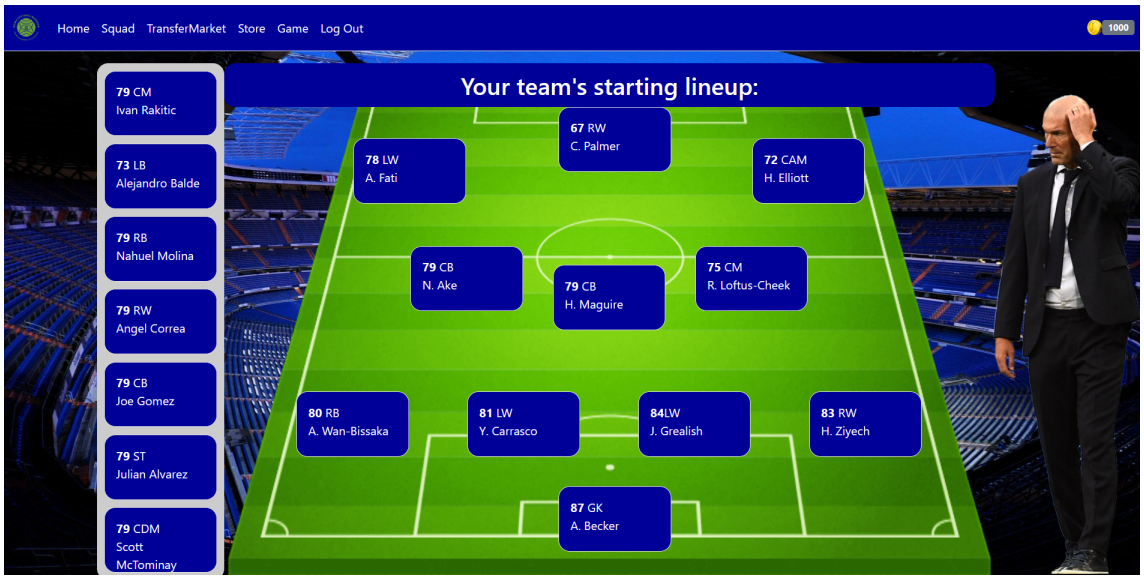

Figure 9: Example of a newly created team

5.3 Home menu, shared Layout and news

5.3.1 Home menu

The home of our application is the first thing that the user sees after log in process completion. It represents a place that offers the user a list of the main options the application offers, each with its own logo. A print screen can be seen in the figure below.

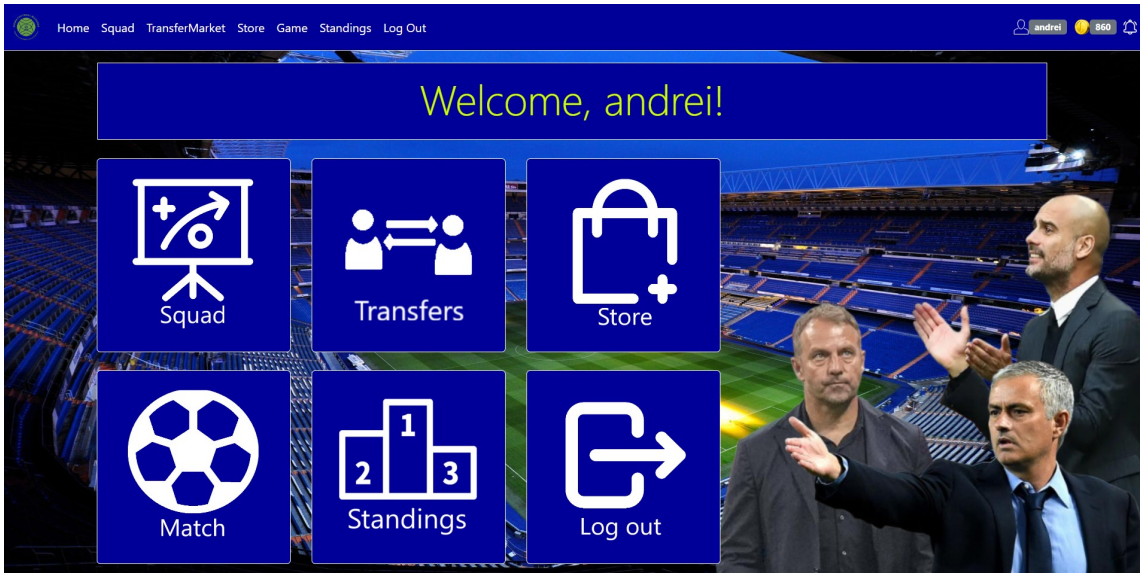

Figure 10: Home page

The six big buttons that can be seen on the page body are links that each contain a specific icon and have been styled using CSS to look like clickable buttons.

Index.cshtml - Home

```
@{
    ViewData["Title"] = "Home Page";
}
<head>
    <link rel="stylesheet" href="~/css/home.css" />
</head>

<div class="welcome">
    <h1 class="display-4">Welcome, @ViewBag.UserName!</h1>
</div>

<div class="menu">
    <a asp-controller="Squad" asp-action="Editor"><img class="option"
    ↪  src="~/css/images/iconSquad.png" alt="Squad"/></a>
    <a asp-controller="Transfer" asp-action="TransferList"><img class="option"
    ↪  src="~/css/images/iconTransfer.png" /></a>
    <a asp-controller="Home" asp-action="Store"><img class="option"
    ↪  src="~/css/images/iconStore.png" /></a>
    <br />
    <a asp-controller="Game" asp-action="GameMenu"><img class="option"
    ↪  src="~/css/images/iconGame.png" /></a>
    <a asp-controller="Home" asp-action="Standings"><img class="option"
    ↪  src="~/css/images/iconStandings.png" /></a>
    <a asp-controller="Home" asp-action="LogOut"><img class="option"
    ↪  src="~/css/images/iconLogout.png" /></a>
</div>

<img style="position: absolute;right: 0%;top: 30%;z-index: -1; max-width: auto; height: 70%;"
↪  src="~/css/legends/goats.png" />
```

5.3.2 Shared Layout - Navigation bar

The same options that are present on the buttons in the menu can also be found on the navigation bar, that is shared by every page in the application. Besides the elements that are offered in the home menu, we also have three elements that are displayed at all times in the top right part of the page: the name of the user that is logged in, the number of coins that the user has and a bell icon, that is a

link that is the connection with the news created through the administrator application and can be accessed when the user wants to see the announcements.

_Layout.cshtml - Shared

```
<ul class="navbar-nav flex-grow-1">
    <li class="nav-item">
        <a class="nav-link" asp-area="" asp-controller="Home" asp-action="Index">Home</a>
    </li>
    <li class="nav-item">
        <a class="nav-link" asp-area="" asp-controller="Squad" asp-action="Editor">Squad</a>
    </li>
    <li class="nav-item">
        <a class="nav-link" asp-area="" asp-controller="Transfer"
        ↪  asp-action="TransferList">TransferMarket</a>
    </li>
    <li class="nav-item">
        <a class="nav-link" asp-area="" asp-controller="Home" asp-action="Store">Store</a>
    </li>
    <li class="nav-item">
        <a class="nav-link" asp-controller="Game" asp-action="GameMenu">Game</a>
    </li>
    <li class="nav-item">
        <a class="nav-link" asp-area="" asp-controller="Home" asp-action="Standings">Standings</a>
    </li>
    <li class="nav-item">
        <a class="nav-link" asp-area="" asp-controller="Home" asp-action="LogOut">Log Out</a>
    </li>
</ul>
<ul class="nav justify-content-end">
    <li class="nav justify-content-end" style="margin-right: 10px;">
        <img height=20 width=20 src="~/css/images/iconUsername.png">
        <span class="badge bg-secondary">@ViewBag.username</span>
    </li>
    <li class="nav justify-content-end">
        <img height=20 width=20 src ="~/css/images/coins.webp">
        <span class="badge bg-secondary">@ViewBag.wallet</span>
    </li>
    <li class="nav justify-content-end">
        <a style="text-decoration:none; margin-top: -20%; margin-right: -40%;" asp-action="News"
        ↪  asp-controller="Home">
            <img height=20 width=20 src="~/css/images/iconBell.png">
        </a>
    </li>
</ul>
```

5.3.3 News

For the first time in this Client application presentation we can see a model transmitted to a view in the first line of the code below. That model is transmitted to the view by the controller and can be accessed as it can be used to access the objects using C# injection in the HTML page using the '@' character. We do this to display all the news that have been sent by the controller.

News.cshtml - Home

```
@model List<News>

@{
    ViewData["Title"] = "News";
}

<head>
    <link rel="stylesheet" href="~/css/site.css" />
</head>

```

```
<bod>
    <div class="BuyPlayerContainer">
        <h3>News</h3>
            @foreach (var item in Model)
            {
                <p style="border-bottom: solid 1px white; color:white; margin: 3%; text-align:
                ↪  center; padding:1%;">
                    @item.Post
                </p>
            }
        <br />
        <a asp-action="Index" asp-controller="Home" class="accessButton"
        ↪  style="text-decoration:none">Back to home</a>
    </div>
</bod>
```

The controller method that returns the News View is part of the Home Controller and can be seen below. Also, the result is displayed in the printscreen below.

HomeController.cs - Home

```
[HttpGet]
// Returns a view that displays news
public IActionResult News()
{
    // Get all news from the database
    var news = _context.News.ToList();

    var userName = GetUserName(HttpContext);
    // Store username to viewbag (for the NavBar)
    ViewBag.UserName = userName;
    // Store the nr of coins to viewbag (for the NavBar)
    ViewBag.wallet = getMoney();

    return View(news);
}
```

Figure 11: News page

5.4 Transfer market

The Transfer Market is the place the users are offered the opportunity to buy and sell players directly from other players.

5.4.1 Transfer Controller

The code from the TransferController.cs can be found below. This controller offers five main methods, which are described in more detail below, together with print screens of the results.

TransferController

```
public class TransferController : Controller
    {
        private readonly FootballDatabaseContext _context;

        public TransferController(FootballDatabaseContext context)
        {
            _context = context;
        }

        [HttpGet]
        public IActionResult BuyPlayer(int transferId)
        {
            //Get the id of the user that is currently logged in
            try
            {
                var username = HomeController.GetUserName(HttpContext);
                var user = _context.Users.FirstOrDefault(u => u.Username == username);
                int userID = user.UserId;

                // Check if the user has enough money to buy the player
                var happeningTransfer = _context.Transfers.FirstOrDefault(t => t.TransferId ==
↪  transferId);

                if (user.Coins < happeningTransfer.TransferFee)
                {
                    return RedirectToAction("TransferError");
                }
                else
                {
                    //Remove the transfer from the available transfers by attaching a buying user
                    happeningTransfer.BuyingUserId = userID;

                    // Update the player's contract so that the player is now owned by the buying
↪  user
                    var player = _context.Players.FirstOrDefault(p => p.PlayerId ==
↪  happeningTransfer.PlayerId);
                    var teamContract = _context.TeamContracts.FirstOrDefault(tc => tc.PlayerId ==
↪  player.PlayerId && tc.SquadId == happeningTransfer.SellingUserId);

                    teamContract.SquadId = userID;
                    teamContract.Position = 19; // 19 is the position for a player that does not
↪  have a role in the team

                    // Solve all money issues (selling user gets money, buying user loses money)
                    var sellingUser = _context.Users.FirstOrDefault(u => u.UserId ==
↪  happeningTransfer.SellingUserId);
                    user.Coins -= happeningTransfer.TransferFee;
                    sellingUser.Coins += happeningTransfer.TransferFee;

                    // Add the username to the viewbag
                    ViewBag.username = HomeController.GetUserName(HttpContext);
                    // Updates coins viewbag fo coin display
                    ViewBag.wallet = user.Coins;

                    _context.SaveChanges();
                }
            }
            catch (NullReferenceException)
            {
```

```
                return RedirectToAction("Login", "Access"); // Should only happen if the user is not logged in
            }
            return View();
        }

        // View if the user doesn't have enough coins
        public IActionResult TransferError()
        {
            // Add nr of coins to the wallet
            ViewBag.wallet = getMoney();
            // Add the username to the viewbag
            ViewBag.username = HomeController.GetUserName(HttpContext);
            return View();
        }

        [HttpGet]
        public IActionResult TransferList(string searchString)
        {
            //Perform database query to retrieve all unsold Transfers from transfer table

            // All unsold transfers
            var transfers = _context.Transfers
                .Include(t => t.Player)
                    .ThenInclude(p => p.TeamContracts)
                .Include(t => t.SellingUser)
                .Where(t => t.BuyingUserId == null)
                .ToList();

            if (!String.IsNullOrEmpty(searchString))
            {
                // Update: Now include only players that have searchString in their name
                transfers = _context.Transfers
                    .Include(t => t.Player)
                        .ThenInclude(p => p.TeamContracts)
                    .Include(t => t.SellingUser)
                    .Where(t => t.BuyingUserId == null && (t.Player.FirstName.Contains(searchString) || t.Player.LastName.Contains(searchString)))
                    .ToList();
            }

            // Add nr of coins to the wallet
            ViewBag.wallet = getMoney();

            // Add the username to the viewbag
            ViewBag.username = HomeController.GetUserName(HttpContext);

            return View(transfers);
        }

        [HttpGet]
        // Gets players that can be added to the transfer list
        public IActionResult SellPlayer()
        {
            //Get the id of the user that is currently logged in
            try
            {
                var username = HomeController.GetUserName(HttpContext);
                var user = _context.Users.FirstOrDefault(u => u.Username == username);
                int userID = user.UserId;

                //Get all players that are owned by the user
                var players = _context.Players
```

```
                    .Include(p => p.TeamContracts)
                    .ThenInclude(tc => tc.Squad)
                    .Where(p => p.TeamContracts.Any(tc => tc.SquadId == userID));

                // Get the players that are not in the first team
                List<Player> playersNotInFirstTeam = players.Where(p => p.TeamContracts.Any(tc =>
↪  tc.Position == 19 && tc.SquadId == userID)).ToList();

                ViewData["Players"] = new SelectList(playersNotInFirstTeam, "PlayerId", "Name");

                // Add nr of coins to the wallet
                ViewBag.wallet = getMoney();
                // Add the username to the viewbag
                ViewBag.username = HomeController.GetUserName(HttpContext);


                return View(playersNotInFirstTeam);
            }
            catch (NullReferenceException)
            {
                return RedirectToAction("Login", "Access"); // Should only happen if the user is
↪  not logged in
            }
        }

        [HttpPost]
        // Puts a player on the transfer list
        public IActionResult ListPlayer(Transfer transfer)
        {
            //Get the id of the user that is currently logged in
            try
            {
                var username = HomeController.GetUserName(HttpContext);
                var user = _context.Users.FirstOrDefault(u => u.Username == username);
                int userID = user.UserId;

                //Add the transfer to the transfer table
                transfer.SellingUserId = userID;
                transfer.SellingUserId = userID;

                // Generate a new transfer id
                var transferIds = _context.Transfers.Select(t => t.TransferId).ToList();
                int newTransferId = 0;
                foreach (int id in transferIds)
                {
                    if (id > newTransferId)
                    {
                        newTransferId = id;
                    }
                }

                transfer.TransferId = newTransferId + 1;

                _context.Transfers.Add(transfer);

                // Update the player's contract so that the player is now on the transfer list
↪  (position 20)
                var teamContract = _context.TeamContracts.FirstOrDefault(tc => tc.PlayerId ==
↪  transfer.PlayerId && tc.SquadId == userID);
                if (teamContract != null)
                {
                    teamContract.Position = 20;
                }
```

```
174
175                 _context.SaveChanges();
176             }
177             catch (NullReferenceException)
178             {
179                 return RedirectToAction("Login", "Access"); // Should only happen if the user is
    ↪  not logged in
180             }
181             return RedirectToAction("TransferList");
182         }
183
184         public int getMoney()
185         {
186             var userName = HomeController.GetUserName(HttpContext);
187             var user = _context.Users.FirstOrDefault(u => u.Username == userName);
188
189             return user.Coins;
190         }
191     }
```

5.4.2 Explications and results

- **TransferList(searchString: string)** - A method that returns a view which uses a list of all players that are available on the transfer list. The searchString can be transmitted through the page URL (using a specific input field from the view of course) and it allows the user to search for a specific player using his name.

Figure 12: Transfer List

- **BuyPlayer(transferId: int)** - This function is responsible for all the necessary modifications in the database, and is called whenever a user clicks the buy button for a player in the transfer list.

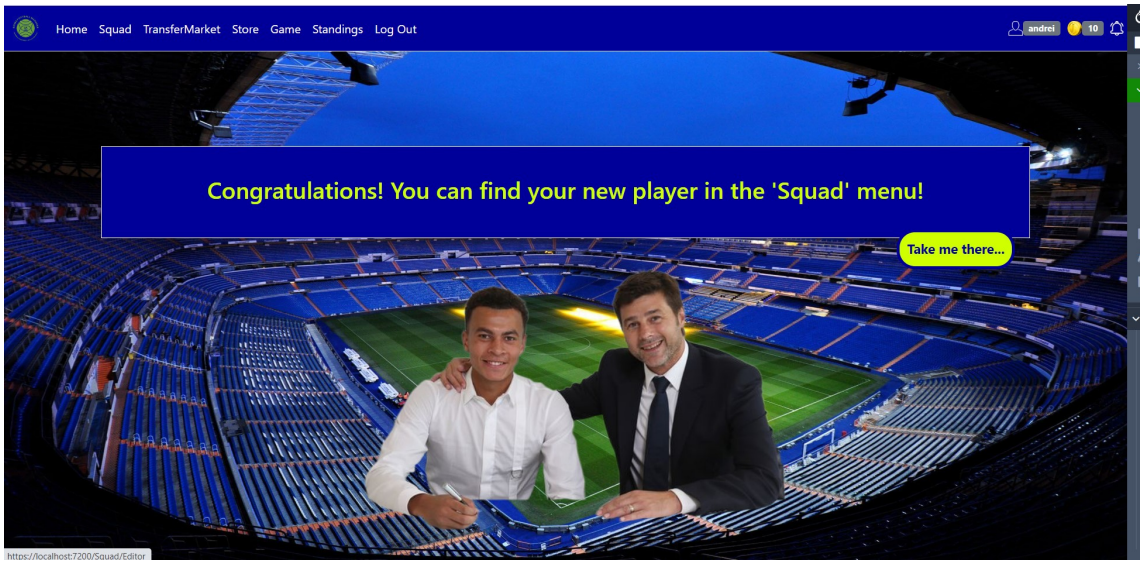

Figure 13: Buy Player

- **SellPlayer()** - Returns a view that allows a user to select one of the players not included in the first team and list them on the transfer list for his desired transfer sum.

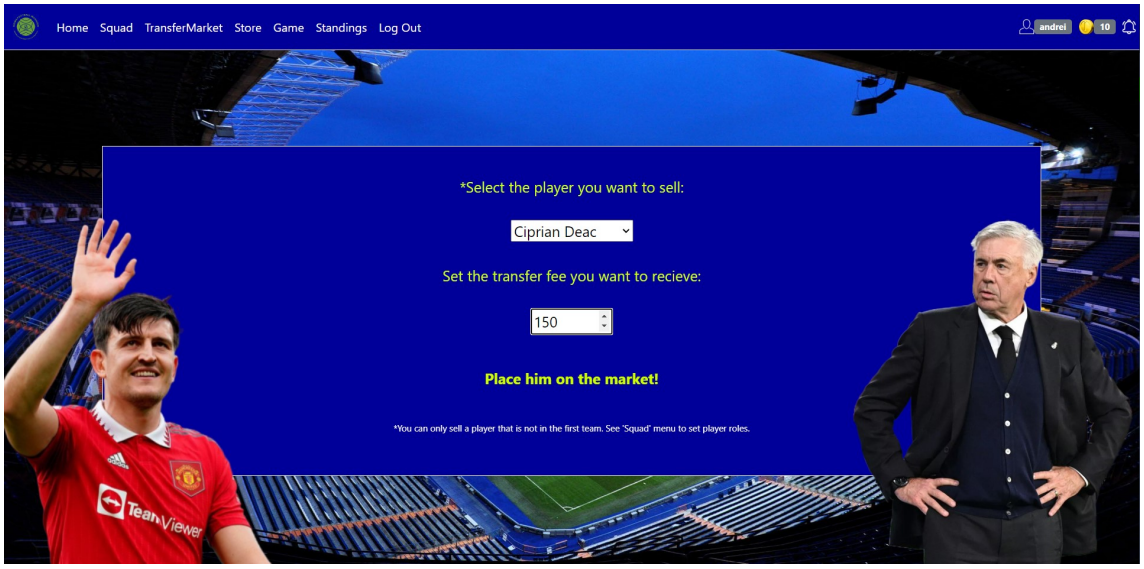

Figure 14: Sell Player

- **ListPlayer(transfer: Transfer** - It is a post method that deals with the required mechanism for actually putting a player on the transfer list. It makes the player available to others to see by inserting a new transfer in the database and it makes the player unavailable for the seller's squad.
- **TransferError()** - A simple method that just returns a View which is displayed whenever a user tries to buy a player for which he doesn't have enough coins.

Figure 15: Transfer Error

5.5 Squad editor

The squad editor is the place where each user can visualise his teams line up, and choose via drag and drop functionality the role each player has in the team.
The SquadController.cs file contains a single get method that returns all the players as a list. The specific view, will then take those players and print them in a list, that is styled using CSS to position each player in the right spot.

SquadController

```
[HttpGet]
public async Task<IActionResult> Editor()
{
    //Get the id of the user that is currently logged in
    try
    {
        var username = HomeController.GetUserName(HttpContext);
        var user = _context.Users.FirstOrDefault(u => u.Username == username);
        int userID = user.UserId;

        // Perform database query to retrieve players from Players table that are linked to the squad of the current user
        var players = _context.Players
            .Include(p => p.TeamContracts)
            .ThenInclude(tc => tc.Squad)
            .ThenInclude(s => s.User)
            .Where(p => p.TeamContracts.Any(tc => tc.SquadId == userID))
            .Select(p => new Team
        {
            Players = new List<Player> { p },
            Contracts = p.TeamContracts.Where(tc => tc.SquadId == userID).ToList()
        })
        .ToList();

        // Add nr of coins to the wallet
        ViewBag.wallet = getMoney();
        ViewBag.username = username;
        return View(players);
    }
    catch (NullReferenceException)
    {
        return RedirectToAction("Login", "Access");
    }
```

```
33  }
```

It can be observed in the code above that we need information about the players from two different tables: TeamContracts and Players, so we had to build a new model of type Team, that holds information about both a player and his contract. The contract determines which players are connected with which teams, and also the positions they occupy. Positions are integers from 1 to 11 for starters, 11 to 18 for reserves and 19 for any other player that is not in the first team. Positions 20 are reserved for players that are placed on the transfer market and are unavailable for selection, therefore the user cannot find the player in this menu. With just this code, and a view that prints the players using CSS, we can obtain the following output:

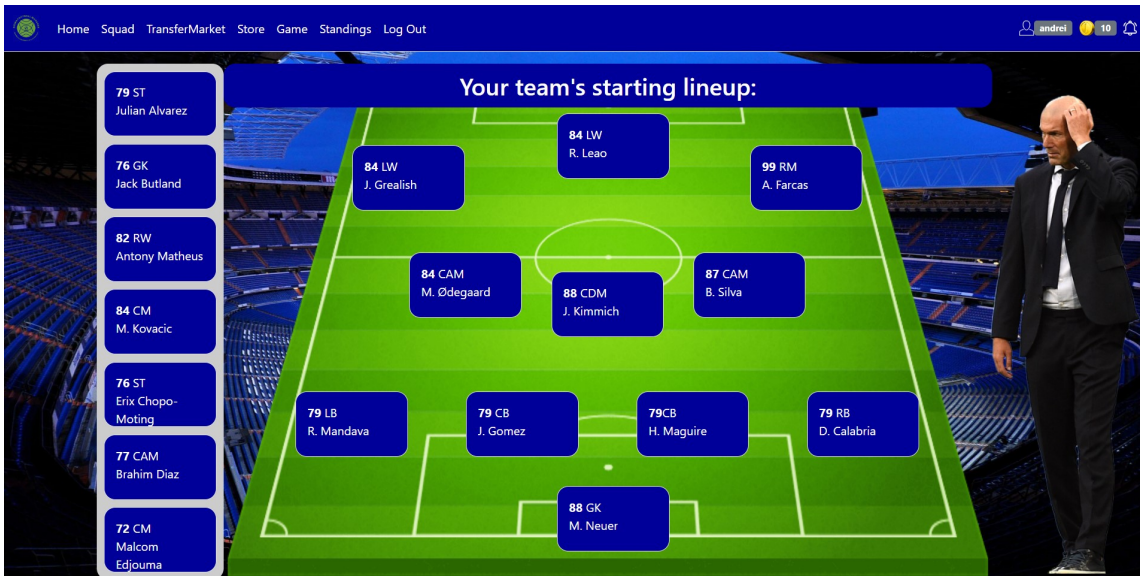

Figure 16: Squad menu

However, our job doesn't end here. We need also Java Script code that allows the user to interact with the listed objects. Below, you can observe how we managed to make each object drag-able and how we called a method from a 'ContractController' that has to update positions of the players whenever the user decides to change the lineup of his team.

squad.js

```
function slist(target) {
    // Gets the list of players and adds the css class to it
    target.classList.add("slist");

    let items = target.getElementsByTagName("li"), current = null;

    for (let i of items) {
        //Make all items draggable
        i.draggable = true;

        // On drag start change color of all items except the one being dragged
        i.ondragstart = e => {
            current = i;
            for (let it of items) {
                if (it != current) { it.classList.add("hint"); }
            }
        };

        // Adds red highlight when drag enters the dropzone
        i.ondragenter = e => {
            if (i != current) { i.classList.add("active"); }
        };

        // Removes red highlight when drag leaves the dropzone
        i.ondragleave = () => i.classList.remove("active");

        // Removes all highlight when drag ends
        i.ondragend = () => {
            for (let it of items) {
                it.classList.remove("hint");
                it.classList.remove("active");
            }
        };

        // Prevents default action of the browser so i can implement mine
        i.ondragover = e => e.preventDefault();

        // On drop, if the item being dropped is not the same as the item being dragged, then swap
        ↪ them
        i.ondrop = e => {
            e.preventDefault();
            if (i != current) {
                // Gets the position of the current item and the item being dropped
                let currentpos = 0, droppedpos = 0;
                for (let it = 0; it < items.length; it++) {
                    if (current == items[it]){
                        currentpos = it; // current = item that is being dropped on
                    }
                    if (i == items[it]){
                        droppedpos = it; // i = dragged item
                    }
                }

                // Get the IDs of the contracts to be updated
                const currentContractId = current.getAttribute('data-ContractId');
                const droppedContractId = i.getAttribute('data-ContractId');

                $.ajax({
```

```
                    type: 'POST',
                    url: '/Contract/UpdatePositions',
                    data: {
                        currentContractId: currentContractId,
                        droppedContractId: droppedContractId
                    },
                    success: function (data) {
                        // Update the player positions dynamically

                        // Swap the html attributes of the two items
                        const tempHTML = current.innerHTML; // stores the current innerHTML of the
                        ↪  current item (all html and csss attributes)
                        current.innerHTML = i.innerHTML;
                        i.innerHTML = tempHTML;

                        // Ensures that the players have the correct data-ContractId attribute
                        current.setAttribute("data-ContractId", droppedContractId);
                        i.setAttribute("data-ContractId", currentContractId);
                    },
                    error: function () {
                        alert("An error occurred while updating positions.");
                    }
                });


            }
        };
    }
}
```

The code above makes an ajax call to an UpdatePositions functions that is responsible for inter-swapping positions of two players whenever they are dropped one over another and can be seen below.

ContractController

```
[HttpPost]
public async Task<IActionResult> UpdatePositions(int currentContractId, int droppedContractId)
{
    // Try to see if user is logged in
    try
    {
        var username = HomeController.GetUserName(HttpContext);

        // Find the contracts to swap by their IDs
        var currentContract = _context.TeamContracts.SingleOrDefault(c => c.ContractId ==
↪  currentContractId);
        var droppedContract = _context.TeamContracts.SingleOrDefault(c => c.ContractId ==
↪  droppedContractId);

        if (currentContract == null || droppedContract == null)
        {
            return BadRequest("One or both of the contracts could not be found.");
        }

        // Swap the positions of the contracts
        var tempPosition = currentContract.Position;
        currentContract.Position = droppedContract.Position;
        droppedContract.Position = tempPosition;

        // Save the changes to the database
        try
        {
            _context.SaveChanges();
```

```
        }
        catch (Exception ex)
        {
            return BadRequest($"An error occurred while saving the changes: {ex.Message}");
        }

        // Return the same page so I can keep the login redirect in case the user is not logged in
        return RedirectToAction("Editor", "Squad");
    }
    catch (NullReferenceException)
    {
        return RedirectToAction("Login", "Access");
    }
}
```

The result of applying the java script code can be seen in a figure below, even though the real life perception is far more magical.

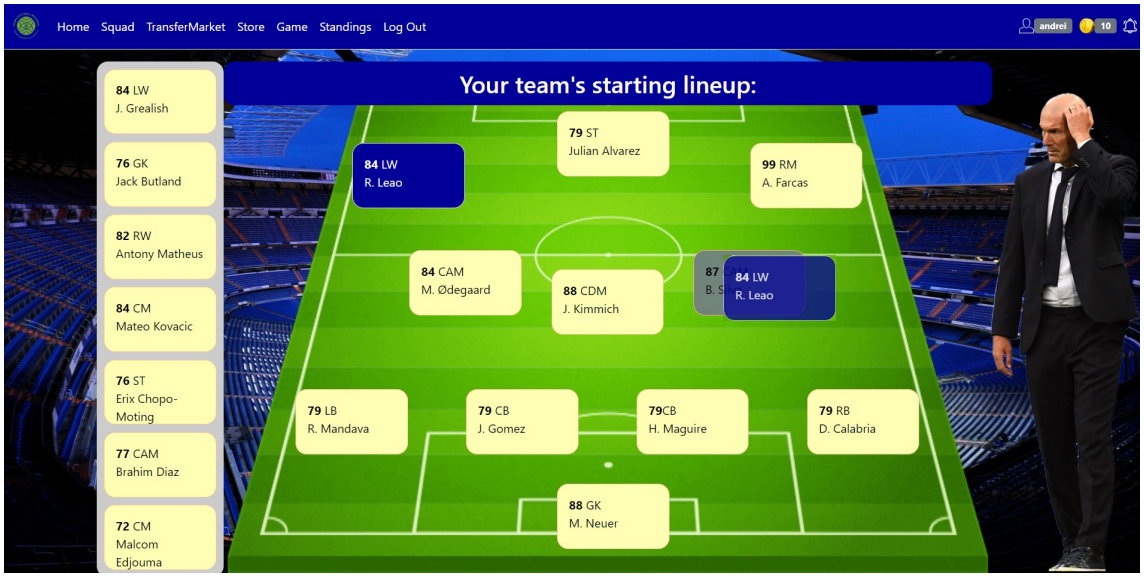

Figure 17: Drag and drop functionality

5.6 Store

Visiting the store greets the user with three packs containing five randomly selected players depending on the pack of their choosing. For the "Basic Pack", as the name suggests, is the cheapest of the three, however the chances of receiving a lower overall ranking player are higher than medium or high ranking ones. The "Silver Pack", with a price in between the cheapest and the most expensive pack, prioritizes players with a medium overall rank. The most expensive pack of them all, "Premium Pack", has the highest chances of receiving players with a high overall rank.

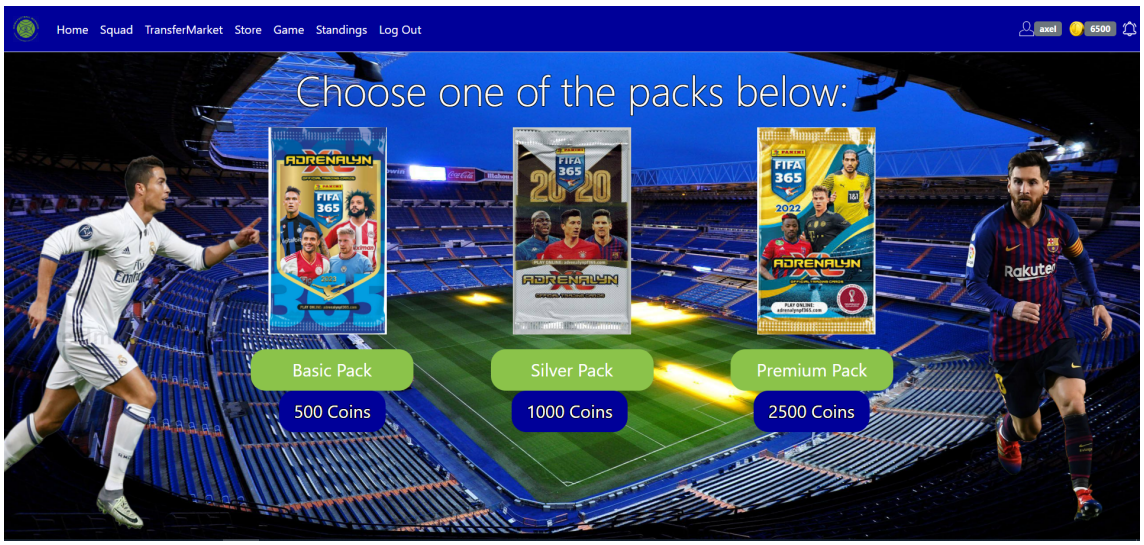

Figure 18: The Store Page

The store page works through a combination of C# and Javascript code. Firstly, these three controller functions select the players from the database based on their overall ranks.

HomeController - getCheapPlayers() getRegularPlayers() getExpensivePlayers()

```
public List<Player> getCheapPlayers()
{
    List<Player> list = new List<Player>();
    List<Player> plist = _context.Players.ToList();

    foreach (Player p in plist)
    {
        if(p.OverallRank <= 75)
        {
            list.Add(p);
        }
    }
    return list;
}

public List<Player> getRegularPlayers()
{
    List<Player> list = new List<Player>();
    List<Player> plist = _context.Players.ToList();

    foreach (Player p in plist)
    {
        if (p.OverallRank > 75 && p.OverallRank < 80)
        {
            list.Add(p);
        }
    }
    return list;
}

public List<Player> getExpensivePlayers()
{
    List<Player> list = new List<Player>();
    List<Player> plist = _context.Players.ToList();

    foreach (Player p in plist)
    {
        if (p.OverallRank >= 80)
        {
```

```
            list.Add(p);
        }
    }
    return list;
}
```

The lists returned by the functions are then passed into the ViewBag so that they can be further passed to the Javascript functions.

Store.cshtml - Getting Variables From The ViewBag

```
@{
    ViewData["Title"] = "Shop";
    var cheapJson = Json.Serialize(ViewBag.cheapPlayers);
    var regularJson = Json.Serialize(ViewBag.regularPlayers);
    var expensiveJson = Json.Serialize(ViewBag.expensivePlayers);
}
<script>
    var cheapDeck = @cheapJson
    var regularDeck = @regularJson
    var expensiveDeck = @expensiveJson
    var wallet = @ViewBag.wallet;
</script>
```

In the **store.js** file, details for the types of packs are defined, such as the pack's price, and the chance to get players of every category. Event listeners are also added to the buttons.

store.js - Defining Pack Information + Event Listners

```
const cheapPack = {
    price: 500,
    rarity: 0,
    commonChance: 0.75,
    regularChance: 0.2,
    rareChance: 0.05
}

const regularPack = {
    price: 1000,
    rarity: 1,
    commonChance: 0.15,
    regularChance: 0.70,
    rareChance: 0.15
}

const expensivePack = {
    price: 2500,
    rarity: 2,
    commonChance: 0.05,
    regularChance: 0.25,
    rareChance: 0.70
}
cheapButton.addEventListener("click", function () { buyDeck(cheapPack) })
regularButton.addEventListener("click", function () { buyDeck(regularPack) })
expensiveButton.addEventListener("click", function () { buyDeck(expensivePack) })
```

When purchasing a pack, the **buyDeck()** function first checks if the user has enough money to buy it, alerting the user if they do not have enough funds for that action. Next, it will call the **updateMoney()** method from the controller using **AJAX** which will update the user's funds with the value after purchasing. Then it will call the **selCards()** method in order to get 5 randomly selected players based on the pack's chance to pull a player of each type. The **playerInDb()** function notifies the controller which players were previously selected so that the controller can add the specified players to the user's team. Finally, the **displayCards()** function, as the name suggests, displays which cards the user recieved.

store.js - buyDeck()

```
function buyDeck(pack) {
    if (wallet < pack.price) {
        alert("You do not have enough money to buy this pack!");
    }
    else {
        wallet = wallet - pack.price;

        $.ajax({
            url: "/Home/updateMoney",
            type: "POST",
            data: { value: wallet },
            success: function (response) {
                console.log("Success:", response);
            },
            error: function (xhr) {
                console.log("Error:", xhr.responseText);
            }
        });

        document.getElementById("coin-counter").innerHTML=wallet

        var cardsGot = selCards(pack.commonChance, pack.regularChance, pack.rareChance)
        console.log(cardsGot)
        playerInDb(cardsGot)
        displayCards(cardsGot)
    }

}
```

The **selCards()** function first selects which type of player will be received and then tries to select a random player from the appropiate list. If the player has already been selected, meaning that the player is in the **found[]** list, to avoid receiving duplicate players, the function tries again until it finds a player that has not been received yet. At that point it puts the player into the **found[]** list and pushes to the **opened[]** list an object containing the name of the player and the relevant statistics to display on the card, the id of the player to pass to the controller and the color of the displayed card. Finally, it returns the **opened[]** list.

store.js - selCards()

```
function selCards(cchance, rchance, echance) {
    var opened = [];
    var found = [];
    let tried;
    for (var i = 0; i < 5; i++) {
        var player;
        tried = closest(Math.random(), cchance, rchance, echance);
        if (tried == "commonPlayer") {
            player = cheapDeck[Math.floor(Math.random() * (cheapDeck.length))]
            if (found.includes(player)) {
                while (found.includes(player)) {
                    player = cheapDeck[Math.floor(Math.random() * (cheapDeck.length))]
                }
            }
            found.push(player)

            opened.push({
                fName : player.firstName,
                lName : player.lastName,
                atk : player.attacking,
                mct: player.midfieldControl,
                def : player.defending,
                pid : player.playerId,
                col: '#CD7F32'
            })
```

```
        }
        else if (tried == "regularPlayer") {
            player = regularDeck[Math.floor(Math.random() * (regularDeck.length))]
            if (found.includes(player)) {
                while (found.includes(player)) {
                    player = regularDeck[Math.floor(Math.random() * (regularDeck.length))]
                }
            }
            found.push(player)
            opened.push({
                fName: player.firstName,
                lName: player.lastName,
                atk: player.attacking,
                mct: player.midfieldControl,
                def: player.defending,
                pid : player.playerId,
                col: '#979A9A'
            })

        }
        else if (tried == "rarePlayer") {
            player = expensiveDeck[Math.floor(Math.random() * (expensiveDeck.length))]
            if (found.includes(player)) {
                while (found.includes(player)) {
                    player = expensiveDeck[Math.floor(Math.random() * (expensiveDeck.length))]
                }
            }
            found.push(player)
            opened.push({
                fName : player.firstName,
                lName : player.lastName,
                atk : player.attacking,
                mct : player.midfieldControl,
                def : player.defending,
                pid : player.playerId,
                col : '#FFD700'
            })
        }

    }
    return opened
    found = [];
    opened = [];
}
```

The **playerInDb()** function creates a list with the player's IDs and then uses **AJAX** to pass the list to the controller's **addPlayerTeam()** method, which creates an entry into the **TeamContracts** table with the information corresponding to the user's squad and to the received player.

store.js - playerInDb()

```
function playerInDb(plyrs) {
    var pidList = []
    for (var i = 0; i < 5; i++) {
        pidList.push(plyrs[i].pid)
    }
    $.ajax({
        url: "/Home/addPlayerTeam",
        type: "POST",
        data: { playerIds: pidList },
        success: function (response) {
            console.log("Success:", response);
        },
        error: function (xhr) {
```

```
            console.log("Error:", xhr.responseText);
        }
    });
}
```

HomeController - addPlayerTeam()

```
[HttpPost]
public IActionResult addPlayerTeam(int[] playerIds)
{
    var userName = GetUserName(HttpContext);
    var user = _context.Users.FirstOrDefault(u => u.Username == userName);
    var squad = _context.Squads.FirstOrDefault(s => s.UserId == user.UserId);
    var contract = new TeamContract();

    int cNumber = _context.TeamContracts.Count() + 1;

    foreach(int id in playerIds)
    {
        contract.ContractId = cNumber;
        contract.SquadId = squad.SquadId;
        contract.PlayerId = id;
        contract.ShirtNumber = null;
        contract.IsCaptain = null;
        contract.Position = 19;
        cNumber++;
^^I_context.TeamContracts.Add(contract);
^^I_context.SaveChanges();
    }
    return Json(new { result = "success" });
}
```

To display the cards the user receives, the **displayCards()** function creates inside an empty div a semi-transparent overlay on the page and creates five divs containing the card's information and displays them on top of the overlay. It also creates a button that when pressed, calles the **clearCardView()** function that clears the div that contains everything created by the **displayCards()** function.

store.js - displayCards() clearCardView()

```
function displayCards(plyrs) {
    var tmp = ""
    for (var i = 0; i < 5; i++) {
        tmp = tmp + "<div class=\"rectangle\" style=\"background-color:" + plyrs[i].col + "\" >" +
        ↪  plyrs[i].fName + "</br>" + plyrs[i].lName + "</br></br>Attacking:" + plyrs[i].atk +
        ↪  "</br>MidControl:" + plyrs[i].mct + "</br>Deffensive:" + plyrs[i].def +" </div>"
    }
    cardView.innerHTML = "<div class=\"overlay\"><div style=\"position: fixed;left: 15%;top:
    ↪  15%;width: 100 %;height: 100 %; \"> " + tmp + "</br><button id=\"close\"
    ↪  style=\"z-index:30;\" onclick=\"clearCardView()\">Back to shop</button></div></div>"
}
function clearCardView() {
    cardView.innerHTML = "";
}
```

Figure 19: The Store Page After Buying A Pack

5.7 Standings

The standings page is basically a leaderboard. Each user receives a 10 trophies for a win and lose 3 trophies for each loss. The manipulation of data happens inside the Simulation Controller after each match. Afterwards, this data is processed inside a get method names Standings, which sorts the players by their points and delivers them to a view that displays them as a table.

Standings() - HomeController.cs

```
[HttpGet]
// Returns a view that displays the leaderboard table
public IActionResult Standings()
{
    // Get all standings from the database
    List<Standing> standings = _context.Standings.ToList();

    // Order the standings by trophies
    standings = standings.OrderByDescending(s => s.Trophies).ToList();

    // Save the position in the list to the standing position
    int position = 1;
    foreach(Standing item in standings)
    {
        item.Position = position;
        position++;
    }

    foreach(Standing item in standings)
    {
        // Bind the user to his standing element
        User user = _context.Users.FirstOrDefault(u => u.UserId == item.UserId);
        item.User = user;
    }

    var userName = GetUserName(HttpContext);
    // Store username to viewbag
    ViewBag.UserName = userName;
    // Store the nr of coins to viewbag
    ViewBag.wallet = getMoney();

    return View(standings);
}
```

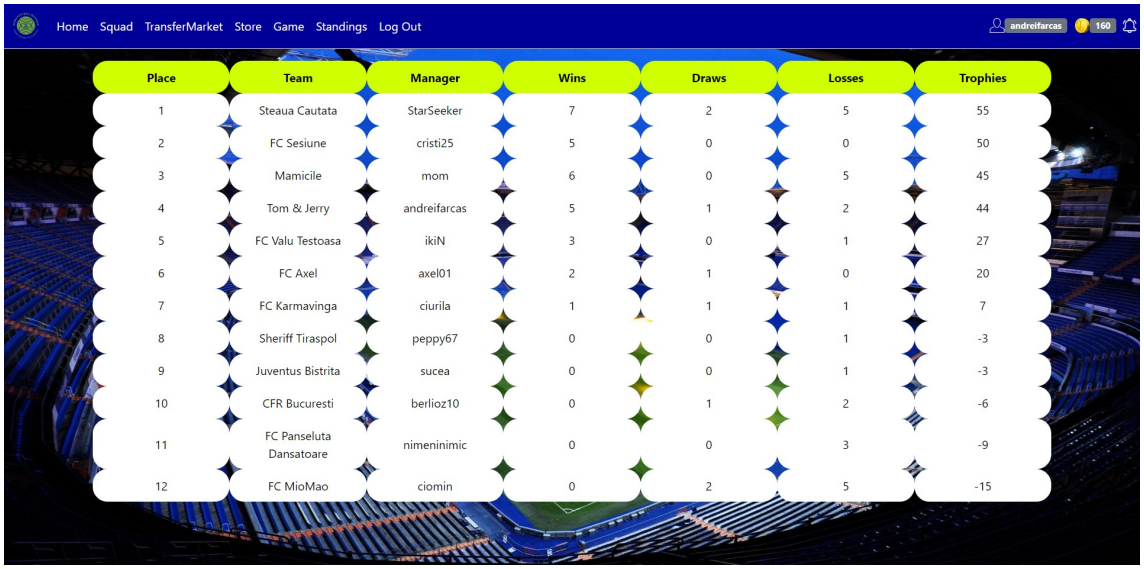

Figure 20: Standings page

5.8 Game

5.8.1 Matchmaking

The game functionality is one of the most complex parts of this project. The first part of simulating games between teams that belong to two users would be to build a matchmaking system.
The first contact the user has with the game process is a page where he has to select the mentality of his team and then he can press a button to look for an opponent. The way we have build our matchmaking is by redirecting the first user that searches for a match to a waiting page, and checking a flag variable every 0.5 seconds. This flag is set as true when another user wants to play a game and a match is created for them. After that, we make sure that both users get redirected to the same game page and can watch their teams perform.

GameController.cs

```
public class GameController : Controller
{

    private readonly FootballDatabaseContext _context;
    private readonly Dictionary<int, string> _userConnections = new Dictionary<int, string>();

    public GameController(FootballDatabaseContext context)
    {
        _context = context;
    }

    // This is a list of all the users that are waiting for a match
        private static readonly List<int> _waitingRequests = new List<int>();
        private static readonly List<int> _waitingMentalities = new List<int>();

    // This is a flag that will be used to notify the user that a match has been found
    private static readonly List<MatchmakingWait> waitFlags = new List<MatchmakingWait>();

    [HttpPost]
    public async Task<IActionResult> JoinMatchmakingAsync(int mentality)
    {
        // Get the id of the user that is currently logged in
        var username = HomeController.GetUserName(HttpContext);
        var user = _context.Users.FirstOrDefault(u => u.Username == username);
        int userID = user.UserId;

        // If user is already in the waiting list, redirect him to the matchmaking page
        if (_waitingRequests.Contains(userID))
```

```
            return RedirectToAction("GameRoom");

        lock (_waitingRequests)
        {
            // Add the user to the waiting list
            _waitingMentalities.Add(mentality);
            _waitingRequests.Add(userID);

            if (_waitingRequests.Count >= 2)
            {
                // The first two users in the list will be matched
                int user1 = _waitingRequests[0];
                int user2 = _waitingRequests[1];
                int mentality1 = _waitingMentalities[0];
                int mentality2 = _waitingMentalities[1];

                // Remove the users from the list
                _waitingRequests.RemoveAt(0);
                _waitingRequests.RemoveAt(0);
                _waitingMentalities.RemoveAt(0);
                _waitingMentalities.RemoveAt(0);

                // Create a new match for them
                int matchId = CreateMatch(user1, user2, mentality1, mentality2);

                // Notify user1 that a match has been found using the existing flag with his id
                MatchmakingWait waitFlag1 = waitFlags.FirstOrDefault(w => w.UserId == user1);
                waitFlag1.MatchId = matchId;
                waitFlag1.MatchFound = true;

                // Redirect the users to the game session
                return RedirectToAction("GameSession", new { matchId });
            }
        }

        // Add the user to he list of flags that will be used to notify him that a match has been
↪   found
        MatchmakingWait waitFlag = new MatchmakingWait();
        waitFlag.UserId = userID;
        waitFlag.MatchFound = false;
        waitFlags.Add(waitFlag);

        return RedirectToAction("GameRoom", "Game");
    }

    private int CreateMatch(int user1, int user2, int mentality1, int mentality2)
    {
        // Create a new match
        // Generate a new id for the match using Match model
        Match match = new Match();

        // If the match table is empty, the id will be 1, otherwise it will be the max id + 1
        if (_context.Matches.Count() == 0)
            match.MatchId = 1;
        else
             match.MatchId = _context.Matches.Max(m => m.MatchId) + 1;

        // Add the two users to the match
        match.HomeTeamId = user1;
        match.AwayTeamId = user2;

        // Add a random stadium to the match
        Random rnd = new Random();
```

```
        int stadiumId = rnd.Next(1, _context.Stadiums.Count() + 1);
        match.StadiumId = stadiumId;

        // Add a random referee to the match
        int refereeId = rnd.Next(1, _context.Referees.Count() + 1);
        match.RefereeId = refereeId;

        match.HomeMentality = mentality1;
        match.AwayMentality = mentality2;

        // Add the match to the database
        _context.Matches.Add(match);
        _context.SaveChanges();

        // Simulate the match
        SimulationController simulation = new SimulationController(_context);
        simulation.Simulation(match);

        return match.MatchId;
    }

    // The actual simulation, both users will be redirected to this page
    [HttpGet]
    public async Task<IActionResult> GameSession(string matchId)
    {
        int match_Id = Int32.Parse(matchId);
        List<string> events = new List<string>();

        // Get the id's of the two users that are playing the match
        Match match = _context.Matches.FirstOrDefault(m => m.MatchId == match_Id);
        int user1 = match.HomeTeamId;
        int user2 = match.AwayTeamId;

        // Bind the two users to the match
        match.AwayTeam = _context.Users.FirstOrDefault(u => u.UserId == user2);
        match.HomeTeam = _context.Users.FirstOrDefault(u => u.UserId == user1);

        // Bind the stadium to the match
        match.Stadium = _context.Stadiums.FirstOrDefault(s => s.StadiumId == match.StadiumId);

        // Bind the referee to the match
        match.Referee = _context.Referees.FirstOrDefault(r => r.RefereeId == match.RefereeId);

        // Retrieve serialized events from the database
        string serializedEvents = match.Events;
        events = JsonConvert.DeserializeObject<List<string>>(serializedEvents);

        ViewBag.Events = events;

        // For shared view
        ViewBag.wallet = getMoney();
        ViewBag.username = HomeController.GetUserName(HttpContext);

        return View(match);
    }

    // Simple menu for the user to select the mentality of his team and join a matchmaking queue
    [HttpGet]
    public IActionResult GameMenu()
    {
        // Add the username to the viewbag
        ViewBag.username = HomeController.GetUserName(HttpContext);

```

```
        // Add nr of coins to the wallet
        ViewBag.wallet = getMoney();


        return View();
    }

    // Waiting page for the user
    [HttpGet]
    public IActionResult GameRoom()
    {
        // Get the flag variable for the user
        var username = HomeController.GetUserName(HttpContext);
        ViewBag.userame = username;

        User user = _context.Users.FirstOrDefault(u => u.Username == username);
        int userID = user.UserId;

        MatchmakingWait waitFlag = waitFlags.FirstOrDefault(w => w.UserId == userID);

        // Add nr of coins to the wallet
        ViewBag.wallet = getMoney();

        return View(user);
    }


    // Returns a flag for the user waiting in GameRoom
    [HttpGet]
    public IActionResult CheckMatchStatus(int userId)
    {
        // Get the flag variable for the user
        MatchmakingWait waitFlag = waitFlags.FirstOrDefault(w => w.UserId == userId);

        if (waitFlag != null)
        {
            // Check the flag variable to determine if the match has been found
            bool matchFound = waitFlag.MatchFound;
            // return the flag variable and the match id
            return Json(new { matchFound, matchId = waitFlag.MatchId });
        }
        else
        {
            return Json(new { matchFound = false });
        }
    }

    // Deletes the flag variable for the user
    [HttpGet]
    public IActionResult DeleteWaitFlag(int userId)
    {
        MatchmakingWait waitFlag = waitFlags.FirstOrDefault(w => w.UserId == userId);

        // Delete the flag variable
        waitFlags.Remove(waitFlag);

        return Json(new { success = true });
    }

    public int getMoney()
    {
        // Check is user is logged in
        if (HomeController.GetUserName(HttpContext) == null)
```

```
        {
            return 0;
        }
        else
        {
            // Get the user's coins
            var userName = HomeController.GetUserName(HttpContext);
            var user = _context.Users.FirstOrDefault(u => u.Username == userName);
            return user.Coins;
        }
    }
}
```

The code above is responsible with the whole logic behind the game session, asides from the actual simulation process, which will be described in detail in the next subsection. The main methods from this controller are presented below:

- **GameMenu()** - Is a simple method that returns the view which allows the user to select the desired mentality and search for a match.
- **JoinMatchmakingAsync(int mentality)** - Whenever a player presses the join match button, this function is called. It is responsible for deciding if a player has to be redirected to the waiting page, or if the player already has an opponent, then calls other methods to create the match and notify the other user that an opponent for him has been found.
- **GameRoom()** - Returns a page where a user is shown an animation while waiting for a match.
- **CheckMatchStatus(int userId)** - The method that a waiting client calls repetitively to check when a match has been found. When the user has been found an opponent, the flag returned by this function is set as true, and also a matchId is returned so that the user can be redirected to his match.
- **DeleteWaitFlag(int userId)** - Once the user gets redirected, his waiting flag must be deleted from the list of waiting players, so that when he will choose to play again, the flag will not ruin the matchmaking process.
- **CreateMatch(int user1, int user2, int mentality1, int mentality2)** - When two users have been grouped, this function creates their match and calls the simulation.
- **GameSession(string matchId)** - Returns a view and a list of events. Will be presented in more detail in the Simulation front-end section.

The results of this process can be seen in the print screens below:

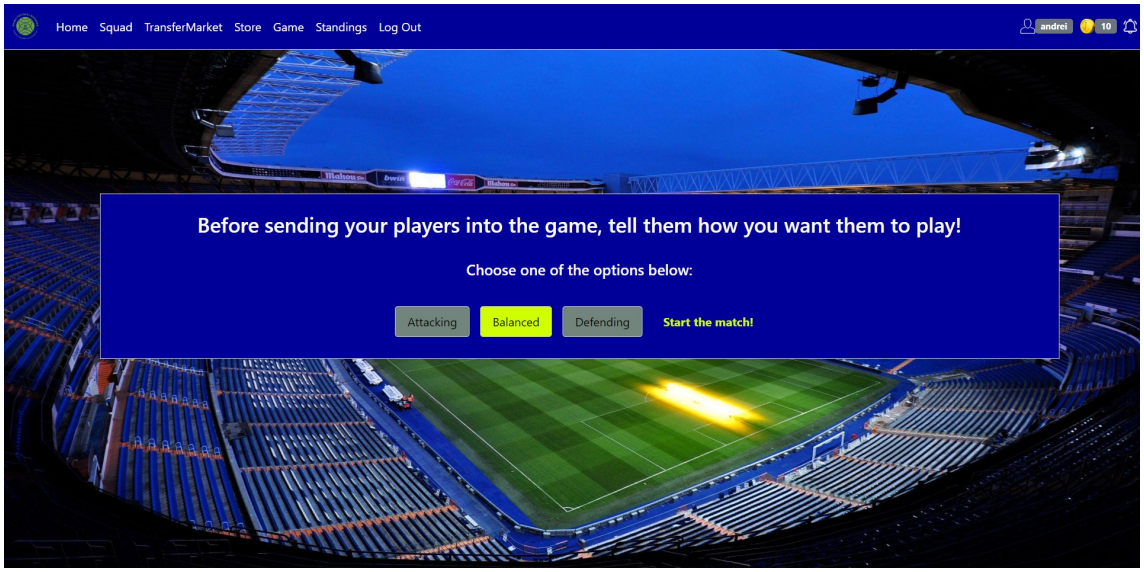

Figure 21: Game menu with join match and select mentality options

Figure 22: Animated waiting room

In the waiting room print screen, it can be observed that the console output shows the matchFound flag as being false, whenever it checks if it was changed with the CheckMatchStatus method from the controller. In the following code, you will find the java script code that is responsible for calling that method every half a second and also redirecting the user to the correct page whenever the match has been created.

Java Script part of GameRoom View

```
var userId = |''|@Model.UserId|''|; // Get the user id from the model
setInterval(function () {
    checkMatchStatus(userId); // Call the checkMatchStatus function with the userId parameter every
    ↪  0.5 seconds
}, 500);

function checkMatchStatus(userId) {
    // Send an AJAX request to the server to check the match status
    .ajax(−
        url: |''|@Url.Action("CheckMatchStatus", "Game")|'',
        type: ''|GET|''|,
        data: { userId: userId }, // Pass the userId as a parameter
        success: function (data) {
            console.log(|''|MatchFound:|''|, data.matchFound); // Log the MatchFound flag in the
            ↪  console
            if (data.matchFound) {
                var id = data.matchId; // Match Id

                // Deletes the wait flag for the user
                .ajax(−
                    url: |''|@Url.Action("DeleteWaitFlag", "Game")|'',
                    type: ''|GET|''|,
                    data: { userId: userId }, // Pass the userId as a parameter
                    success: function (data) {
                        console.log(|''|WaitFlagReset:|''|, data.waitFlagReset); // Log the
                        ↪  WaitFlagReset flag in the console
                    }
                });
                window.location.href = "/Game/GameSession?matchId=" + id; // Redirect to the
                ↪  simulation page with the ID parameter
            }
        }
    });
}
```

5.8.2 Simulation Back-end

When we first envisioned the app, we had the feeling that it was missing a very important aspect. Of course the opening of packs and team assembly can be a fun experience the first couple of times you go through the application, but as a real football manager, you need to have a way for your team to see how it would compete against other managers and their teams.
For this, I was tasked to create a simulation process that would ultimately decide a winner (or draw). Of course, we could have left every aspect of the simulation to chance, but we wanted to create a realistic experience for both users and to make them feel like the decisions and how they manage their teams has an impact on their performance, and ultimately, the leaderboard. To create something that we are proud to call a 'simulation'.
Of course, this came with its own set of challenges, some of which would be making intelligent use of the attributes available, providing a balanced experience, while also leaving room for hazard, and having an entertaining match simulation process that is fun and also a bit thrilling to watch.
Our controller makes use of an internal class which we called >Team Statistics. This class helps by storing imoportant information that is necessary for the simulation process.

Team Statistics internal class

```
//      Class that holds every important aspect of each team

        internal class TeamStatistics
        {
            public int TeamId { get; set; }

            public string TeamName { get; set; }

            public int Score;

            public int TeamAttack { get; set; }
            public int TeamControl { get; set; }
            public int TeamDefense { get; set; }
            public PlayersValue AttackersValue { get; set; }

            public PlayersValue MidfieldersValue { get; set; }

            public PlayersValue DefendersValue { get; set; }

            // public TeamContract Striker { get; set; }
            public int StrikerValue { get; set; }
            public int GoalkeeperValue { get; set; }
            public StatusPercents AttackPercents { get; set; }

            public StatusPercents ControlPercents { get; set; }
            public StatusPercents DefensePercents { get; set; }

            public TeamStatistics(int teamId)
            {
                TeamId = teamId;
                TeamName = string.Empty;
                AttackPercents = new StatusPercents();
                ControlPercents = new StatusPercents();
                DefensePercents = new StatusPercents();
                AttackersValue = new PlayersValue();
                DefendersValue = new PlayersValue();
                MidfieldersValue = new PlayersValue();
                GoalkeeperValue = 0;
                TeamAttack = 0;
                TeamControl = 0;
                TeamDefense = 0;
            }
        }
```

We also used two additional internal classes that fit our needs,

- **PlayersValue** which holds the total sum of all players inside the team for attack, midfield control or defense.
- **StatusPercents** which will have a value from 1 to 100 for each group of players (attackers, midfielders and defenders) which represents their impact, or how much their attributes matter towards the team's value. I will delve further into this shortly.

PlayersValue internal class

```
//     Class that holds the value of each player in a team
        internal class PlayersValue
        {
            public int Attack { get; set; }
            public int Control { get; set; }
            public int Defense { get; set; }

            public PlayersValue()
            {
                Attack = 0;
                Control = 0;
                Defense = 0;
            }
        }
```

StatusProcents internal class

```
         //      Class that indicated the procents of each type of player in a team
        internal class StatusPercents
        {
            public byte Attackers { get; set; }
            public byte Midfielders { get; set; }
            public byte Defenders { get; set; }

            public StatusPercents()
            {
                Attackers = 0;
                Midfielders = 0;
                Defenders = 0;
            }
        }
```

We also use two random generators for our chances, one corresponding to each team.

StatusProcents internal class

```
//      Each team has a specific random number generator. I felt like doing it this way.
        Random randomPlayer1 = new Random(Guid.NewGuid().GetHashCode());
        Random randomPlayer2 = new Random(Guid.NewGuid().GetHashCode());
```

Now the most important part, our simulation. The simulation is split into three parts: Generating the team data, simulating the match and lastly updating the standings.

Simulation method

```
public void Simulation(Match match)
        {

            //---------------------------------------Generating the data for the
↪   simulation----------------------------------------//

            TeamStatistics homeTeam = new TeamStatistics(match.HomeTeamId); //   create a new
↪   TeamStatistics object for the home team
            TeamStatistics awayTeam = new TeamStatistics(match.AwayTeamId); //   and another for
↪   the away team

            homeTeam.TeamName = _context.Squads.FirstOrDefault(m => m.SquadId ==
↪   match.HomeTeamId).SquadName; //  Get the name of the home team
```

```
            awayTeam.TeamName = _context.Squads.FirstOrDefault(m => m.SquadId ==
↪  match.AwayTeamId).SquadName; //  Get the name of the away team

            //      Get teams mentalities from match model
            byte mentalityHomeTeam = (byte)match.HomeMentality;
            byte mentalityAwayTeam = (byte)match.AwayMentality;

            //      Get the players from the database and split them into there respective roles
            GetPlayers(homeTeam);
            GetPlayers(awayTeam);

            //      Get the impact of each type of player in each team based on the mentality the
↪  team selected
            GetPercentage(homeTeam, mentalityHomeTeam);
            GetPercentage(awayTeam, mentalityAwayTeam);


            //      Calculate the team's attack, control and defense values based on the previous
↪  information
            GetTeamStatistics(homeTeam);
            GetTeamStatistics(awayTeam);

            //---------------------------------------Simulating the
↪  match-----------------------------------------------------//
            if (homeTeam.TeamId != 0)
            {
                // Generate a list of events. Each event is a string that describes what happened
↪  during the simulation
                for (int i = 1; i <= 18; i++)
                {
                    // Check the possession of the ball, the team with possession has a chance of
↪  scoring
                    if (Possession(homeTeam, awayTeam))
                    {
                        chooseEvent(homeTeam, awayTeam);
                    }
                    else
                    {
                        chooseEvent(awayTeam, homeTeam);
                    }

                }
            }

            //     Save the score of the match in the database
            match.HomeGoals = homeTeam.Score;
            match.AwayGoals = awayTeam.Score;

            // Serialize the list of strings to JSON
            string EventsJson = JsonConvert.SerializeObject(simulationEvents.ToArray());
            match.Events = EventsJson;

            // Save the changes to the database
            _context.SaveChanges();


            //---------------------------------------Updating the
↪  standings-------------------------------------------------//
            //Find the standing for the home team
            Standing homeTeamStanding = _context.Standings.FirstOrDefault(m => m.UserId ==
↪  match.HomeTeamId);
            Standing awayTeamStanding = _context.Standings.FirstOrDefault(m => m.UserId ==
↪  match.AwayTeamId);
```

```
64
65              //If the home team standing is null, create a new one
66              CreateStanding(homeTeamStanding, match);
67              CreateStanding(awayTeamStanding, match);
68
69              // Update the standings
70              UpdateStanding(homeTeamStanding, match.HomeGoals, match.AwayGoals);
71              UpdateStanding(awayTeamStanding, match.AwayGoals, match.HomeGoals);
72
73              // Save the changes to the database
74              _context.SaveChanges();
75          }
```

To generate data, we made a query on the database that holds the player's attacking score, midfield control and defending score and split them into categories, based on the position inside the squad. For goalkeepers and strikers we made an extra check to see if the players in their respective positions are in fact goalkeepers or strikers. This adds a layer of depth to the squad creation, as it rewards managers that pay attention to the player positions.

GetPlayers method

```
1   //      We make a database query that takes a lot of useful information about the players inside
    ↪  the team
2               //   like their attacking, defending and control attributes, as well as their position
    ↪  in the team, which
3               //   will indicate which role they will play.
4               private void GetPlayers(TeamStatistics team)
5               {
6                   var teamQuery = from player in _context.Players
7                                   join teamContract in _context.TeamContracts
8                                   on player.PlayerId equals teamContract.PlayerId
9                                   join squad in _context.Squads
10                                  on teamContract.SquadId equals squad.SquadId
11                                  where teamContract.SquadId == team.TeamId
12                                  select new
13                                  {
14                                      PlayerId = player.PlayerId,
15                                      SquadId = teamContract.SquadId,
16                                      ContractId = teamContract.ContractId,
17                                      Playerpos = player.Position,
18                                      Overall = player.OverallRank,
19                                      Attacking = player.Attacking,
20                                      MidfieldControl = player.MidfieldControl,
21                                      Defending = player.Defending,
22                                      Position = teamContract.Position
23                                  };
24
25                  //      We then make a generic list using this data
26                  var Players = teamQuery.ToList();
27
28                  //      We make 4 extra lists that split the players into 4 big catogories:
    ↪  Attackers, Midfielders and Defenders, based on
29                  //   their Position.
30                  var teamGoalkeeper = Players.Where(r => r.Position == 1);
31                  var teamDefenders = Players.Where(r => r.Position == 2  r.Position == 3  r.Position
    ↪  == 4  r.Position == 5).ToList();
32                  var teamMidfielders = Players.Where(r => r.Position == 6  r.Position == 7
    ↪  r.Position == 8).ToList();
33                  var teamAttackers = Players.Where(r => r.Position == 9  r.Position == 10
    ↪  r.Position == 11).ToList();
34                  var teamStriker = teamAttackers.Where(r => r.Overall == teamAttackers.Max(r =>
    ↪  r.Overall));
35
36                  team.AttackersValue.Attack = (int)teamAttackers.Sum(x => x.Attacking);
```

```
                team.AttackersValue.Control = (int)teamAttackers.Sum(x => x.MidfieldControl);
                team.AttackersValue.Defense = (int)teamAttackers.Sum(x => x.Defending);
                team.MidfieldersValue.Attack = (int)teamMidfielders.Sum(x => x.Attacking);
                team.MidfieldersValue.Control = (int)teamMidfielders.Sum(x => x.MidfieldControl);
                team.MidfieldersValue.Defense = (int)teamMidfielders.Sum(x => x.Defending);
                team.DefendersValue.Attack = (int)teamDefenders.Sum(x => x.Attacking);
                team.DefendersValue.Control = (int)teamDefenders.Sum(x => x.MidfieldControl);
                team.DefendersValue.Defense = (int)teamDefenders.Sum(x => x.Defending);
                team.StrikerValue = teamStriker.Sum(x => x.Playerpos.Contains("ST")
    ↪  x.Playerpos.Contains("CF")  x.Playerpos.Contains("LW")  x.Playerpos.Contains("RW") ? x.Overall
    ↪  : x.Overall / 10);
                team.GoalkeeperValue = teamGoalkeeper.Sum(x => x.Playerpos.Contains("GK") ?
    ↪  x.Overall : x.Overall / 10);
            }
```

Next, based on the mentality of the team selected previously in the matchmaking phase, I have calculated the impact of each player group. The logic on which we built this system is the following:

- **Balanced mentality (id = 0)** Is a standard way for teams to play, in which neither their attack or defense or control is favored and all players play into their strengths.
- **Attacking mentality (id = 1)** Boosts the team's offensive capabilities to make it more likely to score, but in the same time, weakening it's defense.
- **Defensive mentality (id = 2)** The exact opposite of the above, it boosts its defense, just so that the team may hold its advantage, but sacrifices offensive value.

How we achieved these was through the implementation of a percentage system. In general, each player contributes almost evenly in each field, if the team wants to be more aggressive, for example, the attackers will have more impact on the field, which will increase the offensive value, but weaken the defensive value, as defenders are partially neglected.

GetPercentage method

```
private void GetPercentage(TeamStatistics team, byte mentality)
            {
                //      All procents are assigned using a byte data type to save space. The control
    ↪  procents are not changed
                //   regardless of the mentality as for our application's current state there is no
    ↪  need.
                //   However, in future releases this may be a subject to change.

                const byte ControlPercentOfTheAttackers = 20;       // 20% of the attacker's
    ↪  control will be used for the team's total control
                const byte ControlPercentOfTheMidfielders = 60;     // 60% of the midfielder's
    ↪  control will be used for the team's total control
                const byte ControlPercentOfTheDefenders = 20;       // 20% of the defender's
    ↪  control will be used for the team's total control

                //      I am saving the values inside an array

                byte attackPercentOfTheAttackers;
                byte attackPercentOfTheMidfielders;
                byte attackPercentOfTheDefenders;

                byte defensePercentOfTheAttackers;
                byte defensePercentOfTheMidfielders;
                byte defensePercentOfTheDefenders;

                if (mentality == 1)   // attacking mentality
                {
                    attackPercentOfTheAttackers = 80;
                    attackPercentOfTheMidfielders = 15;
                    attackPercentOfTheDefenders = 5;

```

```
                    defensePercentOfTheAttackers = 40;
                    defensePercentOfTheMidfielders = 30;
                    defensePercentOfTheDefenders = 30;
                }
                else if (mentality == 2)   // defensive mentality
                {
                    attackPercentOfTheAttackers = 30;
                    attackPercentOfTheMidfielders = 30;
                    attackPercentOfTheDefenders = 30;

                    defensePercentOfTheAttackers = 5;
                    defensePercentOfTheMidfielders = 15;
                    defensePercentOfTheDefenders = 70;
                }
                else
                {
                    // Handle other cases or provide default values
                    attackPercentOfTheAttackers = 60;
                    attackPercentOfTheMidfielders = 30;
                    attackPercentOfTheDefenders = 20;

                    defensePercentOfTheAttackers = 20;
                    defensePercentOfTheMidfielders = 30;
                    defensePercentOfTheDefenders = 50;
                }

                team.AttackPercents.Attackers = attackPercentOfTheAttackers;
                team.AttackPercents.Midfielders = attackPercentOfTheMidfielders;
                team.AttackPercents.Defenders = attackPercentOfTheDefenders;
                team.ControlPercents.Attackers = ControlPercentOfTheAttackers;
                team.ControlPercents.Midfielders = ControlPercentOfTheMidfielders;
                team.ControlPercents.Defenders = ControlPercentOfTheDefenders;
                team.DefensePercents.Attackers = defensePercentOfTheAttackers;
                team.DefensePercents.Midfielders = defensePercentOfTheMidfielders;
                team.DefensePercents.Defenders = defensePercentOfTheDefenders;

            }
```

What is left is to compute the team's attributes using the logic we described earlier

GetPlayers method

```
private void GetTeamStatistics(TeamStatistics team)
            {
                team.TeamAttack = (team.AttackersValue.Attack * team.AttackPercents.Attackers +
                    team.MidfieldersValue.Attack * team.AttackPercents.Midfielders +
                    team.DefendersValue.Attack * team.AttackPercents.Defenders) / 100;

                team.TeamControl = (team.AttackersValue.Control * team.ControlPercents.Attackers +
                    team.MidfieldersValue.Control * team.ControlPercents.Midfielders +
                    team.DefendersValue.Control * team.ControlPercents.Defenders) / 100;

                team.TeamDefense = (team.AttackersValue.Defense * team.DefensePercents.Attackers +
                    team.MidfieldersValue.Defense * team.DefensePercents.Midfielders +
                    team.DefendersValue.Defense * team.DefensePercents.Defenders) / 100;
            }
```

For the simulation, we generate 18 events that will be further displayed in the front-end section. To determine which team is having an event generated, we first need to determine which team has possession of the ball. This makes use of the control status of both teams.

Possession method

```
private bool Possession(TeamStatistics team1, TeamStatistics team2)
            {
```

```
                if (team1 == null  team2 == null) return false;
                else if (team1.TeamControl + randomPlayer1.Next(1, 150) >
                            team2.TeamControl + randomPlayer2.Next(1, 150))
                {
                    return true;
                }
                else
                    return false;
            }
```

After a team has possession we will generate a random event, with a likeliness value. For a team to get a penalty, it is a very small chance rather than score a normal goal or be granted a free kick. Every time the team with possession fails to score a goal, there is a chance for the opposing team to have the opportunity for a counter attack. If the counterattack fails too, then a random string is displayed.

The simulation logic

```
private void chooseEvent(TeamStatistics homeTeam, TeamStatistics awayTeam)
            {
                int simulationValue;
                int scoreInitial = homeTeam.Score;
                int currentEvent = randomPlayer1.Next(1, 20);
                if (currentEvent <= 12)
                {
                    simulationValue = Goal(homeTeam, awayTeam);
                    if (simulationValue >= 0)
                    {
                        simulationEvents.Add(GoalScorer(SelectPlayers(homeTeam)) + " scored a goal for "
                                    + homeTeam.TeamName + "!");
                        homeTeam.Score++;
                    } else if (simulationValue >= -20)
                    {
                        simulationEvents.Add(GoalScorer(SelectPlayers(homeTeam)) + " barely missed the Goal!");
                    }
                    else
                        simulationEvents.Add(RandomEvent(homeTeam));
                }
                else if (currentEvent <= 18)
                {
                    simulationValue = FreeKick(homeTeam, awayTeam);
                    if (simulationValue >= 0)
                    {
                        simulationEvents.Add(GoalScorer(SelectPlayers(homeTeam)) + " scores exemplary from a free kick for "
                                    + homeTeam.TeamName + "!");
                        homeTeam.Score++;
                    } else if (simulationValue >= -20)
                    {
                        simulationEvents.Add(GoalScorer(SelectPlayers(homeTeam)) + "'s kick is miraculously saved by the goalkeeper!");
                    }
                    else
                        simulationEvents.Add(RandomEvent(homeTeam));
                }
                else
                {
                    simulationValue = Penalty(homeTeam, awayTeam);
                    if (simulationValue >= 0)
                    {
                        simulationEvents.Add("Penalty scored by " +
GoalScorer(SelectPlayers(homeTeam)) + " for "
```

```
                                    + homeTeam.TeamName + "!");
                        homeTeam.Score++;
                    } else if (simulationValue >= -20)
                    {
                        simulationEvents.Add(GoalScorer(SelectPlayers(homeTeam)) + "'s kick is off
↪ the crossbar!");
                    }
                    simulationEvents.Add(RandomEvent(homeTeam));
                }
                if (scoreInitial == homeTeam.Score)
                {
                    simulationValue = CounterAttack();
                    if (simulationValue >= 0)
                    {
                        simulationEvents.Add("Sensational goal scored by " +
↪ GoalScorer(SelectPlayers(homeTeam)) + " for "
                                    + awayTeam.TeamName + " on a counter attack!");
                        awayTeam.Score++;
                    }

                }
            }
            private int Goal(TeamStatistics team1, TeamStatistics team2)
            {
                int team1Luck = randomPlayer1.Next(10, 150);
                int team2Luck = randomPlayer2.Next(10, 650);
                return (team1.TeamAttack + team1Luck) -
                     (team2.TeamDefense + team2Luck + team2.GoalkeeperValue);
            }

            private int Penalty(TeamStatistics team1, TeamStatistics team2)
            {
                int team1Luck = randomPlayer1.Next(1, 141);
                int team2Luck = randomPlayer2.Next(1, 101);
                return (team1.StrikerValue + team1Luck) -
                     (team2.GoalkeeperValue + team2Luck);
            }

            private int FreeKick(TeamStatistics team1, TeamStatistics team2)
            {
                int team1Luck = randomPlayer1.Next(1, 51);
                int team2Luck = randomPlayer2.Next(1, 101);
                return (team1.StrikerValue + team1Luck) -
                     (team2.GoalkeeperValue + team2Luck);
            }
            private int CounterAttack()
            {
                return (randomPlayer2.Next(1, 100) - 90);
            }

            private string RandomEvent(TeamStatistics homeTeam)
            {
                int randomEvent = randomPlayer1.Next(1, 8);
                switch (randomEvent)
                {
                    case 1:
                        return homeTeam.TeamName + " has possession of the ball!";
                    case 2:
                        return "Dangerous attack for " + homeTeam.TeamName + "!";
                    case 3:
                        return homeTeam.TeamName + " controls the ball!";
                    case 4:
                        return "Crossing opportunity for " + homeTeam.TeamName + "!";
```

```
                    case 5:
                        return "The referee catches" + GoalScorer(SelectPlayers(homeTeam)) + "in
↪  offsite";
                    case 6:
                        return "Corner kick for " + homeTeam.TeamName + "!";
                    case 7:
                        return GoalScorer(SelectPlayers(homeTeam)) + " has an amazing tackle!";
                    default:
                        return homeTeam.TeamName;
                }
            }
```

When a goal does get scored, we need to know who the striker is. For this, we have created the **GoalScorer()** method, which picks a player from the team, with a higher chance for a striker to get picked. After this, we update the team's score and add the result string to our simulation result list.

GoalScorer method

```
//      Function to determine goalscorer
            private string GoalScorer(List<Team> team)
            {
                //      The process will select a random number between 1 and 10 that will
↪  associate with the scorer
                //   1-6 means that the scorer is an attacker; 7-9 means the scorer is a midfielder
                //   and if the random score is exactly 10, then the scorer is a defender
                Random goalScorerRandom = new Random(Guid.NewGuid().GetHashCode());

                int goalScorerCurrentRandom = goalScorerRandom.Next(0, 11);


                //      Condition for an attacker to score a goal
                if (goalScorerCurrentRandom < 7)
                {
                    //     ! DISCLAIMER ! this implementation is very unethical and we would like
↪  to
                    //   change this as soon as possible, but for that is required an entire revamp
                    //   of the database, so bear with us for now.

                    //     This random gives even chance for any of the 3 attackers to score a goal
                    //   followed by a query through the models to extract the player's firstname
↪  and lastname
                    goalScorerCurrentRandom = goalScorerRandom.Next(9, 12);

                    //     Going through each team in the list of teams
                    foreach (Team echipa in team)
                    {
                        //      Going through each player in the current team
                        foreach (Player player in echipa.Players)
                        {
                            foreach (TeamContract contract in echipa.Contracts)
                            {
                                //      Checking to see if the current player has a contract with
↪  the current
                                //      team to get the player at the position our random number
↪  has generated
                                if (contract.Position == goalScorerCurrentRandom)
                                {
                                    // Increment the player's goals in Scorers table
                                    //player.Goals ++;
                                    return player.FirstName + " " + player.LastName;
                                }
                            }
                        }
                    }
```

```
                }

                //      Condition for a midfielder to score a goal
                else if (goalScorerCurrentRandom < 10)
                {
                    goalScorerCurrentRandom = goalScorerRandom.Next(6, 9);
                    foreach (Team echipa in team)
                    {
                        foreach (Player player in echipa.Players)
                        {
                            foreach (TeamContract contract in echipa.Contracts)
                            {
                                // Get player at position random from contracts
                                if (contract.Position == goalScorerCurrentRandom)
                                {
                                    return player.FirstName + " " + player.LastName;
                                }
                            }
                        }
                    }
                }
                else
                {
                    //      Condition for a midfielder to score a goal
                    goalScorerCurrentRandom = goalScorerRandom.Next(2, 6);
                    // Get the player on position random
                    foreach (Team echipa in team)
                    {
                        foreach (Player player in echipa.Players)
                        {
                            foreach (TeamContract contract in echipa.Contracts)
                            {
                                // Get player at position random from contracts
                                if (contract.Position == goalScorerCurrentRandom)
                                {
                                    return player.FirstName + " " + player.LastName;
                                }
                            }
                        }
                    }
                }

                //      In case we something goes wrong, we return an Error message
                return "Error";
            }
```

When creating our standings we first need to assign a new standing to each player, if this is their first game. This is done through an initialization process in the **CreateStanding()** method. and saving the entry in the database.

CreateStanding method

```
private void CreateStanding(Standing standing, Match match)
        {
            //If the home team standing is null, create a new one
            if (standing == null)
            {
                Standing newStanding = new Standing();
                newStanding.StandingsId = match.HomeTeamId;
                newStanding.Position = _context.Standings.Count() + 1;
                newStanding.UserId = match.HomeTeamId;
                newStanding.MatchesPlayed = 0;
                newStanding.Trophies = 0;
                newStanding.MatchesWon = 0;
```

```
                newStanding.MatchesLost = 0;
                newStanding.MatchesDrawn = 0;

                _context.Standings.Add(newStanding);

                // Save the changes to the database
                _context.SaveChanges();

                standing = newStanding;
            }
        }
```

Lastly, we update each team's Standing values, based on the performance during the match. The trophies allow for a quick sorting in the Standings tab and each manager is granted a coin value after the match is over.

UpdateStanding method

```
private void UpdateStanding(Standing standing, int homeScore, int awayScore)
        {
            standing.MatchesPlayed++;
            standing.GoalsFor += homeScore;
            standing.GoalsAgainst += awayScore;
            if (homeScore > awayScore)
            {
                standing.MatchesWon++;
                standing.Trophies += 10;
                _context.Users.FirstOrDefault(m => m.UserId == standing.UserId).Coins += 100;
            }
            else if (homeScore == awayScore)
            {
                standing.MatchesDrawn++;
                _context.Users.FirstOrDefault(m => m.UserId == standing.UserId).Coins += 50;
            }
            else
            {
                standing.MatchesLost++;
                standing.Trophies -= 3;
                _context.Users.FirstOrDefault(m => m.UserId == standing.UserId).Coins += 10;
            }
        }
```

5.8.3 Simulation Front-end

By the time this part of the game process is accessed, the simulation is already finished. As already mentioned in the Matchmaking subsection, the controller GameSession method returns a view and a list of events. This list of events is processed by a java script code that can be seen below and events are being displayed one by one at a fixed time interval. The same script is also responsible for making decisions about what to display, including: when to increment the score and for which team, when was a goal scored and an animated message must be displayed and also which events belong to which team and displays them in the correct box.

Javascript part of GameSession View

```
document.addEventListener('DOMContentLoaded', () => {
    // Select the event column
    const eventColumn_team1 = document.querySelector('#events-team-1');
    const eventColumn_team2 = document.querySelector('#events-team-2');

    // Function to add a new event to the specific column
    function addEvent(eventText) {
        // Create a new event element
        const eventElement = document.createElement('div');
        const $team1 = $('.team#team1');
```

```
        const $team2 = $('.team#team2');
        const $team1Score = $('.score:eq(0)');
        const $team2Score = $('.score:eq(1)');

        eventElement.classList.add('event');

        // Checks which team has the event
        if (eventText.includes($team1.text())) {

            // Logs a message to the console
            console.log("Event for home team!");

            // Add the goal class to the event element if the event is a goal
            if (eventText.includes("goal")  eventText.includes('Penalty')  eventText.includes('free
            ↪  kick')  eventText.includes('scores')) {
                eventElement.classList.add('goal', 'flash-goal'); // make the event goal type
                $team1Score.text(parseInt($team1Score.text()) + 1); // increment the score
            }

            eventElement.textContent = eventText;

            // Add the event element to the column
            eventColumn_team1.appendChild(eventElement);

            // Add the same event but invisible to the other team
            const invisibleEventElement = eventElement.cloneNode(true);
            invisibleEventElement.style.visibility = 'hidden';
            eventColumn_team2.appendChild(invisibleEventElement);
        }
        else {
            if (eventText.includes($team2.text())) {

                // Logs a message to the console
                console.log("Event for away team");

                // Add the goal class to the event element if the event is a goal
                if (eventText.includes("goal")  eventText.includes('Penalty')
                ↪  eventText.includes('free kick')  eventText.includes('scores')) {
                    eventElement.classList.add('goal', 'flash-goal'); // make the event goal type
                    $team2Score.text(parseInt($team2Score.text()) + 1); // increment the score
                }
                eventElement.textContent = eventText;

                // Add the event element to the column
                eventColumn_team2.appendChild(eventElement);

                // Add the same event but invisible to the other team
                const invisibleEventElement = eventElement.cloneNode(true);
                invisibleEventElement.style.visibility = 'hidden';
                eventColumn_team1.appendChild(invisibleEventElement);

            }
        }

        // Scroll to the bottom of the columns
        eventColumn_team1.scrollTop = eventColumn_team1.scrollHeight;
        eventColumn_team2.scrollTop = eventColumn_team2.scrollHeight;

        // Repeat the flash animation every 0.2 seconds within the 2-second duration
        if (eventElement.classList.contains('goal')) {
            const duration = 2000;
            const interval = 200;
            let elapsed = 0;
```

```
            const flashInterval = setInterval(() => {
                if (elapsed >= duration) {
                    clearInterval(flashInterval);
                    eventElement.classList.remove('flash-goal');
                } else {
                    eventElement.classList.toggle('flash-goal');
                    elapsed += interval;
                }
            }, interval);
        }
    }

    // Increments the score of the given team
    function scoreGoal(team) {
        score += 1;
        team.textContent = score;
    }

    // Increments the minute
    function incrementMinute() {
        if(minute < 90){
            minute += 1;
            minuteElement.textContent = `${minute}'`;
        }
        else{
            minuteElement.textContent = `Full Time`;
        }
    }

    //addEvent function with a delay of 5 seconds for each event
    const events = @Html.Raw(Json.Serialize(ViewBag.Events));
    if (events && Array.isArray(events)) {
        let delay = 0;
        for (const event of events) {
            setTimeout(() => {
                addEvent(event);
            }, delay);
            delay += 5000;
        }
    } else {
        console.error('ViewBag.Events is not an array or is undefined/null');
    }

    //call incrementMinute function every 1.1 seconds
    const minuteElement = document.querySelector('#minute');
    let minute = 0;
    setInterval(() => {
        incrementMinute();
    }, 1100);
});
```

6 Application testing

For testing we have continuously tested the application as we developed. The real challenges for testing appeared only when we included the online features for the application (see Transfer market and Game). Many of the bugs that we encountered have been fixed easily, on the spot, but there were instances where it required a deeper dive and analysis of the code. So far, the application is running optimally, but if we want to extend the application further, we will require a revamp of the database. For more information about the tests, see the Test Cases file.

7 Future Development

We appreciated working together on this project and would love to keep developing the application. Some of the features we are looking forward to see in action are:

- Running a real real-time simulation with the use of SignalR and web-sockets; The simulation should allow the user to make changes to the team, like replacing players and changing the mentality in the middle of the simulation.
- A more complex and entertaining simulation experience, including yellow and red cards, player fatigue.
- A chat system for players in the lobby, to find games more easily.
- An inspect feature for the players in the squad in which you can see the full description and attributes of each player together with an image;
- A transfer market that implements auction features, like bidding and buyout prices;
- A more visually appealing user-interface, which updates in real-time and shows the strengths and weaknesses of your squad;
- A training feature that lets the player play against a self generated team to test their team's capabilities;
- Small features like password reset, profile customization, animations;

8 Conclusions

In conclusion, this documentation has provided an in-depth overview of our football manager application and highlighted its key features and functionalities. We have explored the various aspects of the application, including team management, player transfers and match simulations. Through rigorous testing and continuous improvement, we have strived to create a robust and user-friendly platform that meets the needs of football enthusiasts and aspiring managers alike. We believe that our football manager application will empower users to experience the thrill of running their own virtual football team, making strategic decisions, and competing for glory. We are committed to ongoing development and support, ensuring that our application remains up-to-date, reliable, and enjoyable for all users. Thank you for choosing Sigma Soccer application, and we look forward to your success on the virtual pitch.