UNIVERSITY OF ZAGREB

FACULTY OF ORGANIZATION AND INFORMATICS

V A R A Ž D I N

**Antun Tkalčec**

Github issues: <https://github.com/dotnet/maui/issues/14786>, https://github.com/dotnet/maui/issues/14052

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. The paper first lists and describes tools and technologies that are used in software development, before detailing the steps

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

Opsega od 100 do 300 riječi. Sažetak upućuje na temu rada, ukratko se iznosi čime se rad bavi, teorijsko-metodološka polazišta, glavne teze i smjer rada te zaključci.

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

Content

[Content iii](#_Toc137299602)

[1. Introduction 1](#_Toc137299603)

[2. Tools and technologies 2](#_Toc137299604)

[2.1. Microsoft Visual Studio 2](#_Toc137299605)

[2.2. Microsoft SQL Server 3](#_Toc137299606)

[2.3. Microsoft SQL Server Management Studio (SSMS) 4](#_Toc137299607)

[2.4. Git 4](#_Toc137299608)

[2.5. Sourcetree 5](#_Toc137299609)

[2.6. adb 6](#_Toc137299610)

[2.7. Backend tools and technologies 7](#_Toc137299611)

[2.7.1. ASP.NET Core 7](#_Toc137299612)

[2.7.2. Entity Framework Core 7 7](#_Toc137299613)

[2.8. Frontend tools and technologies 9](#_Toc137299614)

[2.8.1. Figma 9](#_Toc137299615)

[2.8.2. XAML 10](#_Toc137299616)

[2.8.3. .NET MAUI 12](#_Toc137299617)

[3. Application idea 13](#_Toc137299618)

[4. Best practices, patterns and important terms 15](#_Toc137299619)

[4.1. Clean Architecture 15](#_Toc137299620)

[4.2. Repository pattern 17](#_Toc137299621)

[4.3. Data Transfer Objects 17](#_Toc137299622)

[4.4. Model-View-ViewModel-Service pattern 18](#_Toc137299623)

[5. Backend development 19](#_Toc137299624)

[5.1. Database modeling 19](#_Toc137299625)

[5.2. Creating a database 20](#_Toc137299626)

[5.3. Creating an API 24](#_Toc137299627)

[5.3.1. JWT Authentication 27](#_Toc137299628)

[5.3.2. Handling HTTP requests 29](#_Toc137299629)

[5.3.2.1. Error handling 32](#_Toc137299630)

[6. Frontend development 33](#_Toc137299631)

[6.1. Designing a user interface 33](#_Toc137299632)

[6.2. Building a user interface 33](#_Toc137299633)

[7. Placeholder 2 34](#_Toc137299634)

[8. Placeholder 3 35](#_Toc137299635)

[9. Placeholder 4 35](#_Toc137299636)

[9.1. Poglavlje druge razine 35](#_Toc137299637)

[9.1.1. Poglavlje treće razine 35](#_Toc137299638)

[9.1.1.1. Poglavlje četvrte razine 35](#_Toc137299639)

[10. Tehničke upute 37](#_Toc137299640)

[10.1. Upute za oblikovanje izgleda rada 37](#_Toc137299641)

[10.2. Navođenje literature 40](#_Toc137299642)

[11. Conclusion 41](#_Toc137299643)

[Literature 42](#_Toc137299644)

[Images 43](#_Toc137299645)

[Tables 44](#_Toc137299646)

[Attachments (1, 2, …) 45](#_Toc137299647)

1. Introduction

When developing an application, a lot of time is spent adapting the business logic and the user interface (UI) to each platform – such as Android, iOS, Windows, Web and macOS. In situations where the developer or 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 used.

.NET MAUI stands for .NET Multiplatform Application User Interface, and it allows the developer to seamlessly target devices of all platforms. MAUI abstracts the latest technologies for building native apps on popular platforms into one common framework [1]. 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, starting by designing the database model, building said 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. However, 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 it's 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.

1. Tools and technologies

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

* 1. Microsoft Visual Studio

All of the code for SportSpark'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

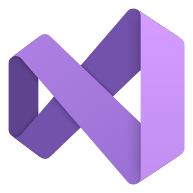


Image 1. Microsoft Visual Studio logo, Source: <https://commons.wikimedia.org/wiki/File:Visual_Studio_Icon_2022.svg>

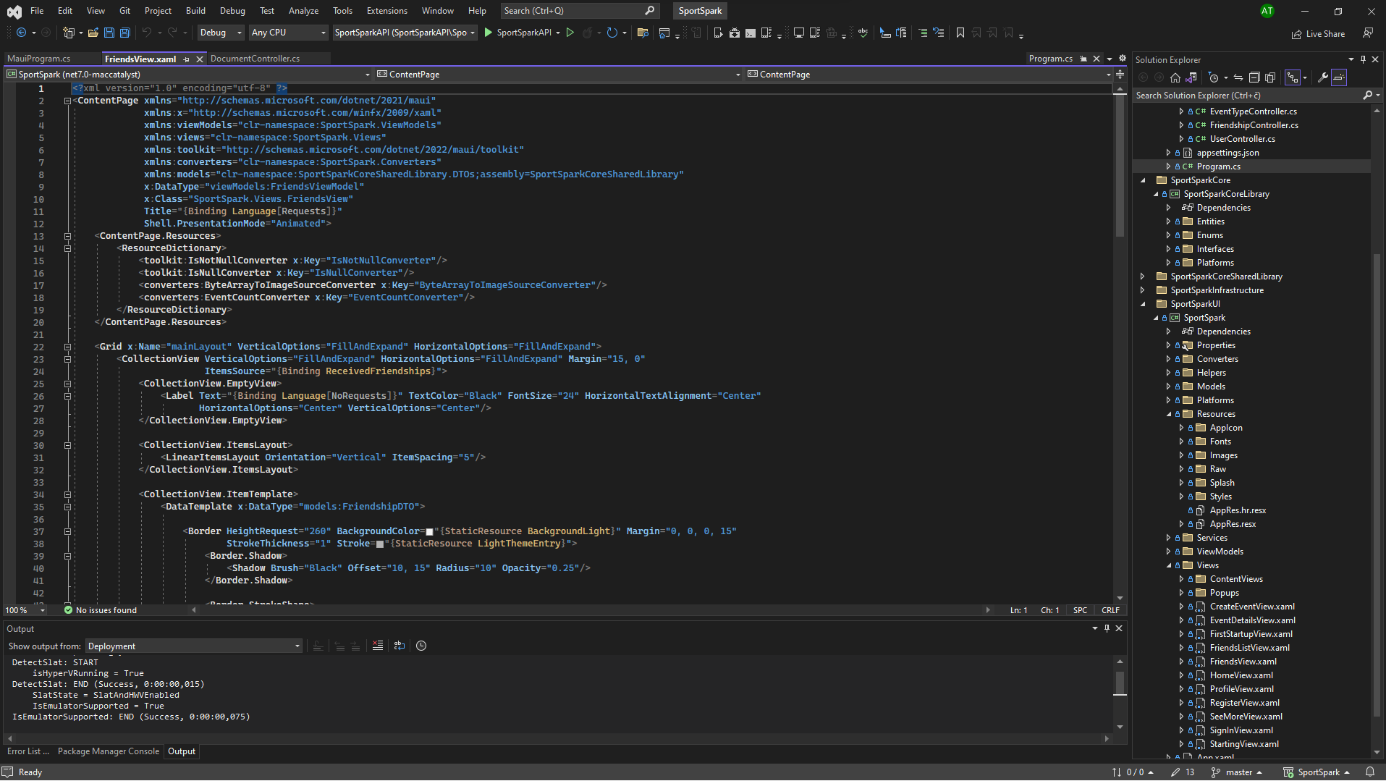


Image 2. Microsoft Visual Studio's User Interface with tabs open, Source: author screenshot

* 1. Microsoft SQL Server

The database for SportSpark 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.

A picture containing sketch, drawing, graphics, diagram

Description automatically generated

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

* 1. 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. Every single thing to do with this application will be done locally.

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

A picture containing design, illustration

Description automatically generated with medium confidence

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

* 1. 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 SportSpark is created by one developer on a single machine, Git will still be used to *commit* and *push* code changes to a GitHub repository. All SportSpark 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].

A picture containing graphics, font, symbol, design

Description automatically generated

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

* 1. 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>

A screenshot of a computer program

Description automatically generated with medium confidence

Image 7. Sourcetree User Interface, Source: author screenshot

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

* 1. adb

'adb' or Android Debug Bridge „is a versatile command-line tool that lets you communicate with a device.“ [6] Given that SportSpark is 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 SportSpark's frontend.

* 1. 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.

* + 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 SportSpark using ASP.NET Core.

A blue and white logo

Description automatically generated with low confidence

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

* + 1. 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.

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.

A purple square with white text and blue circle

Description automatically generated with low confidence

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

SportSpark'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 of SportSpark, 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, *SportSparkDBContext* inherits from *DbContext*, and one of it's 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.

* 1. Frontend tools and technologies

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 SportSpark, mockups will be created using Figma. XAML and .NET MAUI itself will also be described in the following subchapters.

* + 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]

A picture containing graphics, graphic design, colorfulness, screenshot

Description automatically generated

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

For SportSpark, 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.

* + 1. 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, while the underlying functionality and behavior are implemented in the code-behind using C# or another language. 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 SportSpark'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, while designing a complex interface quickly becomes quite complicated. The following code is an example of a, relatively to other views, simple view inside of SportSpark'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 SportSpark reach over 200 lines of XAML code.

* + 1. .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 SportSpark.

1. 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 sport 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 their desired radius in which they wish to see them. Also, a notification could be sent to a user if one of their 'friends' creates an event that the user is interested in. However, notifications are out of scope for this paper, as the aim is to merely present .NET technologies.

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 – will the event take place again, when, where, how often
* 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
* Number of participants – how many people may sign up for the event or how many people the event needs
* 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 will 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** should make sense as the application's name.

1. 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.

* 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].

A picture containing text, screenshot, circle, diagram

Description automatically generated

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.

A screenshot of a computer

Description automatically generated

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.

A screenshot of a computer

Description automatically generated

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

Inside the projects, one can see folders for aforementioned Interfaces, Entities, Services and Repositores. 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.

* 1. 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 are 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).

* 1. 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.

* 1. 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.

1. 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.

* 1. Database modeling

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

A screenshot of a computer screen

Description automatically generated with low confidence

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

A screenshot of a computer program

Description automatically generated with low confidence

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

* 1. 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("LocalConnection"));

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

...

"ConnectionStrings": {

"LocalConnection": "Server=.;Database=SportSpark;Trusted\_Connection=True;MultipleActiveResultSets=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 progress. They may simply change an Entity as required, create a new Migration, and update the database.

* 1. 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 „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.

* + 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, we can see it's 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.

* + 1. 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, because 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 it's 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 -> 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.

* + - 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.

1. Frontend development
   1. Designing a user interface
   2. Building a user interface

placeholder

1. Placeholder 2

placeholder

1. Placeholder 3
2. Placeholder 4

Ovo je glavni dio rada u kojem treba razraditi temu, pojasniti istraživanja, prikazati rezultate i slično. Poželjno je na početku poglavlja dati kratki opis strukture poglavlja, kako bi čitatelj/čitateljica rada mogao/mogla lakše pratiti složenu cjelinu.

* 1. Poglavlje druge razine

Mauris vel lacus vitae felis vestibulum volutpat. Etiam est nunc, venenatis in, tristique eu, imperdiet ac, nisl. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. In iaculis facilisis massa. Etiam eu urna. Sed porta. Suspendisse quam leo, molestie sed, luctus quis, feugiat in, pede. Fusce tellus. Sed metus augue, convallis et, vehicula ut, pulvinar eu, ante. Integer orci tellus, tristique vitae, consequat nec, porta vel, lectus. Nulla sit amet diam. Duis non nunc. Nulla rhoncus dictum metus. Curabitur tristique mi condimentum orci. Phasellus pellentesque aliquam enim. Proin dui lectus, cursus eu, mattis laoreet, viverra sit amet, quam. Curabitur vel dolor ultrices ipsum dictum tristique. Praesent vitae lacus. Ut velit enim, vestibulum non, fermentum nec, hendrerit quis, leo. Pellentesque rutrum malesuada neque.

* + 1. Poglavlje treće razine

Nunc tempus felis vitae urna. Vivamus porttitor, neque at volutpat rutrum, purus nisi eleifend libero, a tempus libero lectus feugiat felis. Morbi diam mauris, viverra in, gravida eu, mattis in, ante. Morbi eget arcu. Morbi porta, libero id ullamcorper nonummy, nibh ligula pulvinar metus, eget consectetuer augue nisi quis lacus. Ut ac mi quis lacus mollis aliquam. Curabitur iaculis tempus eros. Curabitur vel mi sit amet magna malesuada ultrices. Ut nisi erat, fermentum vel, congue id, euismod in, elit. Fusce ultricies, orci ac feugiat suscipit, leo massa sodales velit, et scelerisque mi tortor at ipsum. Proin orci odio, commodo ac, gravida non, tristique vel, tellus. Pellentesque nibh libero, ultricies eu, sagittis non, mollis sed, justo. Praesent metus ipsum, pulvinar pulvinar, porta id, fringilla at, est.

* + - 1. Poglavlje četvrte razine

Phasellus felis dolor, scelerisque a, tempus eget, lobortis id, libero. Donec scelerisque leo ac risus. Praesent sit amet est. In dictum, dolor eu dictum porttitor, enim felis viverra mi, eget luctus massa purus quis odio. Etiam nulla massa, pharetra facilisis, volutpat in, imperdiet sit amet, sem. Aliquam nec erat at purus cursus interdum. Vestibulum ligula augue, bibendum accumsan, vestibulum ut, commodo a, mi. Morbi ornare gravida elit. Integer congue, augue et malesuada iaculis, ipsum dui aliquet felis, at cursus magna nisl nec elit. Donec iaculis diam a nisi accumsan viverra. Duis sed tellus et tortor vestibulum gravida. Praesent elementum elit at tellus. Curabitur metus ipsum, luctus eu, malesuada ut, tincidunt sed, diam. Donec quis mi sed magna hendrerit accumsan:

* Suspendisse risus nibh
* ultricies eu
* volutpat non
* condimentum hendrerit
* augue.

Etiam eleifend, metus vitae adipiscing semper, mauris ipsum iaculis elit, congue gravida elit mi egestas orci. Curabitur pede. Maecenas aliquet velit vel turpis. Mauris neque metus, malesuada nec, ultricies sit amet, porttitor mattis, enim. In massa libero, interdum nec, interdum vel, blandit sed, nulla. In ullamcorper, est eget tempor cursus, neque mi consectetuer mi, a ultricies massa est sed nisl. Class aptent taciti sociosqu ad litora torquent per conubia nostra, per inceptos hymenaeos. Proin nulla arcu, nonummy luctus, dictum eget, fermentum et, lorem. Nunc porta convallis pede.

1. Tehničke upute

Tehničke upute u nastavku opisuju način tehničkog oblikovanja rada i navođenja literature.

* 1. Upute za oblikovanje izgleda rada

**Stranice** se oblikuju korištenjem sljedećih parametara:

* veličina i oblik papira je A4, okomito usmjerenje, margine 2,5 cm na svakoj strani;
* naslovna stranica rada se ne numerira;
* nakon naslovne stranice, sve sljedeće stranice do 1. Poglavlja se numeriraju rimskim brojevima, počevši od i;
* od 1. poglavlja nadalje, stranice se numeriraju arapskim brojevima;
* broj stranice treba pozicionirati desno 1,25 cm od dna stranice, font Arial 9.

**Tekst** rada je potrebno oblikovati sukladno ovom predlošku, odnosno na sljedeći način:

* u pisanju teksta koristite font Arial 11 pt, s proredom 1,5 te razmakom 0 pt prije i razmakom 6 pt poslije odlomka, pri čemu je prvi redak uvučen za 1,25 cm;
* u naslovima prve razine „3. Razrada teme“ koristite font Arial 18 pt, podebljano, prijelom stranice (svaki naslov prve razine treba biti na novoj stranici), s proredom 1,5 te razmakom 0 pt prije i razmakom 18 pt poslije odlomka;
* u naslovima druge razine „2.1. Naslov“ koristite font Arial 16 pt, podebljano, s proredom 1,5 te razmakom 18 pt prije i razmakom 12 pt poslije odlomka;
* u naslovima treće razine „2.1.1. Naslov“ koristite font Arial 14 pt, podebljano, s proredom 1,5 te razmakom 12 pt prije i razmakom 6 pt poslije odlomka;
* u naslovima četvrte razine „2.1.1.1. Naslov“ koristite font Arial 12 pt, podebljano, s proredom 1,5 te razmakom 6 pt prije i razmakom 6 pt poslije odlomka;
* ostalo značajno isticanje cjelina rada može biti istaknuto podebljanim i kurziv slovima, korištenjem fonta Arial 11 pt.

**Slike** u radu je potrebno oblikovati odnosno na sljedeći način:

* naziv slike navedite ispod slike uz numeraciju;
* za nazive slika koristite iste postavke fonta kao i za tekst, ali stavite naziv slike u centrirani položaj;
* za oblikovanje same slike koristite font Arial 9 pt za tekst na slici;
* ispred same slike umetnite jedan prazan redak (osim ako je slika pozicionirana na početku stranice);
* nakon naziva slike ostavite jedan redak prazan (osim ako je naziv slike zadnji redak na stranici);
* kod prijeloma stranice treba obratiti posebnu pozornost da naziv slike, izvor i sama slika moraju biti na istoj stranici;
* slike je potrebno numerirati redom pojavljivanja u tekstu;
* ako je slika preuzeta iz drugog izvora, nakon navođenja naziva slike u zagradi navedite izvor, npr. (autor/autorica, godina);
* dozvoljeno je napraviti vlastitu preradu slika, grafikona ili tablica na način da se zadrži isti smisao sadržaja, ali promijeni izgled. I u takvim se slučajevima obavezno u nazivu navodi referenca izvornog djela ovako: “(Prema: Klačmer Čalopa i Cingula, 2012)“;
* dozvoljeno je preuzeti samo jednu sliku, grafikon ili tablicu u izvornom obliku iz istog izvora. Za doslovno preuzimanje većeg dijela sadržaja potrebno je ishoditi dozvolu nositelja autorskih prava;
* primjer označavanja slike možete vidjeti u nastavku.



Slika 1: Podjela investicijskih fondova (Izvor: Klačmer Čalopa i Cingula, 2012)

**Tablice** rada je potrebno oblikovati sukladno ovim uputama:

* naziv tablice navedite iznad slike;
* za nazive tablica koristite iste postavke fonta kao i za tekst, ali stavite naziv tablice u centrirani položaj;
* za oblikovanje same tablice koristite font Arial 9 pt za tekst u tablici;
* tablice numerirajte redom pojavljivanja u tekstu;
* prije naziva tablice umetnite jedan redak prazan (osim ako je naziv tablice prvi redak na stranici);
* nakon same tablice umetnite jedan prazan redak (osim ako je tablica pozicionirana na kraju stranice);
* kod prijeloma stranice treba obratiti posebnu pozornost da naziv tablice, izvor i sama tablica moraju biti na istoj stranici;
* ako je tablica preuzeta iz drugog izvora, nakon navođenja naziva tablice potrebno je navesti izvor, na isti način kako je opisano kod slika;
* primjer označavanja tablice možete vidjeti u nastavku.

Tablica 1: Prikaz podataka o učestalosti pojavljivanja objekta

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

(Izvor: Klačmer Čalopa i Cingula, 2012)

**Programski kod**

* za oblikovanje teksta koji je programski kôd koristite font Courier, veličine 10 pt, jednostruki prored, 6 pt iza odlomka, npr. HTML kôd dijela zaglavlja početne web stranice FOI weba:

<head>

<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />

<link rel="shortcut icon" href="[https://www.foi.unizg.hr/sites/default/files/favicon\_0\_1.ico](view-source:https://www.foi.unizg.hr/sites/default/files/favicon_0_1.ico)" type="image/vnd.microsoft.icon" />

<meta name="generator" content="Drupal 7 (http://drupal.org)" />

<link rel="canonical" href="[https://www.foi.unizg.hr/hr](view-source:https://www.foi.unizg.hr/hr)" />

<link rel="shortlink" href="[https://www.foi.unizg.hr/hr](view-source:https://www.foi.unizg.hr/hr)" />

<!-- Set the viewport width to device width for mobile -->

<meta name="viewport" content="width=device-width, initial-scale=1.0">

<title>Dobro došli na FOI | FOI</title>...

</head>

**Formule**

* za unos formula koristite editor za formule u svom tekst procesoru.

**Kratice**

* ako želite koristiti kratice pojmova u tekstu, kad prvi put spominjete pojam potrebno je navesti puni naziv, a kraticu navesti u zagradi (npr. Informacijske i komunikacijske tehnologije, kraće IKT). Nakon toga možete koristiti kratice u tekstu. Poželjno je u naslovima koristiti pune nazive.

**Strano nazivlje**

* strano nazivlje se u tekstu navodi u zagradi, napisano *kurzivom*, nakon hrvatskog izraza, npr. Analiza društvene mreže (eng. *Social Network Analysis - SNA*).
  1. Navođenje literature

Za navođenje literature u radu možete odabrati i koristiti jedan od sljedeća dva ponuđena stila: APA ili IEEE stil. Važno je dosljedno primjenjivati odabrani stil u cijelom radu.

U popisu literature potrebno je navesti svu literaturu i samo literaturu koju ste koristili u tekstu.

Uz svaku preuzetu tvrdnju potrebno je navesti njezin izvor, tj. referencu. Reference se u tekstu navode tako da se uz citirani tekst navede izvor, sukladno načinu propisanom odabranim stilom i FOI preporukama za citiranje i referenciranje.

1. Conclusion

Ovdje treba sažeto rezimirati najvažnije rezultate razrade teme rada. Potrebno je sažeto opisati što je predmet rada, koje su metode, tehnike, programski alati ili aplikacije korištene u razradi rada te koje su pretpostavke dokazane, a koje opovrgnute. Sadržajno, ono što se u uvodu rada najavljuje i kasnije je obuhvaćeno u samom radu, moralo bi biti opisano u zaključnom dijelu kroz rezultate rada.

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Nam cursus. Morbi ut mi. Nullam enim leo, egestas id, condimentum at, laoreet mattis, massa. Sed eleifend nonummy diam. Praesent mauris ante, elementum et, bibendum at, posuere sit amet, nibh. Duis tincidunt lectus quis dui viverra vestibulum. Suspendisse vulputate aliquam dui. Nulla elementum dui ut augue. Aliquam vehicula mi at mauris. Maecenas placerat, nisl at consequat rhoncus, sem nunc gravida justo, quis eleifend arcu velit quis lacus. Morbi magna magna, tincidunt a, mattis non, imperdiet vitae, tellus. Sed odio est, auctor ac, sollicitudin in, consequat vitae, orci. Fusce id felis. Vivamus sollicitudin metus eget eros.

Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. In posuere felis nec tortor. Pellentesque faucibus. Ut accumsan ultricies elit. Maecenas at justo id velit placerat molestie. Donec dictum lectus non odio. Cras a ante vitae enim iaculis aliquam. Mauris nunc quam, venenatis nec, euismod sit amet, egestas placerat, est. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Cras id elit. Integer quis urna. Ut ante enim, dapibus malesuada, fringilla eu, condimentum quis, tellus. Aenean porttitor eros vel dolor. Donec convallis pede venenatis nibh. Duis quam. Nam eget lacus. Aliquam erat volutpat. Quisque dignissim congue leo.

Literature

Popis literature treba biti izrađen u skladu s odabranim stilom navođenja literature (APA ili IEEE stilom), a poželjno je korištenje prikladnog programskog alata (npr. Zotero). Primjeri navođenja literature opisani su u dokumentima „FOI: preporuke citiranja i referenciranja primjenom stila referenciranja APA“ i „FOI: preporuke citiranja i referenciranja primjenom stila referenciranja IEEE“.

Images

Popis slika treba biti izrađen po uzoru na indeksirani sadržaj, te upućivati na broj stranice na kojoj se slika može pronaći.

[Slika 1: Podjela investicijskih fondova 6](#_Toc496692358)

Tables

Popis tablica treba biti izrađen po uzoru na indeksirani sadržaj, te upućivati na broj stranice na kojoj se tablica može pronaći.

[Tablica 1: Prikaz podataka o učestalosti pojavljivanja objekta 7](#_Toc496692416)

Attachments (1, 2, …)