Razvoj multiplatformske aplikacije u tehnologiji .NET MAUI

Tkalčec, Antun

UNIVERSITY OF ZAGREB

FACULTY OF ORGANIZATION AND INFORMATICS

V A R A Ž D I N

Antun Tkalčec

DEVELOPMENT OF A MULTIPLATFORM APPLICATION IN .NET MAUI TECHNOLOGY

MASTER'S THESIS

Varaždin, 2023.

UNIVERSITY OF ZAGREB

FACULTY OF ORGANIZATION AND INFORMATICS

V A R A Ž D I N

Antun Tkalčec

JMBAG: 0016136241

Study: Databases and knowledge bases

DEVELOPMENT OF A MULTIPLATFORM APPLICATION IN .NET MAUI TECHNOLOGY

MASTER'S THESIS

Mentor:

Izv. Prof. dr. sc. Mario Konecki

Varaždin, May 2023.

Antun Tkalčec

Statement of originality

I declare my master's thesis as the result of my own work and that, writing it, I did not use sources that are not specified in it. Ethically suitable and acceptable methods and work techniques were used in the creation of this thesis.

Confirmed by the author by accepting the FOI-radovi provisions

_____

Summary

The subject of this paper is the development of a multiplatform application in Microsoft's new technology, .NET MAUI, with a backend built using ASP.NET Core, all with technologies that developers commonly use to build full-fledged production applications.

The paper first lists and describes tools and technologies that are used in software development, before taking the reader through the steps of designing a database model, creating an application backend using ASP.NET Core, designing user interface ideas in Figma and finally creating a .NET MAUI application.

This paper will demonstrate the capabilities of Microsoft's new technology, .NET MAUI (Multiplatform Application User Interface) by displaying the particularities of an application built primarily for Android smartphones, with a backend built using ASP.NET and the Entity Framework Code-First approach. The paper may serve as a guide to those readers who are interested in multiplatform application development, Microsoft technologies or both.

The paper concludes that .NET MAUI may not be the best choice for multiplatform application development at this time, due to numerous bugs and an unsatisfactory frequency of updates and bugfixes.

Key words: .NET; software development; multiplatform application; ASP.NET; .NET MAUI; C#; XAML;

Content

1. Introduction

When developing an application, a lot of time is usually spent adapting the business logic and the user interface (UI) to each platform – such as Android, iOS, Windows, Web and macOS. In situations where developers do not have the time to use specific frameworks or technologies to build the same application for each platform, technologies like Microsoft's new .NET MAUI can be useful.

.NET MAUI stands for .NET Multiplatform Application User Interface, and it allows the developer to seamlessly target devices of most platforms. MAUI abstracts the latest technologies for building native apps on popular platforms into one common framework. This means that developers can build applications that „look and feel like the native platforms“ from a single codebase [1].

This paper will display the power of .NET MAUI by taking the reader through the necessary steps of building a multiplatform application and its backend, starting by designing the database model, building a database using Entity Framework's Code-First approach, creating an API using ASP.NET Core and finally building the user interface using .NET MAUI and XAML. The paper will also show some difficulties, bugs and weaknesses of MAUI.

The first part of this paper will present the technologies that will be used in the creation of the multiplatform application and its backend APIs, relying mostly on cited sources. In the main part of the paper, some techniques that will be shown are taken from accrued experience of the author. This paper assumes the reader has some experience and knowledge of C#, XAML, REST APIs and object-oriented programming in general, as well as terms such as inheritance.

The author has chosen .NET MAUI for his Master's thesis because technologies like these are very valuable in today's job market, as well as because of a general interest in Microsoft development technologies.

2. Tools and technologies

This chapter will list and describe tools and technologies that will be used during development of the multiplatform application.

2.1. Microsoft Visual Studio

All of the code for the application's frontend and backend will be written inside of a technology called Microsoft Visual Studio. „The Visual Studio IDE is a creative launching pad that you can use to edit, debug, and build code, and then publish an app.“ [2]
Microsoft offers Visual Studio in several versions:

1. Community – the free version, which will be used for development throughout this paper (specifically VS Community 2022)
2. Professional – paid, for developers working on commercial projects, adding more advanced capabilities
3. Enterprise – paid, for large-scale enterprise development teams and complex projects

Image 1. Microsoft Visual Studio logo, Source: https://commons.wikimedia.org/wiki/File:Visual_Studio_Icon_2022.svg

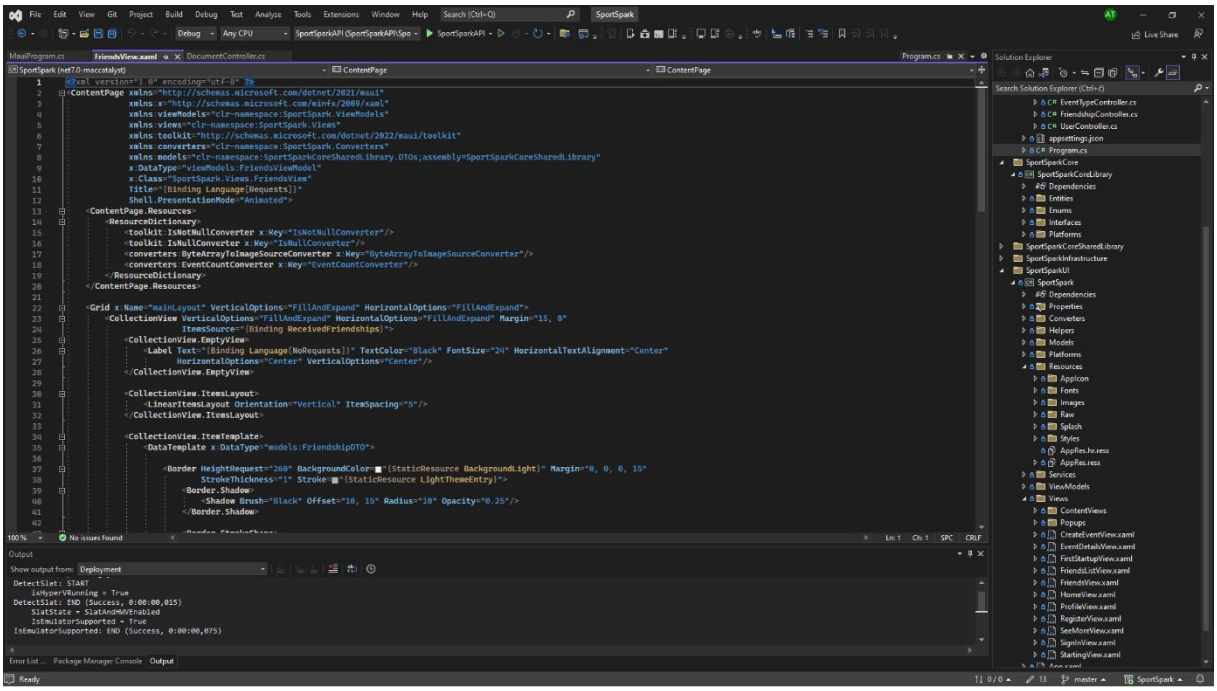


Image 2. Microsoft Visual Studio's User Interface with tabs open, Source: author screenshot

2.2. Microsoft SQL Server

The database for the application will be setup locally, meaning a local database server is required. Microsoft's SQL Server will be used for this purpose. Microsoft SQL Server is a relational database management system, or RDBMS for short, and is a powerful and widely used database platform. It provides an environment for managing and storing data.

Image 3. Microsoft SQL Server logo, Source: https://www.commvault.com/supported-technologies/microsoft/sql

2.3. Microsoft SQL Server Management Studio (SSMS)

Microsoft's SQL Server is only an environment for storing data. To work with the data, another of Microsoft's technologies will be used. SQL Server Management Studio, or SSMS for short, „is an integrated environment for managing any SQL infrastructure...“ and „provides tools to configure, monitor, and administer instances of SQL Server and databases [3]. It is available in a multitude of languages, such as Chinese, English, French, German, Italian, Japanese and others. SSMS can be used to query and manage databases locally and in the cloud, but the cloud is out of scope for this paper. Everything to do with this application will be done locally.

SSMS will be used during the development of the multiplatform application in multiple ways. For instance, the database model diagram will be created in SSMS and shown in a future chapter.

Image 4. Microsoft SQL Server Management Studio logo, Source: https://stackshare.io/microsoft-sql-server-management-studio

2.4. Git

In order to store code in a remote location, which allows developers to collaborate on projects, a version control system such as Git is used. Though the application will be created by one developer on a single machine, Git will still be used to *commit* and *push* code changes to a GitHub repository. All of the code can be found by clicking the GitHub link at the end of this paper.

Using Git, developers can simultaneously develop features or fix bugs by using branches. One of the key features is its ability to track and manage changes by detecting differences between versions of code. Ultimately, Git is a core tool for any developer, and the reader is hereby encouraged to read more about this industry standard technology [4].

Image 5. Git logo, Source: https://commons.wikimedia.org/wiki/File:Git-logo.svg

2.5. Sourcetree

Git alone does not have a *nice-to-look-at* graphical user interface, but is rather (mostly) a command line tool. Sourcetree is a free Git client and a graphical user interface for Git that allows developers to manage and visualize repositories and code changes [5].

Image 6. Sourcetree logo, Source: https://iconduck.com/icons/94916/sourcetree

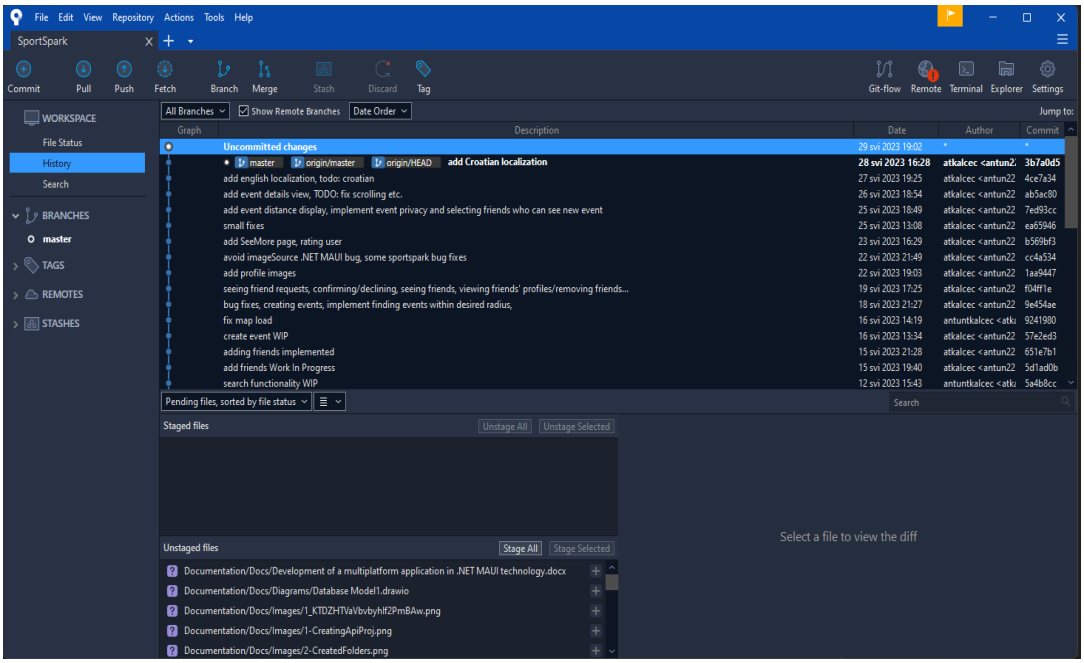


Image 7. Sourcetree User Interface, Source: author screenshot

Image 7. shows Sourcetree's user interface, where the multiplatform application's commits and branches can be seen.

2.6. adb

adb or Android Debug Bridge „is a versatile command-line tool that lets you communicate with a device.“ [6] Given that the multiplatform application will be a primarily Android mobile application, adb is used to connect the phone to the localhost API. Put simply, starting the API project within Visual Studio runs the API locally, but the phone cannot send HTTP requests to it without using a tool such as adb. The API running on localhost will have a port, which needs to be used in the following command-line command:

adb reverse tcp:port tcp:port

Once the Android phone is connected via USB cable to the PC the API is running on and this command is used, the phone will successfully send HTTP requests to the locally running API. This means the API will send back JSON files containing data that is displayed on the application's frontend.

2.7. Backend tools and technologies

An application's backend serves as a bridge between the frontend, or user interface, which is what the user can see, and the database where all the data is stored. Many technologies, frameworks and programming languages may be used to create backends. This paper will focus on C# and ASP.NET, along with Entity Framework Core, both of which will now be described.

2.7.1. ASP.NET Core

As per Microsoft documentation, „ASP.NET Core is a cross-platform, high-performance, open-source framework for building modern, cloud-enabled, Internet-connected apps.“ [7] It is suitable for a wide range of applications, from small websites to large enterprise systems. Furthermore, it is the successor to the older ASP.NET framework, and provides the following benefits and more [7]:

1. Open-source and community-focused
2. Built-in **dependency injection**
3. A lightweight, high-performance and modular HTTP request pipeline
4. Simplifies modern web development

ASP.NET Core offers features to build web APIs and web apps, along with technologies such as Razor Pages and Blazor and patterns such as MVC (Model-View-Controller). However, this paper will only focus on building an API for the application using ASP.NET Core.

Image 8. ASP.NET Core logo, Source: https://www.azureblue.io/tag/asp-net-core/

2.7.2. Entity Framework Core 7

As per Microsoft documentation, „Entity Framework (EF) Core is a lightweight, extensible, open source and cross-platform version of the popular Entity Framework data access technology.“ It is an ORM (object-relational mapper), which means developers can

work with a database using .NET objects and mostly eliminates the need to write SQL code to retrieve rows from a database table or tables. However, it cannot create *stored procedures,* one of which will later be required to retrieve events based on location and a user's chosen radius. That *stored procedure* will be created using SQL, and it will be called using a mix of Entity Framework Core 7 and SQL. If the reader wishes to view the *stored procedure's* creation and implementation, they are hereby encouraged to check the application's code on GitHub.

EF Core 7 is a NuGet package, which means it is installed to a project inside of Visual Studio using the NuGet Package Manager or the .NET Core CLI.

Image 9. Entity Framework Core logo, Source: https://codeopinion.com/porting-to-entity-framework-core/

EF Core supports two development approaches:

1. **Code-First**
2. Database-First

The application's backend will be using the Code-First approach. An integral part of EF is the **DbContext** class, which „represents a session with the database and can be used to query and save instances of entities to a database.“ [9] The next building block of EF Core are **entities**. The code below displays the *Event* entity, and the *DbContext* class which references it.

```
[Table("Event")]
public class Event : BaseEntity
{
    [Required]
    [StringLength(50)]
    public string Title { get; set; }
    [Required]
```

```
    [StringLength(150)]
    public string Description { get; set; }
    ...
    #endregion
}
public class SportSparkDBContext : DbContext
{
...
    public DbSet<Event> Events { get; set; }
...
}
```

As a reminder, the complete code of these classes and more may be found using the GitHub link at the end of this paper. As one can see, the class shown inherits from *DbContext*, and one of its properties is *DbSet<Event> Events*. The *Event* entity has an annotation, `[Table(„Event“)]` which, along with the aforementioned property, lets EF know that it must create a database table named „Event“ with properties defined inside the *Event* class. The way to create this database table, and others, is using so called migrations, which will be covered in a later chapter. Database creation will also be covered in more detail in a later chapter.

2.8. Frontend tools and technologies

The frontend encompasses the development of anything the user can see, or what happens inside of the user's device. Before coding XAML code to create a user interface inside .NET MAUI, mockups will be created using Figma. XAML and .NET MAUI itself will also be described in the following subchapters.

2.8.1. Figma

Figma is a modern interface design tool to create user interfaces for all sorts of applications, ranging from web to desktop to mobile. It was first released in 2016, and offers seamless collaboration between designers. [10]

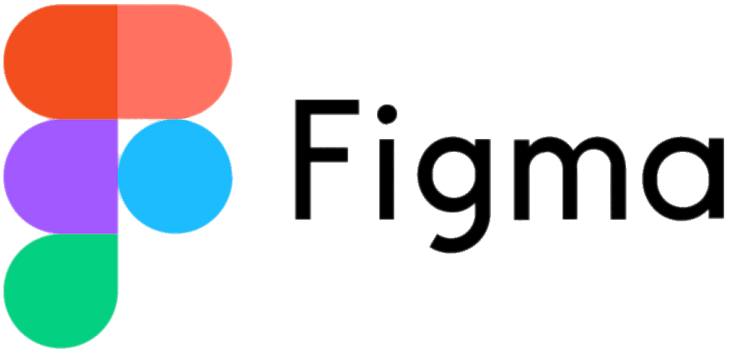


Image 10. Figma logo, Source: https://www.stickpng.com/img/icons-logos-emojis/tech-companies/figma-logo

For this application's development, Figma will merely be used to *brainstorm* UI design ideas. Figma offers many more features and tools, but they are out of scope for this paper. The Figma-created mockups will be shown in a later chapter.

2.8.2. XAML

XAML stands for Extensible Application Markup Language, and is a „declarative language that's based on XML.“ [11] It allows developers to build rich and interactive applications while separating the UI design from the application logic. XAML is quite similar to HTML or XML, while the underlying functionality and behavior are implemented in the code-behind using C#. XAML also supports data binding, which allows UI elements to be connected to data sources, enabling automatic updates and synchronization. This will be crucial to this application's development, and is one of the foundations of the Model-View-ViewModel-Service pattern which will be discussed later. Developers may create interfaces using tags and attributes, instead of writing code to create and position UI elements. These tags and attributes may be very simple, but designing a complex interface quickly becomes quite complicated. The following code is an example of a, relatively to other views, simple view inside of the multiplatform application's final design.

```
<?xml version="1.0" encoding="utf-8" ?>

<ContentPage xmlns="http://schemas.microsoft.com/dotnet/2021/maui"

             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"

             xmlns:skia="clr-
namespace:SkiaSharp.Extended.UI.Controls;assembly=SkiaSharp.Extended.UI"

             xmlns:viewModels="clr-namespace:SportSpark.ViewModels"
```

```
            x:Class="SportSpark.Views.StartingView"
            Shell.NavBarIsVisible="False"
            x:DataType="viewModels:StartingViewModel">
    <Grid RowDefinitions="1.5*, *, 0.5*, *" ColumnDefinitions="*, *" Margin="0, 30, 0, 0" RowSpacing="30">
        <Border BackgroundColor="{StaticResource SportSparkLightGreen}"
                HorizontalOptions="CenterAndExpand" StrokeThickness="0" Margin="15, 0" Grid.RowSpan="1" Grid.Row="0"
                Grid.ColumnSpan="2">
            <Border.StrokeShape>
                <RoundRectangle CornerRadius="30"/>
            </Border.StrokeShape>
            <skia:SKLottieView Source="manonphone.json" IsAnimationEnabled="True" RepeatCount="-1" Padding="20"/>
        </Border>
        <Label Text="{Binding Language[Discover]}" FontSize="32" Grid.Row="1" TextColor="Black"
               HorizontalOptions="CenterAndExpand" HorizontalTextAlignment="Center" Grid.ColumnSpan="2" FontAttributes="Bold"
               Shadow="{StaticResource DefaultShadow}" Margin="20"/>
        <Label Text="{Binding Language[Explore]}" FontSize="15" Grid.Row="2"
               Grid.ColumnSpan="2" TextColor="Black" HorizontalOptions="CenterAndExpand" HorizontalTextAlignment="Center"
               Shadow="{StaticResource DefaultShadow}" Margin="25, 5"/>
        <Button Text="{Binding Language[SignIn]}" FontAttributes="Bold" FontSize="28" BackgroundColor="{StaticResource SportSparkDarkBlue}"
                Grid.Row="3" HeightRequest="70" Margin="15, 0" TextColor="White" Shadow="{StaticResource DefaultShadow}"
                CornerRadius="30" Grid.ColumnSpan="2" Command="{Binding SignInCommand}"/>
    </Grid>
</ContentPage>
```

The code displays tags such as *Button*, and attributes such as *CornerRadius*. Furthermore, it displays the aforementioned data binding, as well as defining namespaces that can be used inside of the XAML page. Other views inside of the multiplatform application will reach over 200 lines of XAML code.

2.8.3. .NET MAUI

As per Microsoft documentation, „.NET Multi-platform App UI (.NET MAUI) is a cross-platform framework for creating native mobile and desktop apps with C# and XAML.“ [12] .NET MAUI is open-source and developers may use it to create apps that run on mobile (Android, iOS), macOS and Windows from a single shared code-base. It is the evolution of Xamarin.Forms, and entered *General Availability* in 2022.

MAUI provides a framework for building the UIs for mobile and desktop apps, unifying „Android, iOS, macOS and Windows APIs into a single API that allows a write-once run-anywhere developer experience.“ [12] As building apps for iOS and macOS requires a Mac computer, this paper will focus on the Android side of .NET MAUI, while displaying the UI difference between Android and Windows at the end of this paper.

In .NET MAUI, the UI is built using a collection of controls that are used to display data, initiate actions, indicate activity and so on. It also provides [12]:

1. Multiple page types
2. Data binding
3. Handler customization
4. APIs for accessing native device features, such as GPS, accelerometer, battery and network...
5. A single project system (different from Xamarin.Forms, MAUI's predecessor)
6. Hot reload, allowing XAML modification while the app is running, which means the developer may make UI changes and tweaks without restarting the application

This paper will not demonstrate all of the capabilities of .NET MAUI, but the ones required to create this particular application.

3. Application idea

The very first thing one needs before developing an application is an idea. The application that will be built throughout this paper is one that serves as a tool to find sports events in the user's vicinity. The idea is, therefore, to give users a quick and easy way to find sports events they might be interested in inside of a certain square kilometer radius. The user starts by creating an account and signing into the application. Furthermore, they may choose to complete their profile with a picture or some information about themselves. On the same profile page, they may create new events. The most important part of this setup process will be specifying a radius inside which they want to see other users' sports events.

Another way to see sports events, other than seeing the nearest one, is to add „friends“. If a user's „friend“ creates an event, the user may visit the friend's profile and view their events, even if those events are outside of the radius in which they wish to see them. Users create events by specifying the following data:

- Event location – where the event will take place, in the form of latitude & longitude
- Event privacy – who can see the event; an event may be public, private or visible to selected friends
- Event repetition type – does the event takes place only once, daily, weekly...
- Event duration – how long the event will last, in hours
- Event price – an event may have an entry fee
- Event time – when the event takes place (time of day)
- Number of participants – how many people the event is meant for
- Event type – of which sport the event is

To combat the creating of fake events, a rating system will be put in place. Users can rate event organizers, and a bad rating may deter potentially interested users from signing up to their events. This system would be especially useful for repeating events. A *gamification* of sorts could be implemented. Users can check an event creator's profile, thus seeing their rating and deciding if commiting their time to that creator's event is worthwhile.

To monetize an app like this, there would be a free version with ads and a paid version with no ads. Users would have a choice to *bump* their events (increase their visibility for a short time) by paying a small fee. However, monetization will not be implemented into the application as part of this paper.

The application might also target video game events. To achieve this feature, there would not be a location requirement, because these events would be played online. For

example, an event that specifies a *roleplay* match of a certain war-simulation video game, where players would go out of their way to play the game in a way that would simulate real wartime tactics and combat. Usually, these types of events are organized by way of online communities like *YouTube communities* or on platforms like *Discord*. This feature would simplify the process of organizing these events.

Marketing a primarily mobile app like this can be done in multiple ways. For free, the app could be marketed through posts on platforms such as *Facebook* or *Discord* groups, *Twitter* and other social networks. Another way is to make deals with sport organizations or clubs, where they would use the app to organize events and therefore, with time, have a higher number of participants in their events.

Ultimately, not all of these ideas will be implemented in the final application. The purpose of this paper is to demonstrate the aforementioned technologies, not to create a real-world consumer-facing mobile application.

Having all this information in mind, **SportSpark** will be the application's name.

4. Best practices, patterns and important terms

This chapter will describe some of the best practices, patterns and important terms that all developers aiming to create an application such as this should be aware of.

4.1. Clean Architecture

As per Microsoft documentation, Clean Architecture has gone by multiple names over the years. Hexagonal Architecture, Ports-and-Adapters and Onion Architecture are some of the older names, and they all describe putting the business logic and application model at the center of the application [13].

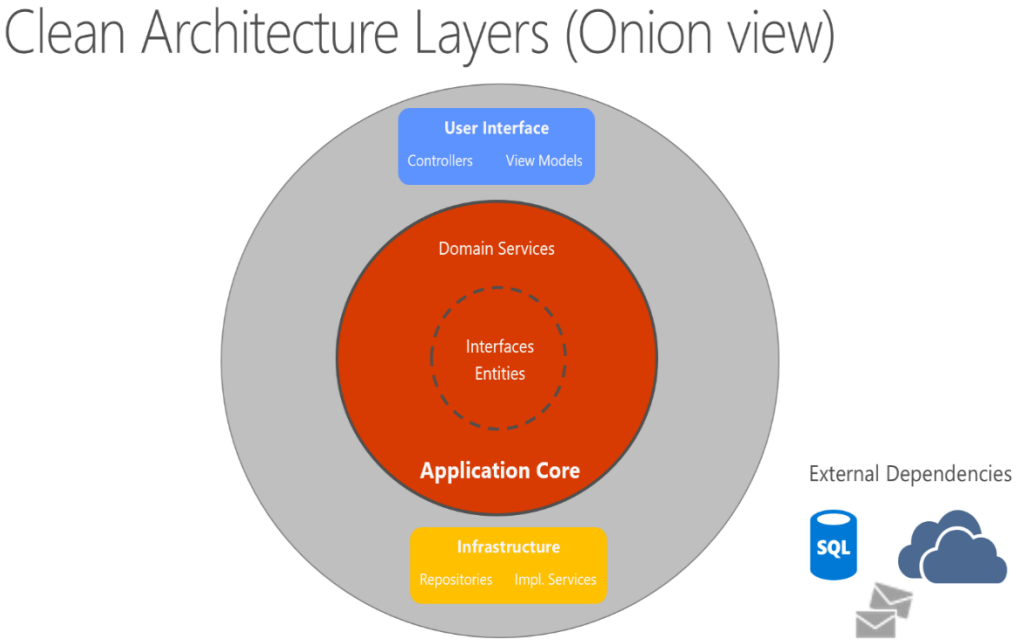


Image 11. Clean Architecture diagram, Source: https://learn.microsoft.com/en-us/dotnet/architecture/modern-web-apps-azure/common-web-application-architectures

Image 11 shows the layers of Clean Architecture, where one can see that infrastructure and implementation details depend on the Application Core, in which our Interfaces and Entities reside. In the Infrastructure layer, one can see 'Repositories' and 'Implementation Services'. These are implementations of abstractions, or interfaces, in the Application Core. The User Inteface layer should not know about the implementation types residing in the Infrastructure layer and only works with interfaces and entities defined in the Application Core.

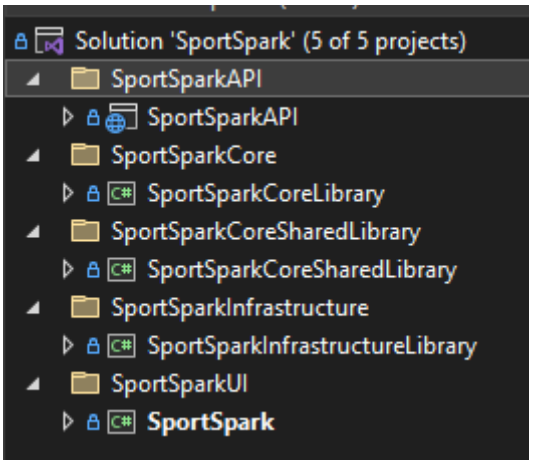


Image 12. Clean Architecture implementation in Visual Studio, Source: author screenshot

Image 12 shows the implementation of Clean Architecture in Visual Studio. Folders for the API (where Controllers reside), the Application Core, Application Core Shared, Infrastructure and the UI projects are created. The projects inside of the folders are, in this case, .NET MAUI class libraries.

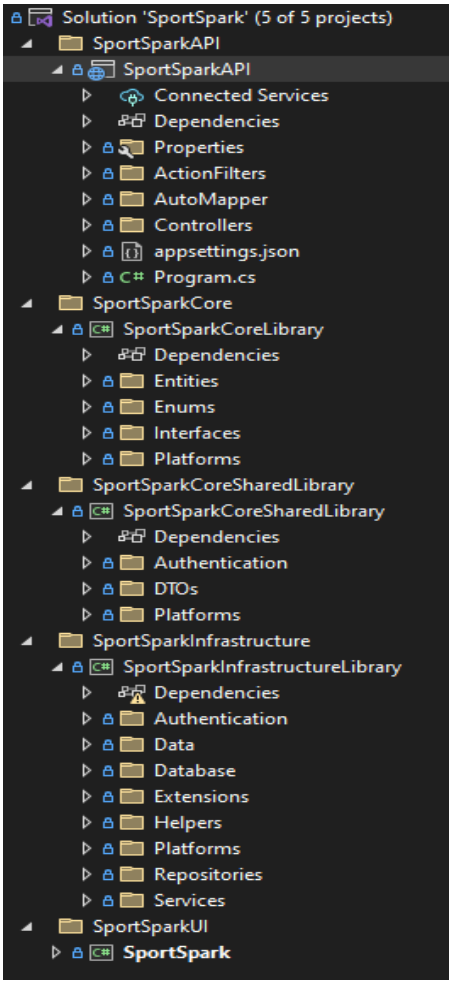


Image 13. Folders inside of Clean Architecture projects, Source: author screenshot

Inside the projects, one can see folders for aforementioned Interfaces, Entities, Services and Repositories. Furthermore, the Application Core also houses Enums. The Infrastructure layer has everything to do with the application's database, along with Helper classes and the implementations of repository interfaces and service interfaces. The Application Core Shared library houses Data Transfer Objects, which will be described in a later chapter, and classes for Authentication. The Shared library exists because Data Transfer Objects and Authentication classes are used by both the UI and the backend.

4.2. Repository pattern

As was mentioned in the previous chapter, Clean Architecture defines interfaces for Service and Repository classes. The repositories are part of the Repository pattern, which is a design pattern that allows developers to have a cleaner separation of code. A repository is a class that implements an interface, and is used to retrieve data from the database using an ORM (Object Relational Mapping) such as Entity Framework Core [14].

The Repository pattern is implemented in SportSpark in the following way: Controllers use Services, where all business logic is located, and which in turn use Repositories, which do naught but retrieve data from the database. To summarize, Repository pattern mediates data from and to the Domain and Data Access Layers [14]. Repositories define methods that are called from inside Services, and serve to retrieve, update or create data in the application database. API Controllers do not know about repositories. They merely use Services, which in turn use repositories, and return the returned values from the Service they call (in the case of a GET operation, for instance).

4.3. Data Transfer Objects

A Data Transfer Object or DTO is an object that carries data between application layers. Applications usually rely on a system of HTTP requests and calls to an API. Returning unprocessed entities from a database to the UI layer is a bad idea, as that might expose sensitive data [15]. For example, the *User.cs* class in SportSpark has a password property, because each user has a password in the database. Even though this password is hashed during user account creation, sending it to the frontend (especially when user 1 visits user 2's profile) is bad practice. HTTP request results may be intercepted, and sensitive data may be exposed to malicious users this way.

Data Transfer Objects serve to fix or alleviate this problem. A *UserDTO* will, in this example, opt to 'ignore' the password property from the *User* entity class during the mapping of *User.cs* to *UserDTO.cs*. A future chapter will go further into mapping these objects.

4.4. Model-View-ViewModel-Service pattern

.NET MAUI documentation describes a Model-View-ViewModel pattern, which consists of three core components: the model, the view, and the view model [16]. Typically, .NET MAUI involves creating a user interface using XAML, then using the code-behind to add code to work with the user interface. This leads to maintenance issues and tight coupling of UI and business logic, as well as difficulty of Unit Testing code [16]. MVVM helps separate UI and business logic, and is a best practice pattern that any developer using .NET MAUI should be aware of.

SportSpark goes one step further, using the MVVMS pattern, which adds a Service component to MVVM. In MVVMS, the View defines the structure, layout and appearance of the UI, using XAML and some code-behind (animations and such, not business logic). The ViewModel exists to house the business logic of the UI, implementing properties and commands which can be bound to using Bindings in XAML. In MVVMS, the ViewModel of each View should not know about the business logic of making and sending HTTP requests. Therefore, SportSpark defines a RestService class, which defines methods that are called from the ViewModels. RestService also implements a single instance of the HttpClient object, whereas it would need to be instantiated multiple times within multiple ViewModels, if not for RestService.

To summarize, MVVMS means that the View contains XAML where properties such as a Label's Text are bound to a property implemented in the ViewModel. The code-behind of the View serves only to unfocus, focus or animate (for example) view controls. The ViewModel implements properties and commands that define actions to be taken when, for instance, a button is pressed. It calls methods from the RestService class when it needs to send an HTTP request to the API, and then works with the returned values.

5. Backend development

This chapter marks the beginning of the multiplatform application development process. The first thing that most applications need is a database, which will hold the data that the application will use. Modeling a robust database is the foundation upon which the rest of the app will be built, so before getting into building the database itself, it will first be modeled. To model a database, an online tool such as *draw.io* can be used, where one can create diagrams that will represent tables in an SQL database, and the connections between these tables. It is often that the first version of a database model quickly becomes outdated. The database model that will be presented in the following subchapter is a diagram created inside of Microsoft SQL Server Management Studio, which represents the final version of the database.

5.1. Database modeling

The first version of the database model for SportSpark looked like the following image:

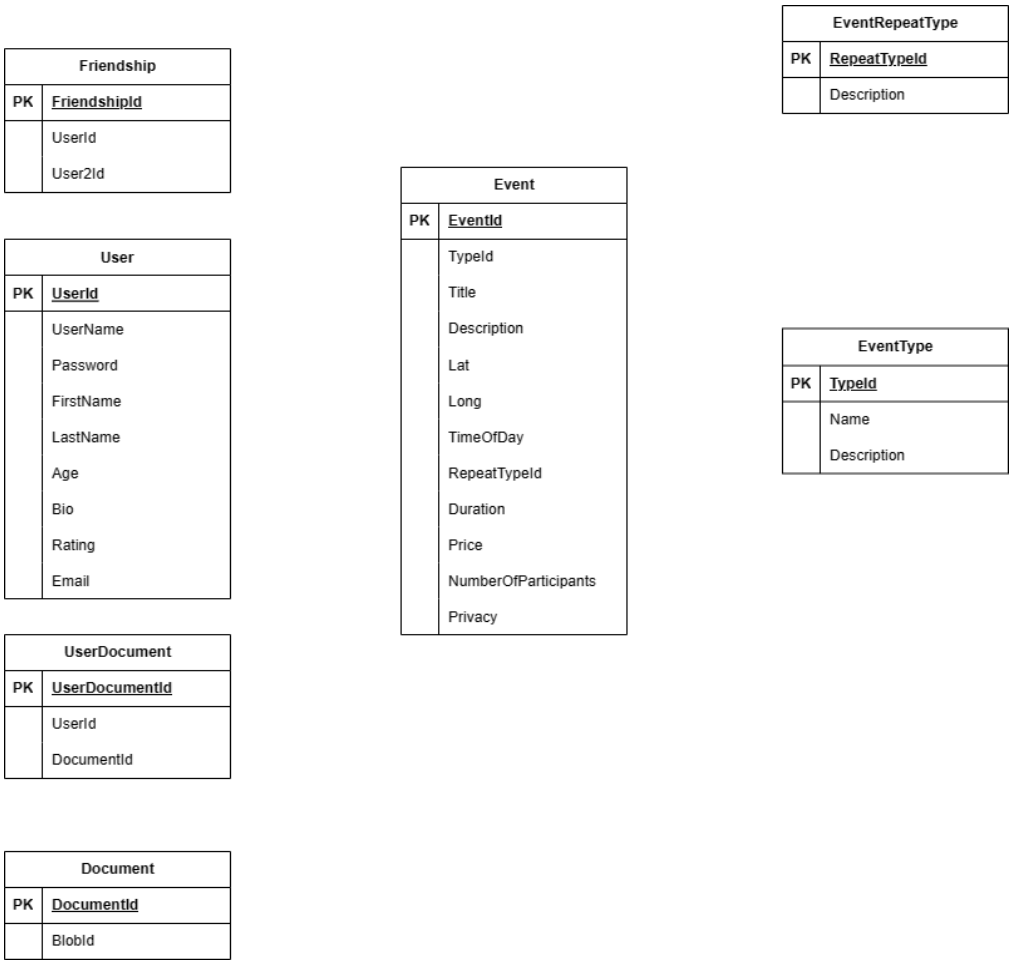


Image 14. SportSpark database model version 1, Source: author screenshot

However, while developing the application, numerous changes were required to be made. Keeping in mind the application idea, the author of this paper created a database model consisting of 6 tables:

1. Event
2. EventType
3. EventRepeatType
4. User
5. Friendship
6. Document

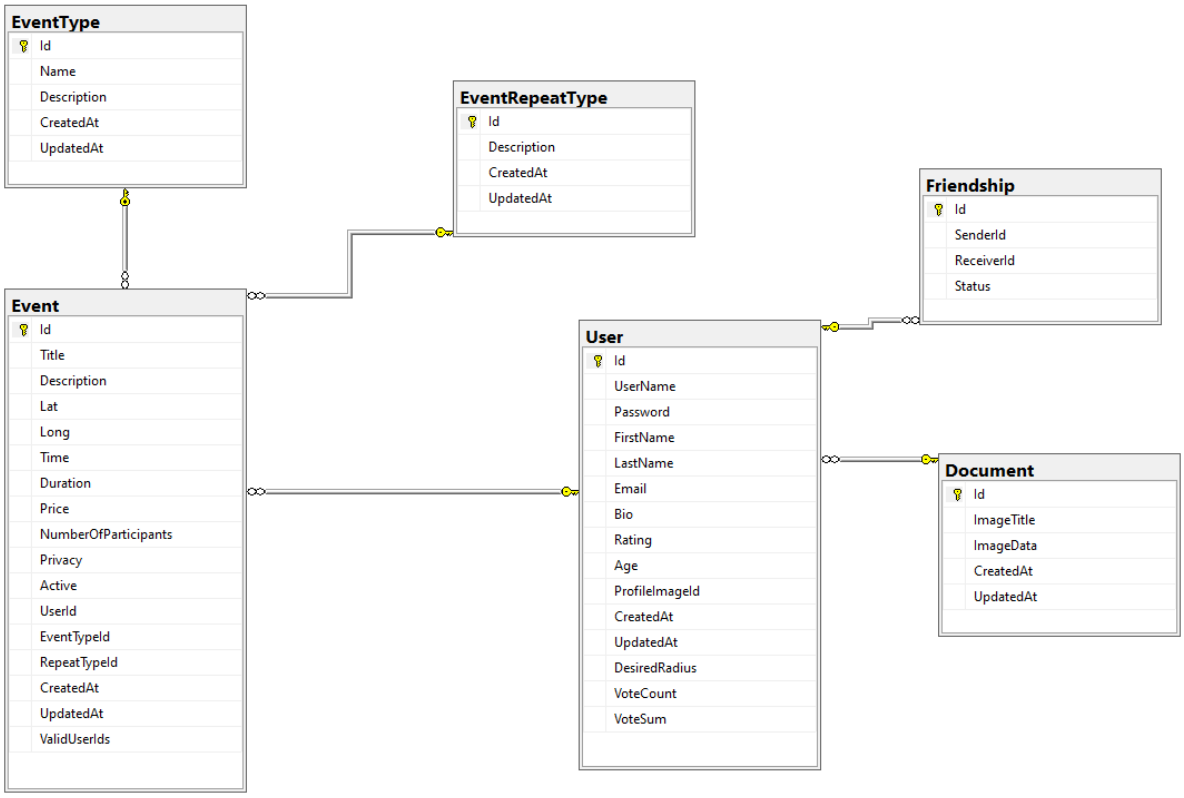


Image 15. SportSpark final database model, Source: author screenshot

5.2. Creating a database

As was mentioned in a previous chapter, the database will be created using Entity Framework Core 7's Code-First approach. In order to create a database, EF requires entities and a *DbContext* class.

An entity corresponds to a table in the newly created database. The table name may be defined by using an annotation: `[Table(„EventType“)]` will create a table named „EventType“ in the database, with properties defined in the *EventType.cs* class.

```
[Table("EventType")]
public class EventType : BaseEntity
{
    [Required]
    [StringLength(50)]
    public string Name { get; set; }

    [Required]
    [StringLength(150)]
    public string Description { get; set; }

    #region Relations
    public ICollection<Event> Events { get; set; }
    #endregion
}
```

The above code means that a table named „EventType“ will be created, with a property called „Name“, which cannot be null, and has a maximum length of 50 characters. Similarly, „Description“ is not nullable, and has a maximum length of 150 characters. The *ICollection<Event> Events* property means that multiple *Events* may be of one *EventType*. Other entities are created in a similar way. They need to be defined as a *DbSet<T>* inside of a class that inherits from *DbContext*.

```
public class SportSparkDBContext : DbContext
{
    public DbSet<User> Users { get; set; }
    public DbSet<Friendship> Friendships { get; set; }
    public DbSet<EventType> EventTypes { get; set; }
    public DbSet<EventRepeatType> EventRepeatTypes { get; set; }
```

```
    public DbSet<Event> Events { get; set; }
    public DbSet<Document> Documents { get; set; }
    public         SportSparkDBContext(DbContextOptions<SportSparkDBContext>
options) : base(options)
    {
    }
protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        base.OnModelCreating(modelBuilder);

        modelBuilder.Entity<User>()
            .HasMany(x => x.Events)
            .WithOne(x => x.User)
            .HasForeignKey(x => x.UserId);
        modelBuilder.Entity<User>().Property(x => x.Password)
            .UseCollation("SQL_Latin1_General_CP1_CS_AS");

        modelBuilder.Entity<Friendship>().HasKey(x => x.Id);
        modelBuilder.Entity<Friendship>()
            .HasOne(x => x.Sender)
            .WithMany(x => x.RequestedFriendships)
            .IsRequired()
            .OnDelete(DeleteBehavior.Cascade);
        modelBuilder.Entity<Friendship>()
            .HasOne(x => x.Receiver)
            .WithMany(x => x.ReceivedFriendships)
            .IsRequired()
            .OnDelete(DeleteBehavior.Restrict);
    }
}
```

The above code shows the *SportSparkDBContext* class, which inherits from *DbContext*. It has multiple *DbSet<T>* properties, one for each SportSpark entity. The constructor must have a parameter of *DbContextOptions<SportSparkDBContext>* and *:base(options)*. *OnModelCreating* is an overridden method inside of which a developer may use FluentAPI to specify relationships or other settings on the tables that will be created. For the *Friendship* entity, it is required to specify that it has one *Sender*, which is of type *User*, with many *RequestedFriendships* and vice-versa. This will mean that Entity Framework will know to populate the *User.RequestedFriendships* collection with *User* objects corresponding to the *SenderId* attribute. If a *User's* Id is a *SenderId* in a *Friendship* entity, then that *User's RequestedFriendships* collection will be populated with that *Friendship* entity.

Finally, to create the database that is described using the annotations inside Entity classes and the FluentAPI definitions inside of *OnModelCreating*, one may use the Package Manager Console and Migrations. Migrations are „a way to incrementally update the database schema to keep it in sync with the application's data model while preserving existing data in the database.“ [17] Before creating a Migration, a database connection string needs to be set up. Inside *Program.cs*, the following line is placed:

```
builder.Services.AddDbContext(builder.Configuration.GetConnectionString("Lo
                                calConnection"));
```

Inside *appsettings.json*, a connection string named *LocalConnection* is defined:

```
...
"ConnectionStrings": {
        "LocalConnection":
"Server=.;Database=SportSpark;Trusted_Connection=True;MultipleActiveResultS
ets=true;Encrypt=False;"
    },
...
```

This also serves to specify a name for the newly created database. As was previously mentioned, Migrations are part of EF Core, and one may be created using the Package Manager Console, for instance.

```
Add-Migration Init -o Data/Migrations
```

The above command will create a new Migration, named „Init“ in the folder Data/Migrations inside the project where a class that inherits from *DbContext* exists. EF Core will therefore create a database schema, and the database may be created using

```
Update-Database.
```

This is all that is required to create a database using EF Core. If a developer needs to change the database, they need not delete the database and recreate the schema, losing all data in the process. They may simply change an Entity as required, create a new Migration, and update the database.

5.3. Creating an API

An API serves as the „middleman“ between the UI and the database. The UI will send HTTP requests to SportSpark API endpoints, which will call Service class methods, which in turn call Repository class methods to retrieve data from the database. The Service classes contain the business logic and map Entity objects to DTO objects using a NuGet package called AutoMapper. They return these DTO objects to the API endpoints, which return them as a JSON to the UI.

The API endpoints are housed inside controllers. This chapter will look at *UserController.cs*, the services it uses, the *BaseController* it inherits from, error handling, and filtering HTTP requests. The API can be considered a website that, instead of a nice-looking UI, returns JSON files. Therefore, it has a domain, just like any website. This paper deals with a locally run application, so SportSpark's API's domain is *localhost:7181.* To specify a route to a specific controller on the API, one may use an annotation. The controller inherits from *BaseController* which is a custom class that inherits from *ControllerBase* (this inheritance is important).

```
[Route("api/v1/[controller]")]
[ApiController]
public class UserController : BaseController
{
	private readonly IUserService _userService;
	public UserController(IUserService userService)
	{
		_userService = userService;
	}

	[ActionFilters.AuthorizationFilter()]
	[HttpGet]
	[ProducesResponseType(typeof(UserDTO), 200)]
```

```
	public async Task<ActionResult<List<UserDTO>>> Get()
	{
		try
		{
			return await _userService.GetAllAsync();
		}
		catch (Exception ex)
		{
			return BadRequest(new ApiResponseHelper(400, ex.Message));
		}
	}
...
```

The above code shows the annotations required to specify a route for the controller, as well as define it as a controller. It inherits from *BaseController*, which implements one property:

```
public class BaseController : ControllerBase
{
	public int UserId
	{
		get
		{
			var userId = -1;
			_ = int.TryParse(this.User.Claims.FirstOrDefault(t => t.Type == "UserId")?.Value, out userId);
			return userId;
		}
	}
}
```

The *UserId* property will be used throughout some controllers. It contains the Id of the currently logged in user, or in other words, the user that is sending an HTTP request to the API. This

UserId is parsed from so-called „User claims“. „User claims“ are part of the API's authentication and authorization. API authentication and authorization is set up by writing the following lines into *Program.cs*:

```
builder.Services.AddAuthentication(opt =>
{
    opt.DefaultAuthenticateScheme = JwtBearerDefaults.AuthenticationScheme;
    opt.DefaultChallengeScheme = JwtBearerDefaults.AuthenticationScheme;
})
    .AddJwtBearer(options =>
    {
        var tokenData = builder.Configuration.GetSection("TokenData");
        options.TokenValidationParameters                                    =
TokenValidationConfiguration.GetTokenValidationParameters(tokenData["Issuer
"],
            tokenData["Audience"], tokenData["SecretKey"]);
    });
...
app.UseAuthentication();
app.UseAuthorization();
...
```

This sets up the JSON Web Token authentication for the API.

5.3.1. JWT Authentication

JWTs define a way to securely transmit information between parties as a JSON object. Part of the token is the payload, which contains claims, which contain, for instance, user data. An example JWT is:

eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJVc2VySWQiOiI3IiwibmJmIjoxNjg2MzkzMzk2LCJleHAiOjE2ODY5OTgxOTYsImlzcyI6IlNwb3J0U3BhcmsiLCJhdWQiOiJTcG9ydFNwYXJrIn0.zaaPTBrZSpZ0aZbb24QlvpH6NGFFTTyLhAlIw5_mvHI.

When decoding this JWT using an online tool, one can see its payload:

{
"UserId": "7",
"nbf": 1686393396,
"exp": 1686998196,
"iss": "SportSpark",
"aud": "SportSpark"
}

This JWT token is attached to each HTTP request's header the frontend sends to the backend. The backend, or API, then checks the JWT token before each HTTP request is processed. If the user is not logged in (therefore the JWT is broken or is not attached to the HTTP request) it returns *401 Unauthorized*. This processing takes place inside of the *AuthorizationFilter* class, which is part of the `[ActionFilters.AuthorizationFilter()]` annotation. The *AuthorizationFilter* class overrides the *OnActionExecuting* method:

```
public class AuthorizationFilter : ActionFilterAttribute
{
	public override async void OnActionExecuting
	(ActionExecutingContext context)
	{
		var httpUser = context.HttpContext.User;

		if (!httpUser.Identity.IsAuthenticated) context.Result = new
	ForbidResult();
		else base.OnActionExecuting(context);
	}
}
```

This class checks if the user sending the HTTP request to an API endpoint is authenticated. If not, it returns *401 Unauthorized*, as was explained previously. This is how to prevent malicious users from tampering with the application's database.

The JSON Web Token is created during a user's log in process. *AuthenticationController.cs* defines a *Login* method:

```
[HttpPost("login")]
public async Task<ActionResult<UserDTO>> Login(UserLogin userLogin)
{
	try
	{
		var user = await
	_userService.UserValid(userLogin.EmailOrUserName,
	userLogin.Password);
		if (user is not null)
		{
			var userDto = _userService.Login(user);
			return Ok(userDto);
		}
		return Unauthorized();
	}
	catch (Exception ex)
	{
		return BadRequest(new ApiResponseHelper(400, ex.Message));
	}
}
```

UserService.Login creates the user claims and the JWT:

```
public UserDTO Login(User user)
{
	UserDTO userDto = _mapper.Map<UserDTO>(user);
	List<Claim> claims = new()
	{
```

```
            new Claim("UserId", user.Id.ToString())
      };
      AuthenticationInfo authInfo = new()
      {
            AccessToken         =         _tokenService.GenerateJwt(claims,
      _tokenDataConfiguration.AccessTokenExpirationInMinutes),
            RefreshToken        =         _tokenService.GenerateJwt(claims,
      _tokenDataConfiguration.RefreshTokenExpirationInMinutes)
      };
      userDto.AuthenticationInfo = authInfo;
      return userDto;
}
```

The *TokenService's GenerateJwt* method creates the token string that was displayed earlier. The string is attached to a *UserDTO's AuthenticationInfo* property, which defines *AccessToken* and *RefreshToken*. When the user logs in, the *UserDTO* object is returned, having *AuthenticationInfo* populated with the user's JWT, which is then attached to the *HttpClient's* headers on the frontend.

5.3.2. Handling HTTP requests

After an incoming HTTP request passes through the filter, the controller calls a method in the corresponding Service class.

```
[ActionFilters.AuthorizationFilter()]
[HttpGet("{id}")]
[ProducesResponseType(typeof(UserDTO), 200)]
public async Task<ActionResult<UserDTO>> GetById(int id)
{
      try
      {
            return await _userService.GetByIdAsync(id);
      }
      catch (Exception ex)
```

```
        {
                return BadRequest(new ApiResponseHelper(400, ex.Message));
        }
}
```

In the case of *UserController.cs*' *GetById* method, the id passed to the endpoint is then passed to `UserService.GetByIdAsync()`. The *_userService* object is of type *IUserService*, which is an interface that *UserService* implements. As could be seen in the code previously, the interface is injected using Dependency Injection. To do this, the following line is required in *Program.cs*:

```
services.AddScoped<IUserService, UserService>();
```

In SportSpark, this is implemented in a different way. In Program.cs, this line is written:

```
builder.Services.RegisterServices();
```

The *RegisterServices* method is an extension, and resides inside *DependencyContainer.cs* in the Infrastructure layer. It adds all services and repositories used with Dependency Injection. This is to prevent unclean code inside *Program.cs*, as adding services for Dependency Injection can quickly become a mess, consisting of many lines of code.

IUserService inherits from *IBaseService<UserDTO>*, which defines methods such as *GetAllAsync, GetByIdAsync, CreateAsync* and other methods that all service classes will use. Other than the methods *IUserService* inherits, it also defines some of its own. For this particular example, *GetByIdAsync* is implemented inside of *UserService* in the following way:

```
public async Task<UserDTO> GetByIdAsync(int id)
{
        var user = await _userRepository.Fetch()
                .Include(u => u.Events)
                .Include(u => u.ReceivedFriendships)
                    .ThenInclude(_ => _.Sender)
                        .ThenInclude(u => u.ProfileImage)
                .Include(u => u.RequestedFriendships)
                    .ThenInclude(_ => _.Receiver)
```

```
                        .ThenInclude(u => u.ProfileImage)
                .Include(u => u.ProfileImage)
                .FirstOrDefaultAsync(u => u.Id == id);
        return _mapper.Map<UserDTO>(user);
}
```

As one can see, the service simply calls the corresponding repository, specifies which objects to include (tables to join) and return the user whose *Id* is equal to the one the HTTP request sent. Finally, the entity object is mapped to a *UserDTO* object using **AutoMapper**. AutoMapper is a NuGet package that serves to automatically and more easily map entity objects to DTO objects. To set it up, one must simply add the following line to *Program.cs*:

```
builder.Services.AddAutoMapper(AppDomain.CurrentDomain.GetAssemblies());
```

Furthermore, a class that inherits from *Profile* must be created, where map settings are specified. The following code displays an example mapping between *User* to *UserDTO* and vice-versa:

```
public class AutoMapperProfile : Profile
{
        public AutoMapperProfile()
        {
                CreateMap<User, UserDTO>()
                .ForMember(x => x.RequestedFriendships, opt => opt.MapFrom(_ => _.RequestedFriendships)).MaxDepth(2)
                .ForMember(x => x.ReceivedFriendships, opt => opt.MapFrom(_ => _.ReceivedFriendships)).MaxDepth(2)
                .ForMember(x => x.Events, opt => opt.MapFrom(_ => _.Events)).MaxDepth(2)
                .ForMember(x => x.Password, opt => opt.Ignore())
                .ForMember(x => x.ProfileImageData, opt => opt.MapFrom(_ => _.ProfileImage.ImageData));
                CreateMap<UserDTO, User>();
...
```

As one can see, when mapping the *User* object to a *UserDTO* object, the password property is ignored. This prevents a user's password from ever being sent to the frontend.

5.3.2.1. Error handling

If an exception is raised anywhere between the endpoint's service method call and the returning of the values, the API will return an object containing the error status code, and a message describing the error.

```
...
catch (Exception ex)
{
      return BadRequest(new ApiResponseHelper(400, ex.Message));
}
...
```

ApiResponseHelper is a custom class that consists of two properties: the error status code and an error message. This error message can be custom. For example, in *UserService* an exception is thrown if a user's desired radius is over 500km when trying to update the user:

...

```
if (entity.DesiredRadius > 500)
{
      throw new Exception("Radius cannot be higher than 500.");
}
...
```

This custom error message will be returned to the frontend, where it may be displayed inside of a dialog, alert, toast or snackbar.

6. Frontend development

The following chapters will go through the process of brainstorming user interface ideas, converting those ideas into a real, usable UI using XAML and writing necessary code in the views' code-behind and view models.

6.1. Designing a user interface

As was previously mentioned, user interface ideas were created using a tool called Figma. This chapter will display each of SportSpark's views as first imagined in Figma. Note that it is usual for the UI to change during the development process, so the end result may differ from the following images.

Image 16. SportSpark splash screen, Source: Figma export

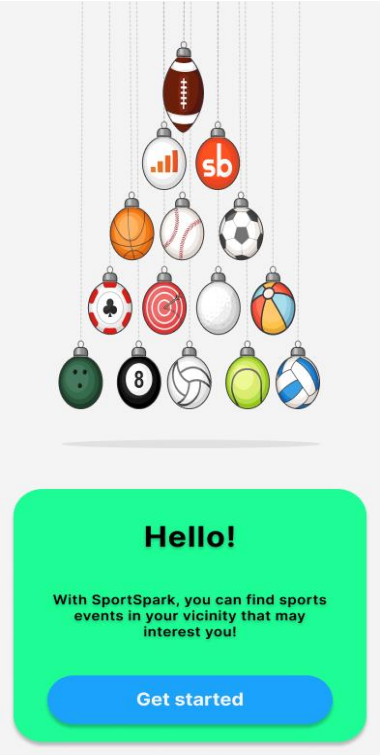


Image 17. FirstStartupPage, Source: Figma export

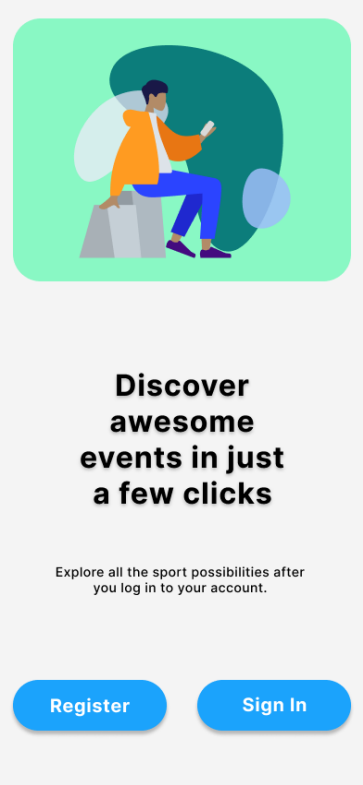


Image 18. StartingView, Source: Figma export

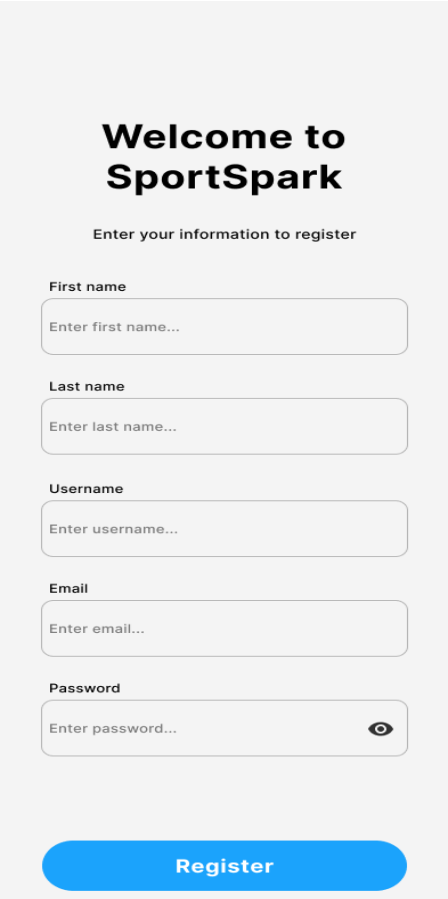


Image 19. Register, Source: Figma export

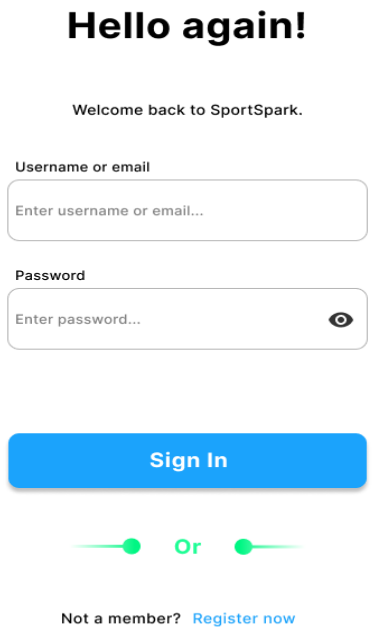


Image 20. SignIn, Source: Figma export

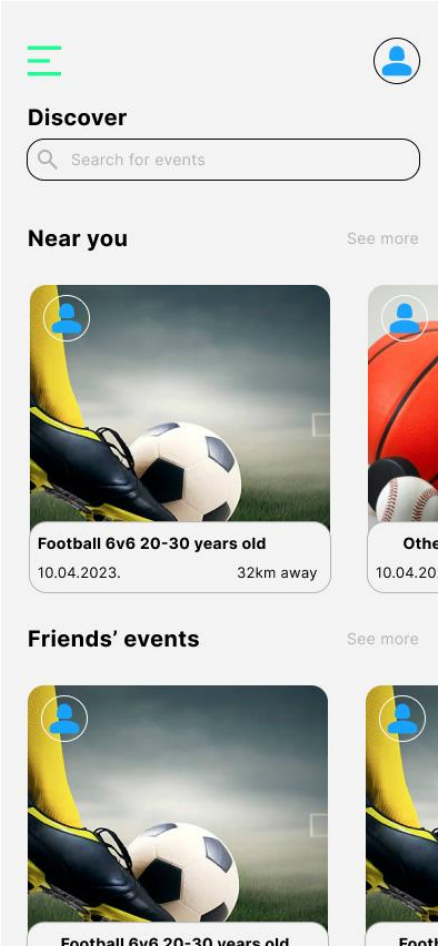


Image 21. Home, Source: Figma export

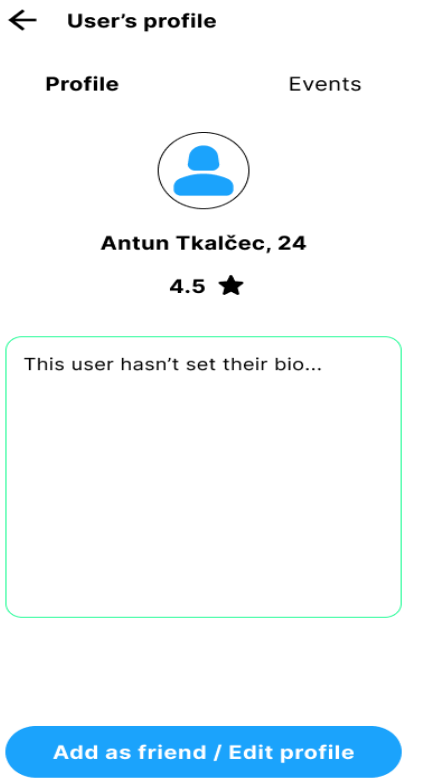


Image 22. Profile, Source: Figma export

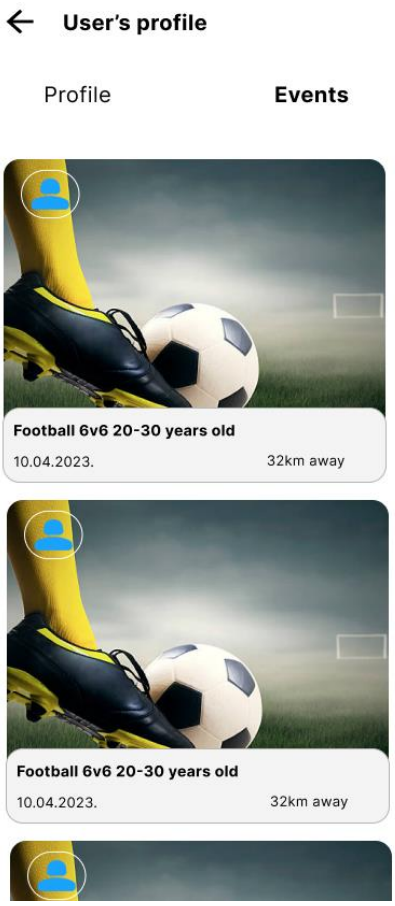


Image 23. Profile 2, Source: Figma export

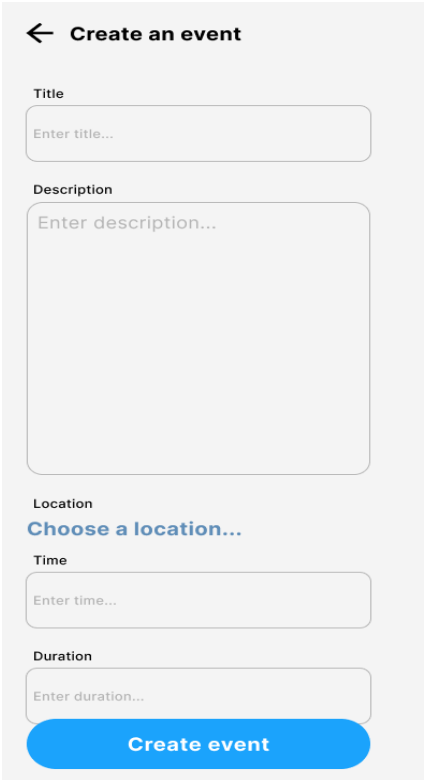


Image 24. CreateEvent, Source: Figma export

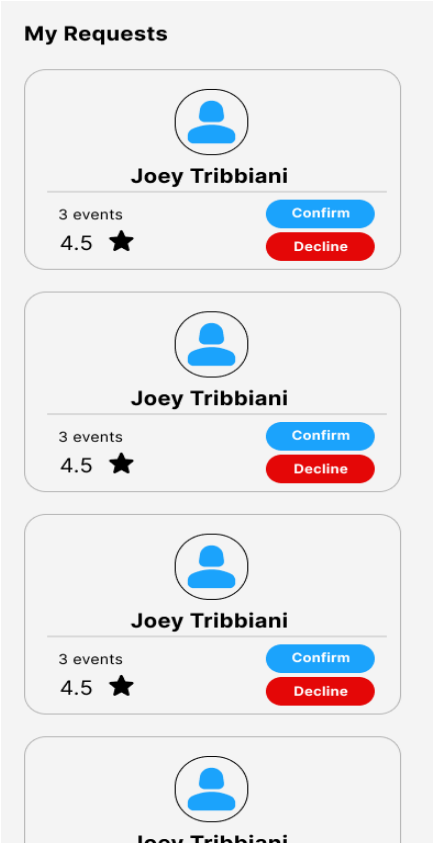


Image 25. Friends, Source: Figma export

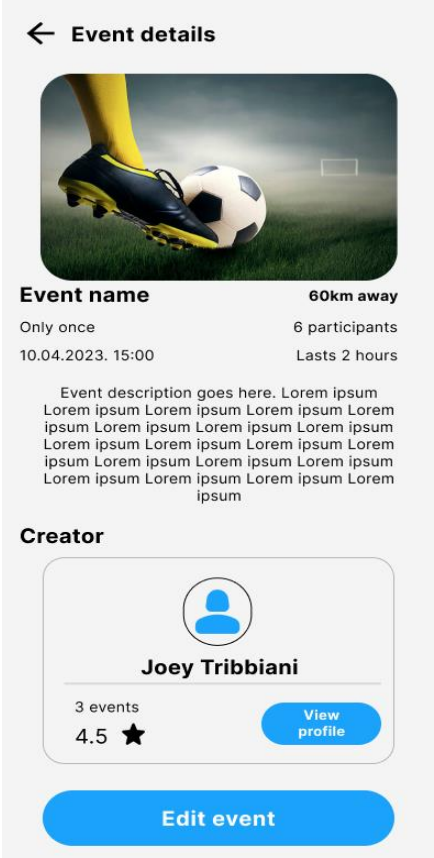


Image 26. EventDetails, Source: Figma export

As one can see, SportSpark consists of the following views/pages:

1. FirstStartupView
2. StartingView
3. SignInView
4. RegisterView
5. HomeView
6. ProfileView (profile 2 relates to the „events“ part of the view)
7. CreateEventView
8. FriendsView (plus a FriendsListView that looks the same, showing confirmed requests)
9. EventDetailsView

In addition, SportSpark has a „splash screen“, which consists of a simple white background and the SportSpark logo, which can be seen on Image 16.

6.2. Building a user interface

This chapter will go through each of the previously mentioned views, displaying some interesting or more advanced parts of the code. As this paper assumes the reader has at least some knowledge of XAML, the entirety of each view's XAML code will not be displayed, for brevity. The reader may visit the project's GitHub page and view the code in its entirety.

6.2.1. FirstStartupView

FirstStartupView is a view that should appear to the user only once, when they are first starting SportSpark. It tells the new user what they can expect from SportSpark, while displaying an interesting animation that should grab their attention. The topmost part of this view is a *Lottie animation.* A developer may go to *Lottie's* website, find an animation they like, download the animation's .json file, import it into the .NET MAUI project's *Resources/Raw* folder, and set its Build Action to *MauiAsset*. To use the .json file and show the animation in .NET MAUI, a NuGet package is required. In this case, *SkiaSharp.Extended.UI.Maui* was used. After installing this NuGet package, the animation .json file can be consumed inside XAML:

```
...
xmlns:skia="clr-
namespace:SkiaSharp.Extended.UI.Controls;assembly=SkiaSharp.Extended.UI"
...
```

```
<skia:SKLottieView HeightRequest="450" WidthRequest="450"
IsAnimationEnabled="True" RepeatCount="-1" Source="greet.json"
TranslationY="-40" Grid.RowSpan="2" Grid.Row="0"/>
...
```

First, the developer must add the NuGet package's namespace. Then, using the namespace, the developer may use the *SKLottieView* control to display a .json animation of their choice.

FirstStartupView's text, which can be seen on Image 17, is created using *Label* controls. The *Labels* have their *Text* property **bound** to a property on *FirstStartupViewModel.cs*:

```
...
<Label Text="{Binding Language[Hello!]}" FontSize="32"
HorizontalOptions="CenterAndExpand" TextColor="Black" FontAttributes="Bold"
Shadow="{StaticResource DefaultShadow}" VerticalOptions="StartAndExpand"
Margin="0, 10, 0, 0"/>
...
```

This is where **language localization** comes into play. As one can see, the *Label's Text* property is bound to *Language[Hello!]*. In the ViewModel, *Language* is a property:

```
...
public LanguageHelper Language
{
	get
	{
		return LanguageHelper.Instance;
	}
}
...
```

It returns an instance of *LanguageHelper*:

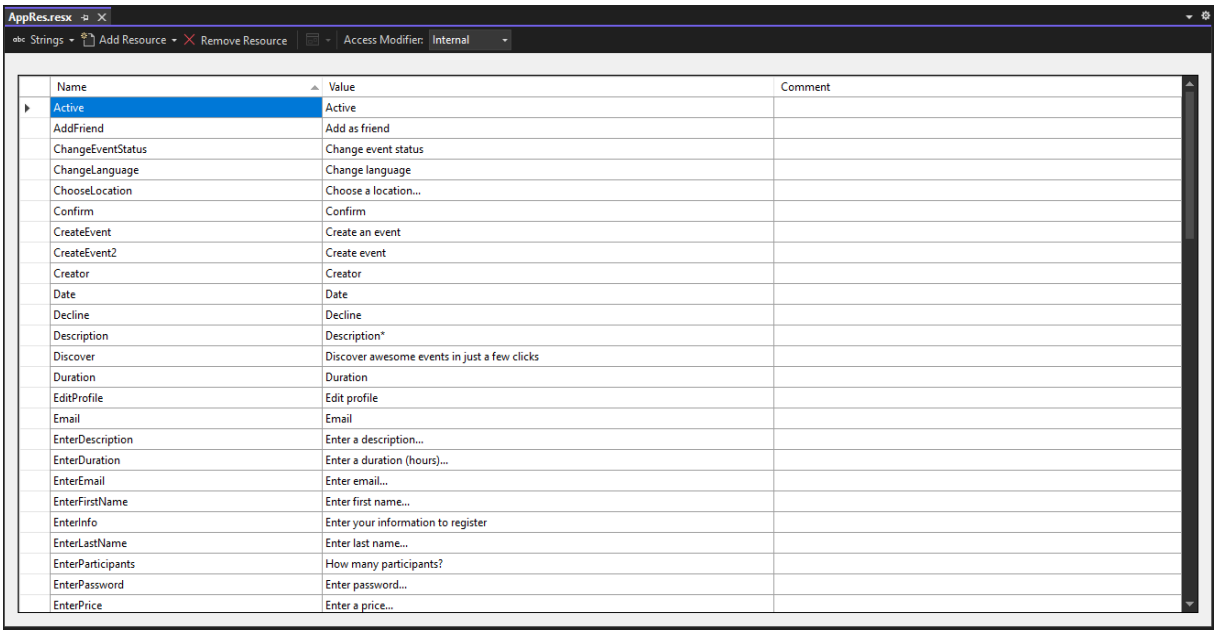
```
...
static LanguageHelper instance;
```

```
public static LanguageHelper Instance
{
	get
	{
		if (instance == null)
		{
			instance = new LanguageHelper();
		}
		return instance;
	}
}
...
```

Then, *Language* can be accessed with, for example, *Language[Hello!]*:

```
...
public string GetString(string resourceName)
{
	return manager.GetString(resourceName);
}
public string this[string key] => GetString(key);
...
```

The *manager* property is a *ResourceManager* type property that equals *Resources.AppRes.ResourceManager.* This means *manager* points to the *ResourceManager,* which works will files such as *AppRes.hr.resx* and *AppRes.resx,* where key-value pairs are created for any text in the application. *Hello!* would be the key in this instance, and „Hello!“ would be the value, if the current application culture is English. If the user sets the application culture to Croatian, the value is read from *AppRes.hr.resx*, under the same *Hello!* key. Then, the *Label's* text changes to „Bok!“. Language localization is done the same way throughout most of the rest of the application.



Name	Value	Comment
Active	Active	
AddFriend	Add as friend	
ChangeEventStatus	Change event status	
ChangeLanguage	Change language	
ChooseLocation	Choose a location...	
Confirm	Confirm	
CreateEvent	Create an event	
CreateEvent2	Create event	
Creator	Creator	
Date	Date	
Decline	Decline	
Description	Description*	
Discover	Discover awesome events in just a few clicks	
Duration	Duration	
EditProfile	Edit profile	
Email	Email	
EnterDescription	Enter a description...	
EnterDuration	Enter a duration (hours)...	
EnterEmail	Enter email...	
EnterFirstName	Enter first name...	
EnterInfo	Enter your information to register	
EnterLastName	Enter last name...	
EnterParticipants	How many participants?	
EnterPassword	Enter password...	
EnterPrice	Enter a price...	

Image 27. AppRes.resx, Source: author screenshot

Image 27 displays the *AppRes.resx* file. .NET MAUI knows to use the values from this file when the application's „culture“ is set to *en-US*. If the culture is set to *hr-HR*, it will use the values from *AppRes.hr.resx.*

On Image 17, one can also see a button. Inside XAML, the button has a *Command* property that is bound to *FirstStartupViewModel's GetStartedCommand*. However, *FirstStartupViewModel* defines a method called *GetStartedAsync*. SportSpark uses .NET MAUI's Community Toolkit, which simplifies binding properties to things in the view model. Because of the Community Toolkit, *FirstStartupView* defines the method this way:

```
...
[RelayCommand]
async Task GetStartedAsync()
{
	var status = await Permissions.RequestAsync<Permissions.LocationWhenInUse>();
	if (status == PermissionStatus.Granted)
	{
		await Application.Current.MainPage.Navigation.PushAsync(new StartingView());
	}
```

```
        else
        {
              await Application.Current.MainPage.ShowPopupAsync(new
        ErrorPopup("SportSpark cannot function without location
        permissions"));
        }
}
...
```

[RelayCommand] is an annotation that tells the compiler to compile *GetStartedAsync* as *GetStartedCommand,* as well as take care of the rest of the code required to assure that commands are bound properly. Other views will have things bound to properties in a similar way.

6.2.2. StartingView

This view is quite similar to the previous view. However, once the user visits this view and selects to leave it, thus entering the application proper, SportSpark should not show this view and the previous one to the user again. This is achieved using *Preferences*:

```
...
Preferences.Set("welcomed", "1");
...
```

Once the user exits this view in order to begin registering or signing in, „welcomed“ is set to „1“ in the application's preferences. SportSpark checks if the user has gone through the first two views in *App.xaml.cs'* constructor*:*

```
...
public App()
{
        InitializeComponent();
        SetCulture();
        if (!(Preferences.Get("welcomed", "0") == "1"))
        {
```

```
            MainPage = new NavigationPage(new FirstStartupView());
      }
      else
      {
            MainPage = new AppShell();
      }
}
...
```

This code checks if „welcomed“ is not „1“, and if it isn't, the app's *MainPage* is set to *FirstStartupView*. Otherwise, the app instantiates a new instance of *AppShell*.

6.2.2.1. AppShell

AppShell, by default, displays the *SignInView*. However, if the user has previously signed in, they might be already authenticated. To prevent forcing the user to sign in each time they start the application, the access token that was explained in the backend development part of this paper is inserted into the application preferences. The token lasts for one week. *AppShell* checks if the access token exists in preferences, and if it is not expired. If both of those are true, *AppShell's ContentTemplate* will instead be *HomeView*. After *FirstStartupView* and *StartingView*, *AppShell* is used for SportSpark's navigation.

6.2.3. RegisterView

RegisterView is the first view that binds controls' properties to properties using the Community Toolkit in the view model. For example, Image 19 shows text input fields. The text input field for the user's first name, in XAML, looks like this:

```
...
<Entry x:Name="firstName" Placeholder="{Binding Language[EnterFirstName]}"
TextColor="Black" PlaceholderColor="{StaticResource LightThemeEntry}"
Completed="FirstName_Completed"             Unfocused="FirstName_Unfocused"
Text="{Binding FirstName}">
      <Entry.Keyboard>
            <Keyboard x:FactoryMethod="Create">
```

```
                  <x:Arguments>
            <KeyboardFlags>CapitalizeWord</KeyboardFlags>
                  </x:Arguments>
            </Keyboard>
      </Entry.Keyboard>
</Entry>
...
```

As one can see, *Completed* and *Unfocused* events correspond to methods in *RegisterView's* code-behind, as they have to do with the UI. The *Entry* control's *Keyboard* property is custom, making each word in the input have its first letter capitalized. *Text* is bound to *FirstName,* which is a property on the view model:

```
...
[ObservableProperty]
[NotifyPropertyChangedFor(nameof(FirstNameValue))]
string firstName = string.Empty;
public string FirstNameValue => FirstName;
...
```

Annotations like *[ObservableProperty]* and *[NotifyPropertyChangedFor(nameof(FirstNameValue))]* are part of .NET MAUI's Community Toolkit, and serve to automatically generate necessary code required for binding. The reader is hereby encouraged to refer to the Community Toolkit's documentation for more details on this, as the particulars of certain NuGet packages are out of scope for this paper, for brevity.

After the user inputs the required fields and clicks the button to register, the following code is invoked:

```
...
if (await _restService.RegisterAsync(userDTO))
{
      await Toast.Make("Registration successful", ToastDuration.Short, 24).Show();
      await _navigationService.NavigateToAsync("..");
```

```
}
...
```

This code invoked the *_restService.RegisterAsync()* method, which is part of the MVVMS pattern. The view model, here, does not know about the workings of *HttpClient* and sending HTTP requests.

6.2.4. SignInView

As for *SignInView*, what is special about it is that it sets the authorization headers for the application's *HttpClient* singleton, as well as setting the access token and refresh token in the application's preferences.

```
...
HttpResponseMessage response = await _httpClient.PostAsync($"{SettingsManager.BaseURL}/Authentication/login", content);


if (response.IsSuccessStatusCode)
{
      string responseContent = await response.Content.ReadAsStringAsync();
      UserDTO userDto = JsonConvert.DeserializeObject<UserDTO>(responseContent);
      AuthenticationInfo authInfo = userDto.AuthenticationInfo;
      JwtSecurityTokenHandler handler = new();
      var tokenS = handler.ReadToken(authInfo.AccessToken) as JwtSecurityToken;
      Preferences.Set(AccessTokenKey, authInfo.AccessToken);
      Preferences.Set(RefreshTokenKey, authInfo.RefreshToken);
      Preferences.Set(TokenExpirationKey, tokenS.ValidTo);
      _httpClient.DefaultRequestHeaders.Authorization = new AuthenticationHeaderValue("Bearer", authInfo.AccessToken);
...
```

Once the user inputs their username/email and password, an HTTP request is sent to *api/Authentication/Login,* which returns the user's Json Web Token. The JWT is then set into the application's preferences and attached to *RestService's _httpClient's* authorization headers. This will prevent the user from having to sign in for one week.

6.2.5. HomeView

There are multiple interesting things to do with *HomeView*. Here, .NET MAUI's *WeakReferenceMessenger* is finally used. A custom view control is part of *HomeView.xaml*, and the user's location data is retrieved. *HomeView* also differs the most from the UI idea created in Figma, because the *ScrollView* and *CollectionView* controls' scrolling capabilities conflict with one another, essentially preventing both of them being part of a view.

Image 28. Final version of HomeView, Source: author screenshot

6.2.5.1. WeakReferenceMessenger

Inside *HomeView's* code-behind, this code is part of the constructor:

```
...
WeakReferenceMessenger.Default.Register<Message>(this, (r, m) =>
{
      OnMessageReceived();
});
...
private async void OnMessageReceived()
{
      await btmGrid.TranslateTo(0, 400, 250, Easing.SinInOut);
}
```

This sets up *HomeView.xaml.cs* to listen for messages. Upon receiving a message, it „translates“ the *btmGrid* view control. The *btmGrid* view control is a grid that, by default, sits below the *HomeView*. *OnMessageReceived* brings the grid up into visibility. The grid contains the custom view control, *MenuView*:

```
...
<Grid Grid.Row="1" VerticalOptions="End" TranslationY="400" x:Name="btmGrid"
Grid.ColumnSpan="2">
      <views:MenuView HeightRequest="400" Padding="10" Grid.RowSpan="2"
      Grid.ColumnSpan="2"/>
</Grid>
...
```

MenuView contains buttons that the user can click in order to sign out, change the application's language, visit their profile and so on. Once the button to visit the user's own profile is clicked, this code is invoked:

```
...
private void GoToProfile(object sender, EventArgs e)
{
      WeakReferenceMessenger.Default.Send(new Message("GoToProfile"));
}
...
```

This sends a message containing „GoToProfile“, which is then caught inside *HomeViewModel.cs:*

```
...
public async void Receive(Message message)
{
      switch (message.Value)
      {
            ...
            case "GoToProfile":
                  await
            _navigationService.NavigateToAsync(nameof(ProfileView),      new
            Dictionary<string, object>
                  {
                        { "User", LoggedInUserValue }, { "SameUser", true },
                  { "UserIsNotFriend", true }, { "UserProfilePicture",
                  LoggedInUserValue.ProfileImageData }
                  });
                  break;
...
```

This code shows that upon receiving the „GoToProfile“ message, the user is navigated to the *ProfileView.*

6.2.5.2. Retrieving location data

As for retrieving the user's location data, which is required on this view in order to retrieve events in the users' vicinity, this code is that is required:

```
...
GeolocationRequest       request       =       new(GeolocationAccuracy.Best,
TimeSpan.FromSeconds(60));
Location = await Geolocation.Default.GetLocationAsync(request);
...
```

The above code uses MAUI's power to invoke platform specific APIs, and retrieve the device's location.

6.2.6. ProfileView

ProfileView consists of two side-by-side grids, one of which is placed to the side of the Visible part of the device's screen, similar to *MenuView* on *HomeView*. This way, it appears as if it is two pages instead of one. Once the user clicks the „Events“ *Label,* the *EventsLayout* grid is moved to the center of the screen:

```
...
ProfileLabel.FontAttributes = FontAttributes.None;
EventsLabel.FontAttributes = FontAttributes.Bold;
EventsLayout.TranslateTo(0, 0, 250, Easing.SinInOut);
await  ProfileLayout.TranslateTo(DeviceDisplay.Current.MainDisplayInfo.Width * 2, 0, 250, Easing.SinInOut);
...
```

The above code once again displays the usage of MAUI's power to access device specific APIs in order to retrieve the device's display width, and use it to move the *ProfileLayout* to the side of the screen, while moving *EventsLayout* to the center. Another interesting thing on *ProfileView* is that it changes depending on if the user is visiting their own profile, or another user's profile. This is achieved using bindings and **converters**. For example, the user's age is not required, so SportSpark needs to not display an empty *Label* in the case of a user's age being *null*.

```
...
<Label FontSize="20" FontAttributes="Bold" TextColor="Black" HorizontalOptions="Center" IsVisible="{Binding UserValue.Age, Converter={StaticResource IsNotNullConverter}}">
    <Label.Text>
        <MultiBinding StringFormat="{}{0} {1}, {2}">
            <Binding Path="UserValue.FirstName"/>
            <Binding Path="UserValue.LastName"/>
            <Binding Path="UserValue.Age"/>
        </MultiBinding>
```

```
      </Label.Text>
</Label>
...
```

The above code displays the *IsVisible* binding, where the user's *Age* property is sent to a converter called *IsNotNullConverter*, which happens to be part of .NET MAUI's Community Toolkit and returns *true* or *false* depending on *UserValue.Age's* value. If the user's age is *null*, the *Label* is not part of the view. A developer may create their own custom converters. Custom converters are regular classes that must inherit from *IValueConverter*, thus inheriting the *Convert* and *ConvertBack* methods. *IMultiValueConverter* also exists, but MAUI seems to currently have a bug at the time of the making of this paper which prevents the multi-value converter from working.

If the user is visiting their own profile, they may click their profile picture, which will invoke platform specific APIs to open the platform's *FilePicker.* The user may then select an image that will become their new profile picture:

```
...
var res = await FilePicker.PickAsync(new PickOptions
{
      PickerTitle = "Choose a new profile picture",
      FileTypes = FilePickerFileType.Images
});
...
```

The file must then be converted to a byte array before being saved to the database:

```
...
var stream = await res.OpenReadAsync();
byte[] imageData;
using var memoryStream = new MemoryStream();
await stream.CopyToAsync(memoryStream);
imageData = memoryStream.ToArray();
...
```

For a more detailed look at converting image files to byte arrays and saving them to the database, the reader may visit the projects GitHub page.

6.2.7. CreateEventView

On *CreateEventView,* there is a modal view that opens once the user selects to choose the event's location. Once the *Label* to open this modal is clicked, the following code is invoked:

```
...
var res = await Application.Current.MainPage.ShowPopupAsync(new LocationSelectionPopup());
if (res is List<double> result)
{
	...
	NewEvent.Lat = (decimal?)result[0];
	NewEvent.Long = (decimal?)result[1];
}
...
```

The *LocationSelectionPopup* is a view that inherits from .NET MAUI Community Toolkit's *Popup* class, which allows developers to display simple popups as part of their user experience. The popup contains a *map* control, which is part of the Microsoft.Maui.Controls.Maps NuGet package:

```
...
<Grid RowDefinitions="*" ColumnDefinitions="*" Margin="5" x:Name="mainGrid">
	<maps:Map x:Name="map"/>
</Grid>
...
```

In the popup's code-behind, the user's location is first retrieved, after which the map is instantiated, and moved to around the user's location:

```
...
```

```
GeolocationRequest request = new(GeolocationAccuracy.Best,
TimeSpan.FromSeconds(60));
Location location = await Geolocation.Default.GetLocationAsync(request);
if (location != null)
{
	map.MapClicked += OnMapClicked;
	map.Pins.Add(new Pin
	{
		Label = "You",
		Location = location
	});
	map.MoveToRegion(MapSpan.FromCenterAndRadius(location,
Distance.FromKilometers(0.4)));
}
...
```

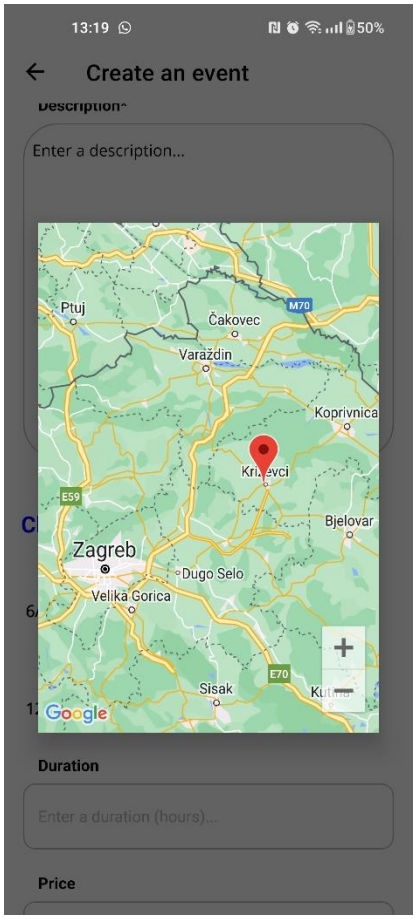


Image 29. LocationSelectionPopup, Source: author screenshot

The user may click anywhere on the map, which returns the selected location's coordinates to *CreateEventView*. It is important to note that this uses Google's Maps API. An API key is required for this, which means that if the reader runs SportSpark locally on their own device, this part of the app will not work, as the API key is private.

6.2.8. FriendsView and FriendsListView

These views display a user's current friend requests and their confirmed friend requests respectively. The views essentially look the same, just display different collections. The user may navigate to *FriendsListView,* which in turn allows them to navigate to their friends' profiles, where they can see their friends' events, even if those events are outside of their desired event visibility radius. Other than that, the user may decline or accept friend requests on *FriendsView*, which removes the request from the view.

6.2.9. EventDetailsView

This view simply displays information on the event that the user clicked on, either on *HomeView,* or a user's *ProfileView*. The view differs slightly from the Figma UI idea. The final version of this view also contains a *Label* that, when clicked, loads the *map* control once again, displaying the event's location on the platform specific map (Google Maps on Android, for example).

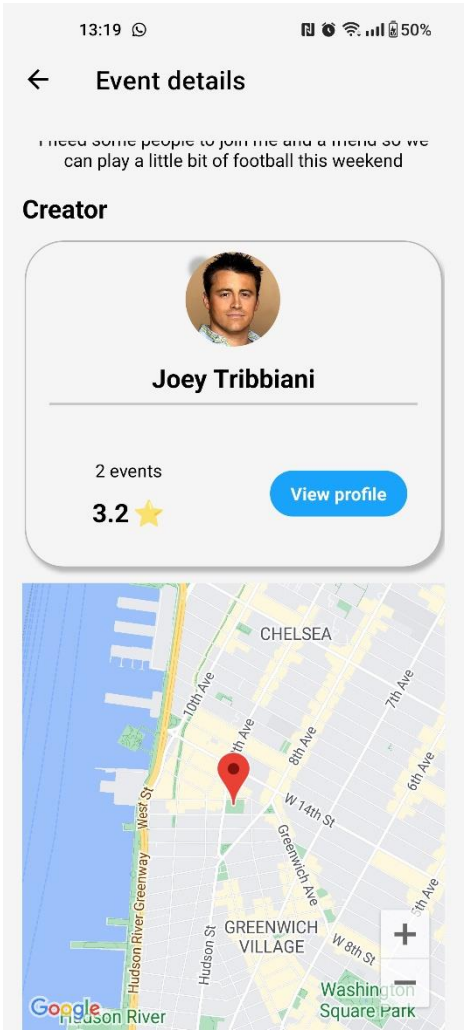


Image 30. EventDetailsView with map control, Source: author screenshot

7. Comparison with Windows version

This chapter will compare the UI appearances between Android and Windows. The *SignInView* will be used as an example. While .NET MAUI is multiplatform, and does allow developers to create applications from a single codebase, the user interface specifically tends to differ a lot between platforms. This is because MAUI converts its view controls to the underlying platform APIs. Furthermore, APIs like the *Map* control that was discussed earlier tend to differ heavily between platforms, and problems may arise. This chapter, however, will only display the difference of a view on Android and Windows, and how the big UI differences may be alleviated using MAUI's *OnPlatform* or *OnIdiom* XAML markup extensions.

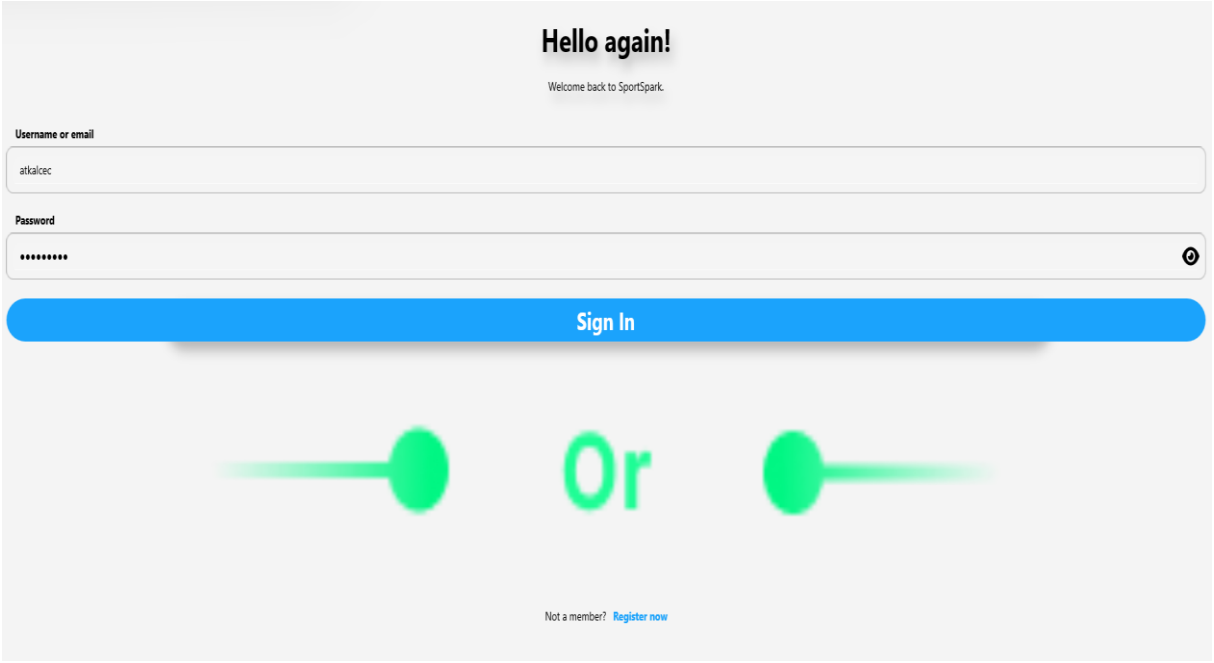


Image 31. SignInView on Windows, Source: author screenshot

Image 30 displays the *SignInView* as it appears on Windows while the app is fullscreen. As one can see, the controls are stretched out, as the XAML states that they should take up all of the horizontal space on the screen, while having a little padding so the text inputs do not stretch to the very borders of the screen. In order to make these inputs smaller, one can use the *OnPlatform* extension on the *VerticalStackLayout* containing the „Username or email“ *Label* and its *Entry* control:

```
...
<VerticalStackLayout.Margin>
      <OnPlatform x:TypeArguments="Thickness">
            <On Platform="WinUI">400, 0, 400, 0</On>
```

```
        </OnPlatform>
</VerticalStackLayout.Margin>
...
```

The resulting UI change looks like this:

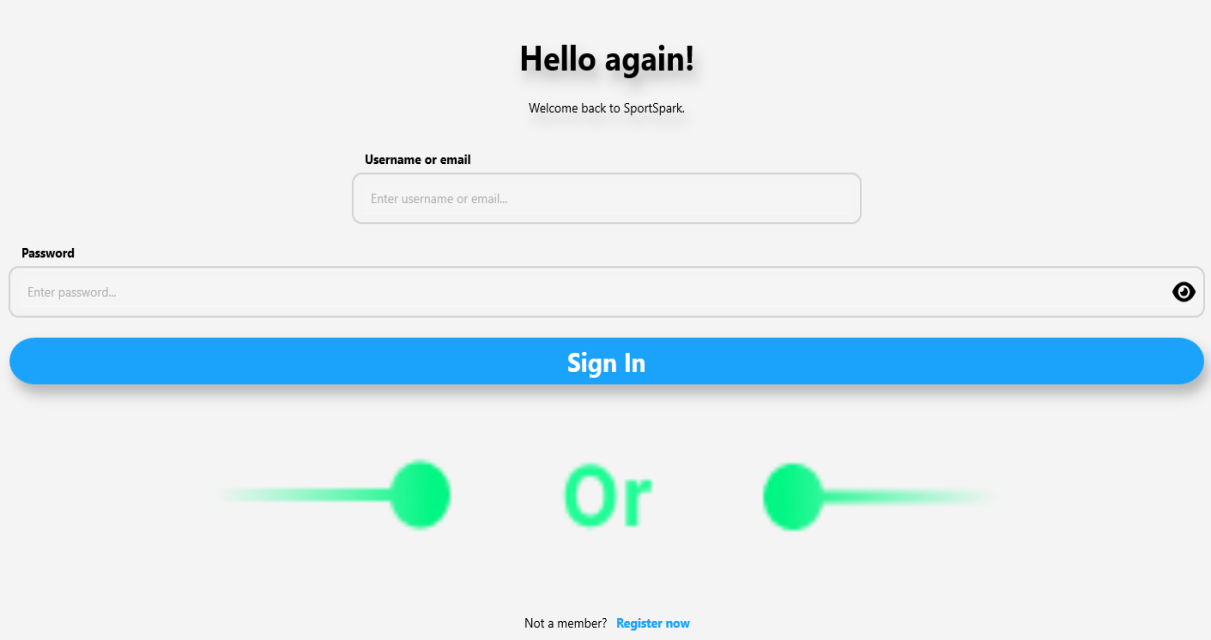


Image 32. SignInView on Windows using OnPlatform, Source: author screenshot

Alternatively, one can use the *OnIdiom* extension, which sets up properties based on the current device, instead of the platform:

```
...
<VerticalStackLayout.Margin>
        <OnIdiom x:TypeArguments="Thickness">
                <OnIdiom.Desktop>
                        <Thickness>400, 0, 400, 0</Thickness>
                </OnIdiom.Desktop>
        </OnIdiom>
</VerticalStackLayout.Margin>
...
```

The resulting UI change looks like this:

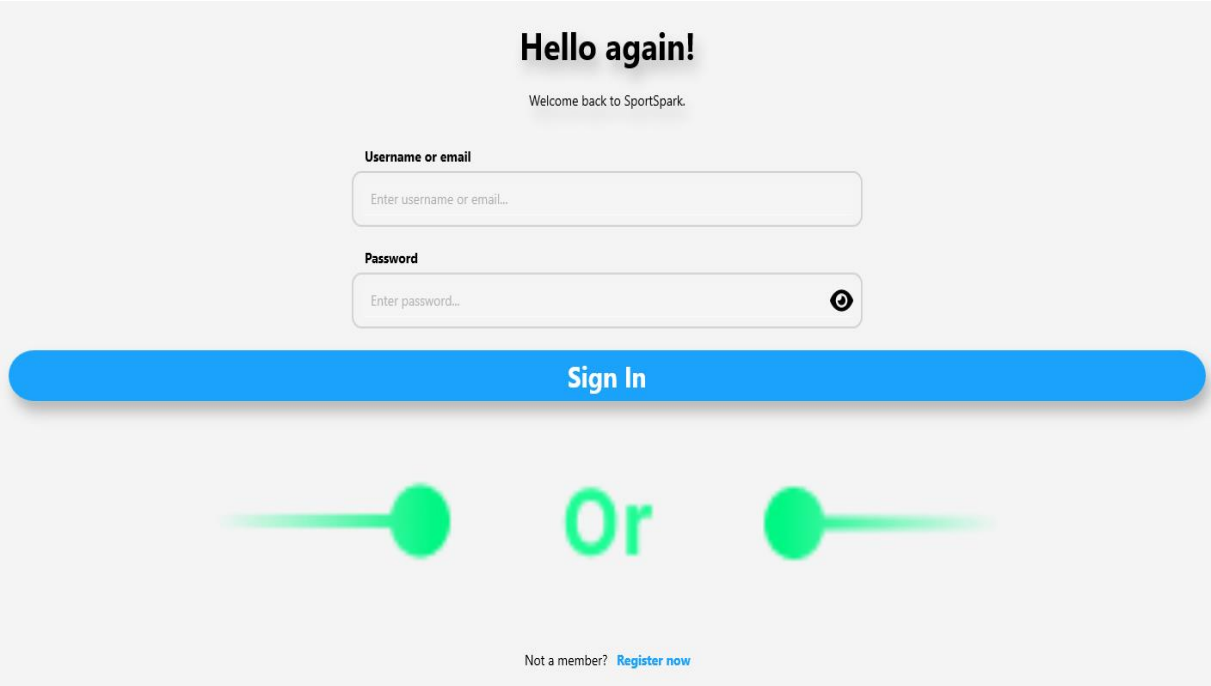


Image 33. SignInView using OnPlatform and OnIdiom extenstions, Source: author screenshot

This is usually how developers may transform the platform or idiom specific UI appearances for each of the controls in the application. Sometimes, however, when working with more specific properties than *margin*, things may get more complicated.

8. Author's opinion on .NET MAUI

.NET MAUI is a relatively new technology, having been released to *general availability* mid-2022. However, it is the author's opinion that the bugs are too plentiful, and some framework design choices poor. The .NET MAUI development team seems quite small, and updates to the framework take too long. On GitHub, there are issues about known bugs that have been active for months on end, some of which are catastrophic.

As one can see on Image 29, the profile picture's shadow is incorrectly rendered. It appears to the image's top-left side and appears as a small circle, while the XAML code specifies it should appear to the image's lower-right side and look as a natural circular shadow. This is not a breaking bug, but would prevent developers from creating the beautiful UI their application deserves.

A catastrophic, breaking bug inside SportSpark, caused by .NET MAUI's *ImageService*, is the app crashing when refreshing the value of a property that an *Image* control's *Source* is bound to. Essentially, an *Image* may have its *Source* property bound to a property in the view model. The first time the *Source* is set up, the image loads correctly. If the property the *Source* is bound to updates, the application will crash. This happens because the same thread should work on any UI changes, but .NET MAUI (at the time of writing) does not update *Image Source* on the main UI thread, instead doing it on a different thread each time. The author has not found a workaround for this, and as such is a breaking bug.

A design choice that the author does not like is sending *QueryProperty* objects between view models. Once *ViewModel1* sends an object to *ViewModel2* through *AppShell's* navigation, *ViewModel2* will „receive“ that object **after** its constructor. This means, for example, a developer may not send a user's *Id* to another view model, which will then call an API to retrieve that user's data. Instead, the developer must use the user's *Id* to call an API and retrieve the user's data inside *ViewModel1*, then send the new object containing the user's data to *ViewModel2*.

Usually, bugs are not catastrophic, and design choices such as this can be worked around. However, due to the number of bugs and the speed, or lack thereof, that they are being fixed, it is the author's opinion that .NET MAUI is currently not the framework developers should use to create multiplatform applications.

9. Conclusion

The purpose of this paper was to present Microsoft's new technology, .NET Multiplatform Application User Interface in the context of a full-fledged Android application with an ASP.NET Core backend. The paper took the reader through the technologies commonly used to create such an app, before presenting an idea around which the application was built. The backend built using ASP.NET Core was briefly described to the user, with more advanced features such as JWT Authentication receiving more attention. Then, the specifics of MAUI as a framework were explained, as were certain NuGet packages associated with some of SportSpark's features. The user interface differences that arise between platforms were displayed, as was the way to mitigate them. Finally, the author reviewed .NET MAUI as a technology, concluding that it is not the framework of choice at this time when it comes to building multiplatform applications.

Hopefully, this paper would be useful to any readers who may have an interest in multiplatform application development, Microsoft technologies, or both. The entirety of SportSpark's code is present on GitHub, the link for which will be placed under attachments.

Sources

[1] “NET Multi-platform App UI (.NET MAUI) | .NET,” *Microsoft*, Mar. 05, 2023. https://dotnet.microsoft.com/en-us/apps/maui (accessed Mar. 05, 2023).

[2] “Visual Studio: IDE and Code Editor for Software Developers and Teams,” *Visual Studio*. https://visualstudio.microsoft.com (accessed May 29, 2023).

[3] erinstellato-ms, “Download SQL Server Management Studio (SSMS) - SQL Server Management Studio (SSMS),” May 24, 2023. https://learn.microsoft.com/en-us/sql/ssms/download-sql-server-management-studio-ssms (accessed May 29, 2023).

[4] “Git.” https://git-scm.com/ (accessed May 29, 2023).

[5] Atlassian, “Sourcetree | Free Git GUI for Mac and Windows,” *SourceTree*. https://www.sourcetreeapp.com (accessed May 29, 2023).

[6] “Android Debug Bridge (adb) | Android Studio,” *Android Developers*. https://developer.android.com/tools/adb (accessed May 29, 2023).

[7] tdykstra, “Overview of ASP.NET Core,” Nov. 15, 2022. https://learn.microsoft.com/en-us/aspnet/core/introduction-to-aspnet-core (accessed May 30, 2023).

[8] ajcvickers, “Overview of Entity Framework Core - EF Core,” May 25, 2021. https://learn.microsoft.com/en-us/ef/core/ (accessed May 30, 2023).

[9] “Entity Framework Core DbContext.” https://www.entityframeworktutorial.net/efcore/entity-framework-core-dbcontext.aspx (accessed May 30, 2023).

[10] “Free, Online UI Design Tool & Software For Teams,” *Figma*. https://www.figma.com/ui-design-tool/ (accessed May 30, 2023).

[11] maddymontaquila, “XAML overview - Visual Studio (Windows),” Mar. 10, 2023. https://learn.microsoft.com/en-us/visualstudio/xaml-tools/xaml-overview (accessed May 30, 2023).

[12] davidbritch, “What is .NET MAUI? - .NET MAUI,” Jan. 30, 2023. https://learn.microsoft.com/en-us/dotnet/maui/what-is-maui (accessed May 30, 2023).

[13] ardalis, “Common web application architectures,” Mar. 07, 2023. https://learn.microsoft.com/en-us/dotnet/architecture/modern-web-apps-azure/common-web-application-architectures (accessed Jun. 01, 2023).

[14] “Repository Pattern in ASP.NET Core - Ultimate Guide,” Jun. 28, 2020. https://codewithmukesh.com/blog/repository-pattern-in-aspnet-core/ (accessed Jun. 10, 2023).

[15] “Data Transfer Object DTO Definition and Usage | Okta.” https://www.okta.com/identity-101/dto/ (accessed Jun. 10, 2023).

[16] michaelstonis, “Model-View-ViewModel,” Nov. 04, 2022. https://learn.microsoft.com/en-us/dotnet/architecture/maui/mvvm (accessed Jun. 10, 2023).
[17] bricelam, “Migrations Overview - EF Core,” Jan. 12, 2023. https://learn.microsoft.com/en-us/ef/core/managing-schemas/migrations/ (accessed Jun. 10, 2023).
[18] auth0.com, “JWT.IO - JSON Web Tokens Introduction.” http://jwt.io/ (accessed Jun. 10, 2023).

Images

Attachments

SportSpark GitHub repository:
https://github.com/AntunTkalcec/SportSpark

Github issues (mentioned bugs):
https://github.com/dotnet/maui/issues/14786, https://github.com/dotnet/maui/issues/14052