- 2. Theoretischer Hintergrund / Grundlagen und Definitionen

  - 2.1 Grundbegriffe / Forschungsstand
  - 2.2 Technologien / Instrumente
    - 2.2.1 ASP.NET Boilerplate
    - 2.2.2 ASP.NET Core
    - 2.2.3 ASP.NET Core MVC
    - 2.2.4 Entity Framework Core
    - 2.2.5 ML.NET

2. Theoretical concepts

In the current day and age, many ordinary aspects of one's life have already been or are currently being ported into the online realm, with a person's dependency on internet access becoming more of a basic human need, rather than a whim or a simple want. Such is the case of an activity which can either be a leisure activity or perhaps a passion that can turn into a career such as karting.

Much like any other business, the Romanian Karting Federation FRK has its own website where one can find information about official events, rules and regulations and so on. However, what the currently existent solution lacks is exactly the core aspect of the "old way" of doing things, interaction and straightforwardness. There is not much more to do other than browse through the available pages and read the available information, there is no way to directly contact event organizers, no way to contact karting academies or access an official market that can offer exactly what one needs, it simply lacks humanity.

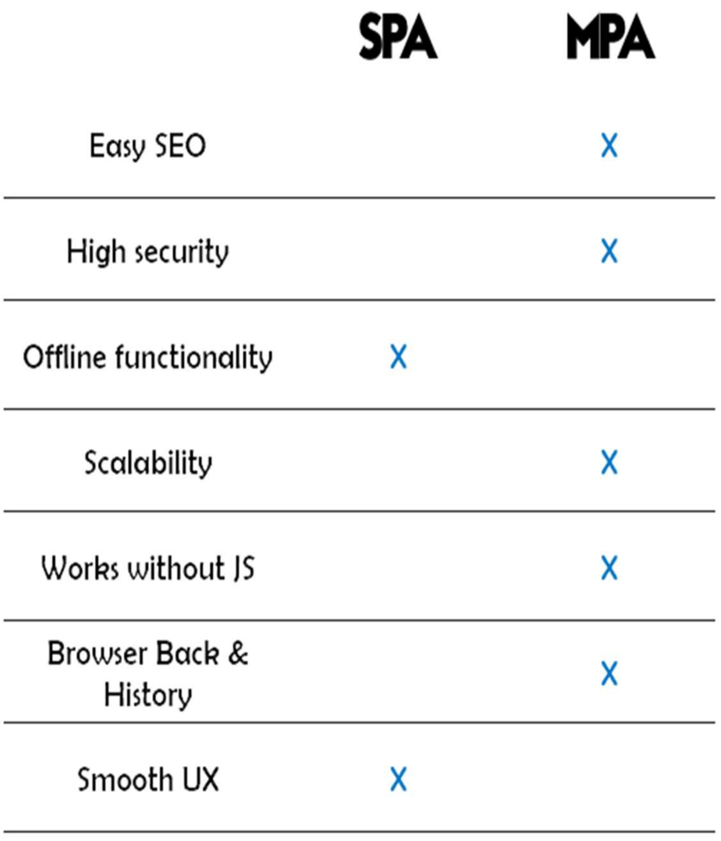
These are exactly the issues that the solution proposed in this paper aims to solve. More specifically, turning the website that is currently available into a web application which allows its users to directly communicate with equipment sellers, event organizers and administrators and will enable them to begin their journey into the karting world with little to no effort whatsoever.

2.1 Basic concepts and state of the art

Multi-Page Web Applications are traditional applications that depend on the pages refreshing to update the data that is displayed to the user and to communicate with the server. These types of applications are suited for large web platforms with multiple, complex and different views. It is a classic architectural approach that has proven to be dependable, easily scalable and simpler in terms of technology stacks.

As technology advances and more modern approaches to developing web applications become more and more popular, the diversity of architectural structures also grows proportionally. Nowadays, Single-Page Applications are hastily gaining traction as more popular choices for different types of applications and customer needs. However, SPAs have only been around for a little more than a decade now and, although their issues and cons have started to be solved as time goes on, this is a work in progress.

In response to the aforementioned SPAs comes the traditional and stable MPA approach. This application type is highly scalable, featuring multiple page types and more a more complex structure, can easily allow the use of the browser-based *Backwards* button and browsing history since navigating from page to page calls for changes in the URL path, thus allowing the user to take advantage of all the incorporated features of the browser. Moreover, this type of behaviour allows for better and easier SEO optimisation, since crawlers can more efficiently check page contents compared to SPA pages which require JavaScript to load the pages and therefore making such integrations more difficult.



	SPA	MPA
Easy SEO		X
High security		X
Offline functionality	X	
Scalability		X
Works without JS		X
Browser Back & History		X
Smooth UX	X	

Figure 1 – Single-Page Applications compared to traditional Multi-Page Applications

In terms of security, it definitely requires more time and effort to secure an MPA completely and efficiently with multiple pages since every single page needs to be secured, but safety is nevertheless much greater due to the heavy dependence on JavaScript that SPAs have, making them more prone to attacks and proving an easy target for cross-site scripting. SPAs also require APIs to be publicly exposed, thus creating the need for the endpoints to also be secured properly. Moreover, compared to SPAs, MPA pages are "*short lived*" and don't create the problematic issue of memory leaks that the long lifecycles of SPA-based platforms encounter. Therefore, as far as data safety and integrity is concerned, MPAs feature increased security, however at the cost of more time and effort.

Performance-wise, MPAs take more time and resources to retrieve data from the server and present to the user since each action represents a request and retrieves the page from the server, therefore also feeling more rigid and not providing the user with the smooth experience an SPA is able to. This is a by-product of different page lifecycles, as each approach features different particularities that suit their individual needs. Both of these start their lifecycle with an initial page request from the client that reaches the server, which in turn returns HTML, JavaScript and CSS content to the client as a response. However, this is the point where the two begin to differ in approach. MPAs submit a POST or GET request following a user action which results in the client having to reload the page to update the HTML content of the page. This, however, is where the modern aspect of SPAs comes into light more visibly. The modern approach is to use AJAX to make requests to the server and, instead of having to get whole HTML pages from the server, the client receives JSON or other data formats which it uses to update the user's view, eliminating the need for the page to reload and only updating certain fragments of the page. This technique proves more efficient, faster and provides the user a smoother transition between changing states. One of the drawbacks of SPAs is that the initial, first load is very slow. This is caused by client-side rendering, because as efficient as SPAs are, their performance is ensured by bulky JavaScript frameworks.

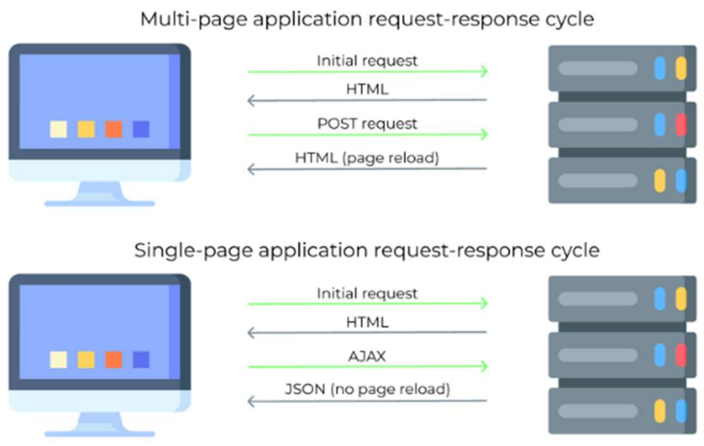


Figure 2 - Page lifecycle differences between traditional MPAs and modern SPAs

2.2 Technologies

In order to ensure ease of scalability, use and maintenance, the application framework used for the application this paper features is ASP.NET Boilerplate [1], which is an open-source application framework encompassing multiple tools and making use of popular good-practices. The technology stack used in the application features ASP.NET Core for its foundation, uses Razor Pages for the presentation layer and Entity Framework for the Infrastructure Layer managing a SQL-Server-based database.

2.2.1 ASP.NET Boilerplate

One of the key aspects of ABP is that it provides several useful and important features for the developer to efficiently and more methodically develop an application. Some of these integrated features are Dependency Injection, Default Repositories, Permissions, Unit of Work, Localization, Auto Mapping and others.

Dependency Injection is a technique which allows for one or more dependencies to be injected, removing the creation from the dependent object and therefore removing a hard dependency through the use of Constructor Injection for example. Managing these dependencies would be tedious if not for the way that ABP uses dependency injection frameworks, registering dependencies conventionally using transient lifestyles. Of course, there are also simple ways of registering dependencies directly should conventional registrations not be sufficient.

Default Repositories in ABP are used to perform CRUD operations for Entities, using a separate repository for each Entity. Through integration with Entity Framework, entering a repository method automatically opens a transaction which is either committed, if the method ends and returns, or rolled back, if the method throws any type of Exception. This type of behaviour is accommodated by using a Unit of Work. All Repository, Application Service and API Controller actions are by default unit of work and are, therefore, atomic. Since they are heavily used in Dependency Injection, every repository instance has a transient lifestyle.

Since a multi-tenant application frequently needs different authorization levels, this can easily be accomplished thanks to the Permissions features integrated in the framework. Permissions can be enforced not only for API endpoints, but also for views, thanks to the versatility of Razor Views, and even in client-side JavaScript through the ABP namespace.

A Unit of Work, as described briefly above, when talking about Default Repositories, is a mechanism that allows for transactional behaviour to be enforced in an application which uses a database. Methods which are managed through the unit of work system are called atomic and opening, managing and closing the connection to the database is handled automatically by the implemented system. Should one atomic method call another atomic method, they both use the same transaction, managed by the first entered method. Naturally, even though some methods are by default classified as unit of work, there is also the option of explicitly controlling the unit of work. Moreover, a non-transactional unit of work can also be explicitly used, if locking rows or tables in the database proves problematic for a developer's implementation. Saving changes can also be done automatically or explicitly, depending on the use and specific needs of the application.

Since, more often than not, applications tend to be suited not only to one culture but to a larger number of cultures, ABP features integrated Localization. Localization Sources can be accessed simply by calling the L method, after which the system, based on the user's current culture, establishes the correct localized text to suit the user. This is done either through ASP.NET Core default providers or custom, ABP defined providers. This method allows for usage in server-side code, controllers, views, and also JavaScript code.

In order to efficiently isolate the Presentation Layer from the Domain Layer efficiently, Data Transfer Objects are used to call an Application Service that in turn uses these domain objects for specific operations and returns it back. Although DTOs might seem tedious and exhausting, they are an excellent tool to completely avoid the presentation layer from working directly with domain objects such as Entities or Repositories through abstraction. However, there needs to be a process that can map DTOs to entities and vice-versa. ABP solves this with the integration of AutoMapper, this way, mappings only have to be created once. The specific differences between the objects must be specifically stated, after which all one has to do to transform objects back and forth is to call pre-defined methods.

By encompassing multiple popular and useful tools, ABP manages to create a perfect environment for an application to be developed in, enabling the developer to scale it indefinitely and easily maintain it, even though it is suited for both small and large applications alike. Not only that, but ABP also provides a multi-layered architecture and a strong infrastructure, enabling developers to work on projects that feature different technologies, such as Angular or React, or use traditional HTML, CSS and JavaScript through Razor Pages.

2.2.2 ASP.NET Core

ASP.NET Core is a web framework designed to run on .NET Core, developed by Microsoft and initially released in 2016, 7 years after its predecessor ASP.NET was first released. Since the release of the newer ASP.NET Core, the latter has been retired, meaning that ASP.NET Core is now under Long Time Support. The main difference between the two is their approach on cross-platform development. ASP.NET Core can be used for applications targeting not only Windows, but also Linux and MacOS [2]. Meanwhile, ASP.NET is only available for applications running on Windows machines. Another difference among them is that ASP.NET Core is meant to only use ASP.NET Core MVC solutions for the Presentation Layer, while its older brother features Web Forms, MVC and Web API. Not only that, but ASP.NET Core is optimized for use of both .NET Core, which feature modular and optimized .NET libraries and runtimes for both native Windows as well as for other OS, and .NET Framework, unlike the older version.

Concerning the Presentation layer, for which ASP.NET Core brings many changes with ASP.NET Core, performance has been proven to be a big plus for the more recent framework. Featuring an MVC design pattern [3], the View component is by far the one with the shinier upgrades, with the Razor view engine being the one to embed .NET Code in HTML markup. Features of ASP.NET Core MVC include Routing, Model Binding, Model Validation, Razor View Engine, Tag Helpers and others.

Based on ASP.NET Core's routing, ASP.NET Core MVC features URLs with a searchable and simple structure. Not only can routing be done conventionally, where the routing engine parses incoming request's URLs matching them to specific templates defined by the developer in a seamless manner, but specific routing information can be added to Controllers through Attribute Routing.

Another important activity performed by the framework is working with Models. Controllers can now have request data logic removed through Model Binding, which allows for the data to be converted into objects, rather than HTTP headers and string parameters. Furthermore, Model Validation is also supported through unobtrusive annotations used to decorate the models. Not only are the attributes checked on the server, before a controller calls an action, but also on the client side prior to displaying information to the user through jQuery Validation.

Razor is a compact, expressive and fluid template markup language for defining views using embedded C# code [3]. The Razor view engine is capable of generating web content on the server, while making use of server code within its client-side content, using partial views, which increase modularity and ensure ease of use in a larger application, and working with strongly typed views, adding type checking and IntelliSense support to its multiple features.

Repetitive code has been a menace for developers for a very long time, now being seen in an even more negative light than ever with the increase in popularity of clean code followers. It is not a problem that only affects server-side code, but client-side code such as HTML as well, creating frustrating scenarios that can be avoided with the use of Tag Helpers. They create and render HTML content with the help of server-side code by defining custom tags which offer the advantage of having server-side rendering used through HTML coding style. Making great use of the open-source aspect of ASP.NET Core, Tag Helpers can be created from scratch by developers and are also available through NuGet packages and GitHub repos.

2.2.3 ASP.NET Core MVC

The main frameworks available with ASP.NET Core for creating web applications are Razor Pages and MVC, for server-side rendering, and Blazor for client-side rendering. Razor Pages and MVC are very similar since they both feature the same approach with server-side rendering, having a small number of differences among them. Basically, MVC is the legacy version of the newer Razor Pages. Big differences are featured, however, in .NET's newest version of web application framework, Blazor.

The structural difference between MVC and Razor Pages is that the former features separated models, views and controller, staying true to the MVC design pattern, while the latter focuses on a more unified structure with all its infrastructure organized in Pages that feature code-behind closely tied to the HTML content.

2.2.4 Entity Framework Core

In order to work with a database, a developer needs to have an object-relational mapper known as an ORM, Entity Framework Core is a version of Entity Framework that is open-source and helps .NET developers use .NET objects with their database of choice. There's multiple ways of using EF Core depending on an application's specific situation, with data access being performed through models. A model can be either generated from an existing database, configured manually to match an existing database or create the model first and use Migrations to generate the database afterwards. The last is the most straightforward solution out of the three, enabling developers to save time by changing the database automatically as the model is changed, therefore, being the most efficient and least time-consuming.

EF Core allows the developer to choose which database type it should work with, allowing SQL Server or Azure SQL, SQLite, MySQL, PostgreSQL and more to be used. Being a newer and improved version of Entity Framework, it offers most features implemented in the former version, but also many more features that will only be specific to it, since it is the only one currently receiving support.

2.2.5 ML.NET

Since in this day and age automation and artificial intelligence are rapidly gaining popularity, .NET features its own Machine Learning framework since 2018, completely open-source and cross-platform. It provides a wide range of benefits, such as TensorFlow integration, image processing capabilities, forecasting and anomaly detection, text processing tools and traditional machine learning algorithms that are common among ML frameworks.

Compared to other popular machine learning frameworks, ML.NET proved to be faster and more accurate, being able to train and test a sentiment analysis model with 93% accuracy in only 11 minutes, which proved 6 times quicker than the more popular scikit-learn framework [4]. Researching is not limited to C#, it can also be done in Python, allowing .NET developers to completely train a model in Python and simply load the trained model within minutes.

Bibliography

[1] Volosoft, “AspNet Boilerplate - Web Application Framework,” Volosoft, 2013. [Online]. Available: https://aspnetboilerplate.com/.

[2] A. Lock, ASP.NET Core in Action, Second Edition, 2021.

[3] S. Smith, Overview of ASP.NET Core MVC, Microsoft, 2018.

[4] Z. Ahmed, S. Amizadeh and others, *Machine learning at Microsoft with ML.NET,* 2019.