BABEŞ-BOLYAI UNIVERSITY CLUJ-NAPOCA

FACULTY OF MATHEMATICS AND COMPUTER SCIENCE

SPECIALIZATION COMPUTER SCIENCE IN GERMAN

DIPLOMA THESIS

Traditional Multi-Page Applications and Modern Machine Learning Frameworks

Supervisor

Lecturer Sanda-Maria Avram, PhD

Author

Victor-Cristian Catana

**2022**

Table of contents

* Acknowledgements
* Table of contents
* List of figures
* List of tables
* List of abbreviations
* Introduction
  + Motivation
  + Structure of the thesis
* 2. Theoretical concepts
  + 2.1 Basic concepts and state of the art
  + 2.2 Technologies
    - 2.2.1 ASP.NET Boilerplate
    - 2.2.2 ASP.NET Core
    - 2.2.3 ASP.NET MVC
    - 2.2.4 Entity Framework Core
    - 2.2.5 ML.NET
* 3. Core of the work
  + 3.1 Problem and solution
  + 3.2 Methodology and procedure
  + 3.3 Overview of architecture
  + 3.4 Implementation
  + 3.5 Evaluation
  + 3.5.1 Structure of the test environment
  + 3.5.2 Results and observations
  + 3.5.3 Evaluation and discussion
* 4. Conclusions and Future Work
  + 4.1 Conclusions
  + 4.2 Future Work
* Bibliography / References

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 “new 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 providers, 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 communicate with sellers and officials, also being able to receive personalized product recommendations based on previous purchase history or overall product selling performance.

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.

A screenshot of a computer

Description automatically generated with medium confidence 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 behavior allows for better and easier SEO optimization, 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.

In terms of security, it 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.

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

Graphical user interface, diagram

Description automatically generated 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 as an Object-Relational Mapper for the Infrastructure Layer managing an 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 behavior 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 behavior 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’re 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.

The most important aspect of Entity Framework Core, however, is that it provides ways for the software developer to create relationships programmatically and automatically between tables and entities without ever having to open a Database Management System. Moreover, cascading actions are also configurable and can be enabled by the developer to ensure proper and facilitated data workflow from source code actions to database data manipulation, allowing for a straightforward approach in programming logic.

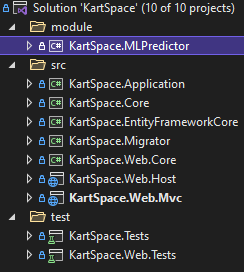
2.2.5 ML.NET

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.

Since ML.NET is relatively a newborn framework, it can make use of all the latest jumps in technology, including AutoML, short for *Automated machine learning* which automates the process of applying machine learning to data [5]. This technology can use training data with different machine learning algorithms and uses a variety of settings for each and every one of them to determine which algorithm is best suited to the problem by evaluating accuracy across all explored models. Different problem and dataset types can be used with specific training scenarios, ranging from Data classification and Forecasting to Object detection and Image classification. Furthermore, the training environment can also be different for each scenario, as ML.NET provides the option to use an Azure and GPU based training for certain scenarios. In order to properly train a model to be trustworthy and dependable, a dataset has to be provided, which can be either a structured text file or an SQL Server database table. In order evaluate multiple models and find the best option for the problem at hand, giving the model builder more time to thoroughly explore various models works best and results in a better final model. Ultimately, consuming the trained model is made easy through project templates, which allow a developer to use either a Console App or an ASP.NET Core web API to make predictions. Moreover, deploying the model can easily be done within minutes by deploying it to Azure.

3. Core of the work

In aiming to create a stable and dependable solution, combining modern and mature technologies was necessary. In order for the mix between the two to be successful and not lead to unexpected issues, the structural and architectural layout of the application had to be very well defined, be it the separation of projects within the solution through an N-Layered architecture or the structure of the database.

In order to preserve principles and best practices necessary for a SOLID programming environment, the application was structured as for allowing ease of use and effortless scalability through popular and common separation and abstraction techniques. Moreover, tests were implemented in order to ensure a normal and predictable behavior of the features within the application. Endpoints and base methods for more used features were thoroughly tested.

Figure 3 - Projects' layout within the solution

3.1 Problem and solution

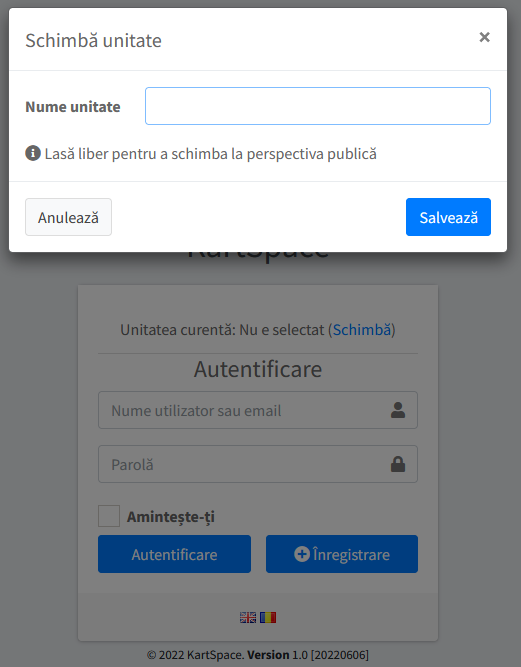
The aforementioned problem of the current situation was that information could not be accessed by users or merchants and the process of gaining access to that information was tedious and frustrating. Through the developed solution, any used can access all the information they might need directly from one source, albeit only by creating an account. Not only that, but merchants can also deliver their products to clients by simply having a Tenant Account created. Once that is done, company users can manage the items for sale, client orders and check statistics.

Figure 4 - Tenant change menu

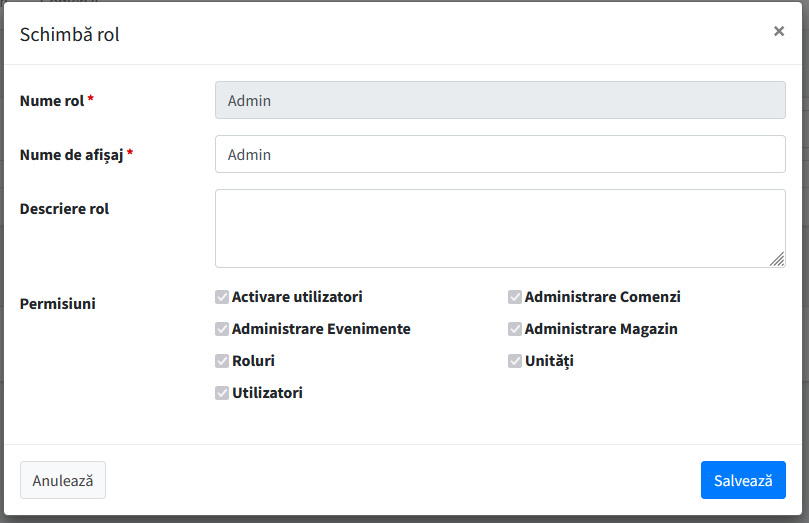
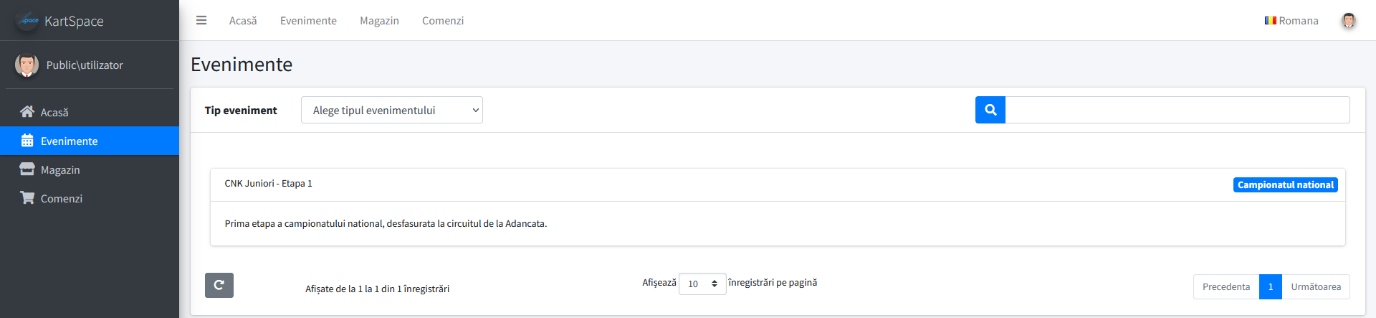
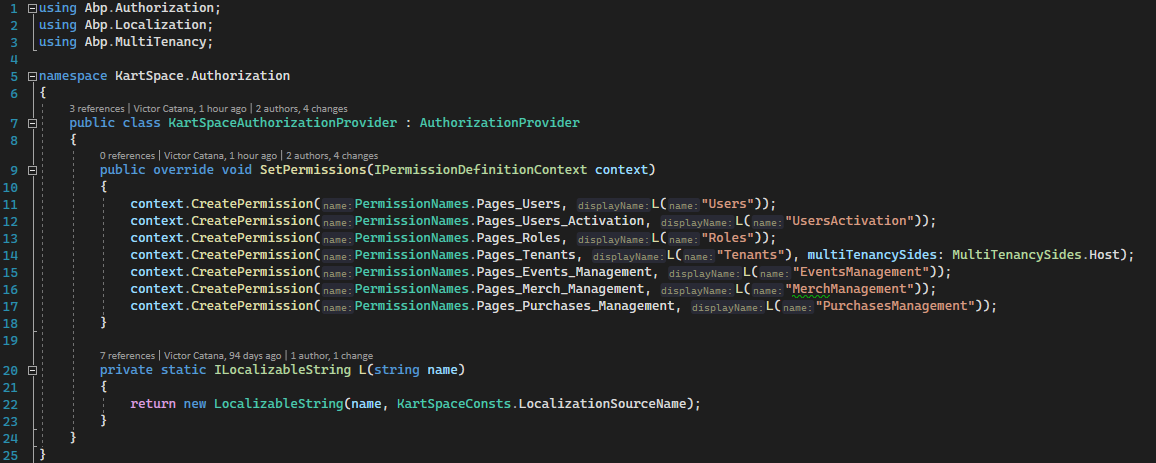
After logging in, one of the features available for every user is the Events section. In this section, administrator can modify or delete existing events or create new ones. On the other hand, users can see past, ongoing or future events in a table featuring pagination and filtering while also being able to select specific event types they would like to see from a drop-down menu. This behavior is implemented through jQuery DataTables and permissions for the administration of this sections is managed through permission checkers in the page’s code. Such permissions can be managed by administrators in the Roles section of the application, allowing for different roles to have specific permissions assigned to them, without the need for roles to be made for each task. Each user can have a role, and each role can have different permissions, enabling administrators to easily manage activities through configurable settings. On tenant creation, an admin account is automatically created and given all permissions. However, there can also be host-specific permissions awarded only by the host administrator, such as tenant management. In code, this is achieved through an AuthorizationProvider class that can be configured for the needs of the application. By specifying the tenancy side that can obtain a permission, the permission only becomes available to host accounts or tenant accounts, whichever is specified. This type of behavior allows for further configurability and separates host admins from tenant admins without the need for complex permission checks.

Figure 5 - Events section for a regular user

Figure 6 - Edit role modal

Figure 7 - Authorization Provider for permission creation (Code snippet)

Events can be accessed both by admins as well as regular users, yet can only be managed by admins which have been granted the Events Management permission. This is only done through checking permissions in the front-end, meaning that in the back-end the endpoints are not permission-proofed.

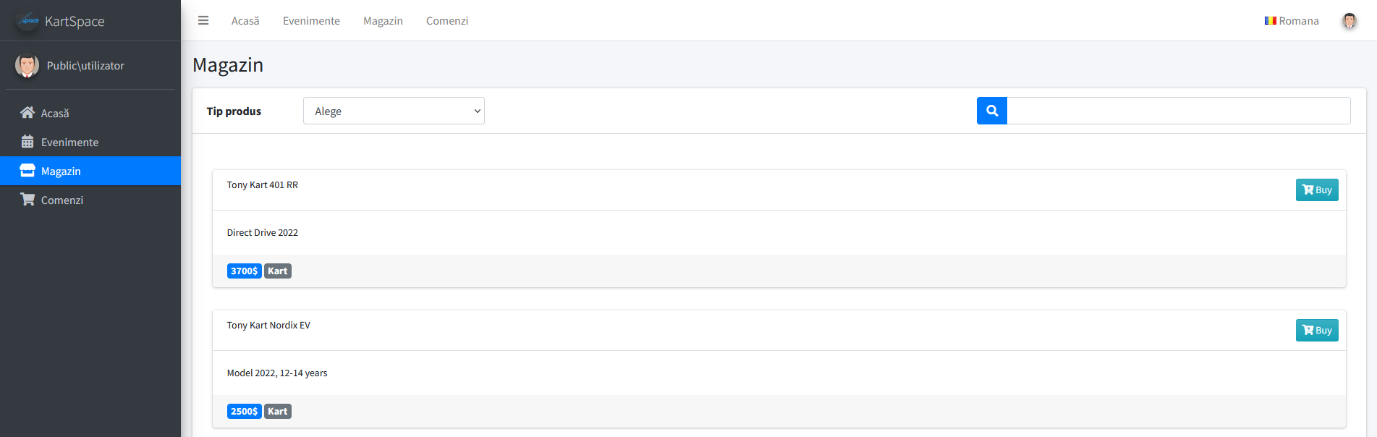
 The second and third views are by far the most important in the whole application since they are the main focus of this work. By using these two pages, Merchandise and Orders, the platform is able to make predictions based on what the user has previously purchased or, if he did not purchase anything, it recommends products which are most popular among other users. The Merchandise page features products from all sellers, unseparated to ensure ease of use and straightforwardness ensuring a smooth user experience. Here, regular users can filter though all available products by either searching for a specific character sequence or choose a product category from the dropdown menu. Also, it is possible for them to purchase a product by pressing the *Buy* button in the top-right corner of each product card, enter their phone number in the pop-up modal and hit save. Tenant users, on the other hand, can not buy products on their tenant accounts. They can all see the products their company has for sale, but only the ones with the *Merchandise Management* permission can create, update or delete merchandise. Such behavior is enforced through permission checks in the JavaScript code of the page, creating and rendering specific buttons and card layouts according to current user data.

Figure 8 - Merchandise section for a regular user

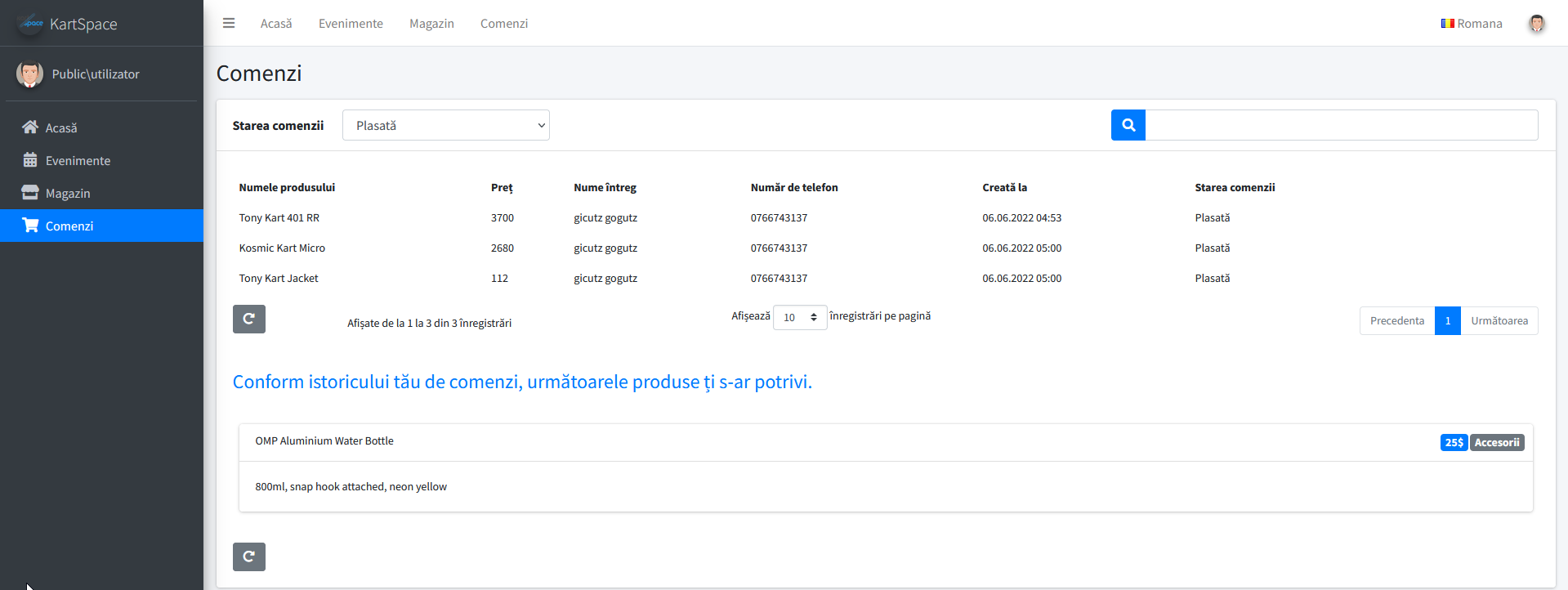
After a product has been bought by a user, they can view their order in the Orders page, along with recommendations based on their last purchase. They can filter their purchases by *Order status* or by simply searching for a specific character sequence which filters through all visible columns except order status, since there is already a dropdown for that. Should the user want to check their recommended product out, all they need to do is press the header, which has an embedded hyperlink, and they will be redirected to the Merchandise page. Tenant users, on this page, can not see recommendations since there is no way for them to make purchases on this type of account. Moreover, unless they have the *Purchases Management* permission where they can also edit orders’ status, they are not allowed to see order details. Instead, they will be met with a message telling them that they are not authorized to view orders in the place where the purchases normally render.

Figure 9 - Orders section for a regular user

3.2 Methodology and procedure

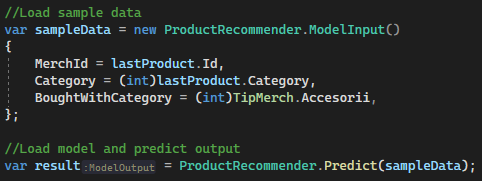
In order to provide the user with meaningful and relevant recommendations, the Purchases application service creates an input data model for the model to consume and calls the *Predict* method of *ProductRecommender*. This is done only if the user has purchases, in order to minimize unnecessary database queries and improve code quality. In order to correctly predict recommendations, a maximum entropy classification model trained with the L-BFGS method is used, providing the best accuracy out of 88 explored models after a training period of 600 seconds. In order to train this model, a dataset of 77 entries was used. The dataset features a product’s Id and its category along with a co-purchased product’s Id and the product category it belongs to. Maximum entropy model is a generalization of linear logistic regression, and the optimization technique implemented in this case is based on the limited memory Broyden-Fletcher-Goldfarb-Shanno method, shortly called L-BFGS. This method is a quasi-Newtonian method which replaces the expensive computation of the Hessian matrix with an approximation but still enjoys a fast convergence rate like Newton’s method.

Figure 10 - Consuming the model

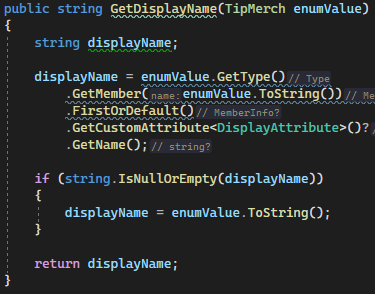
 Displaying data to the end-user is done through AJAX calls from the JavaScript code of each page to the Application Services. In order to properly display the data paginated, the methods each have a request DTO as one of the input parameters, which contains the keyword a user can search for, the skip count and a maximum result count for pagination. After the method processes the input data and retrieves the necessary information from the database, the results are returned in a paged format through a specific Data Transfer Object. In case of a category filter being used, a standard method is used to retrieve the chosen Enum value’s name. This is done by accessing the *Display* annotation of the Enum.

Figure 11 - Retrieving the Display attribute from an Enum value

Accessing Application Service methods is possible from JavaScript code thanks to Dependency Injection, being registered to it and allowing it to be used by other classes. ASP.NET Boilerplate registers Application Services automatically by convention and provides ease of use in JavaScript code. Moreover, the Controllers in the Presentation Layer also benefit from conventional routing, making them easy to access and straightforward to use.

3.3 Overview of architecture

In order to properly reduce complexity, the application follows the principles of Domain Driven Design and features an N-Layered Architecture with the main layers being the Application Layer, providing Application Services and DTOs, the Domain Layer, called Core in the Solution, featuring Entities and Specifications, the Infrastructure Layer, making use of Entity Framework Core and enabling use of Repositories through Dependency Injection and Object-Relational Mapping, and the Presentation Layer, represented by an ASP.NET MVC application. Also, a Swagger integration is also provided and features easy access to the API Endpoints. Moreover, in order to separate external components, not bound to the application’s behavior, the Machine Learning project, which contains the model that is consumed in the Purchases Application Service, is contained, alike the Test projects, in another Solution Directory, other than the source one where the primary, main projects and related functionalities reside. Moreover, the ML project also contains the training information, dataset and other files which have no connection to the main projects, serving a logical separation as well.

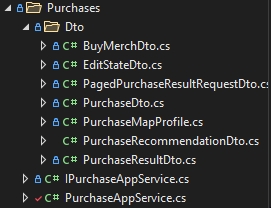
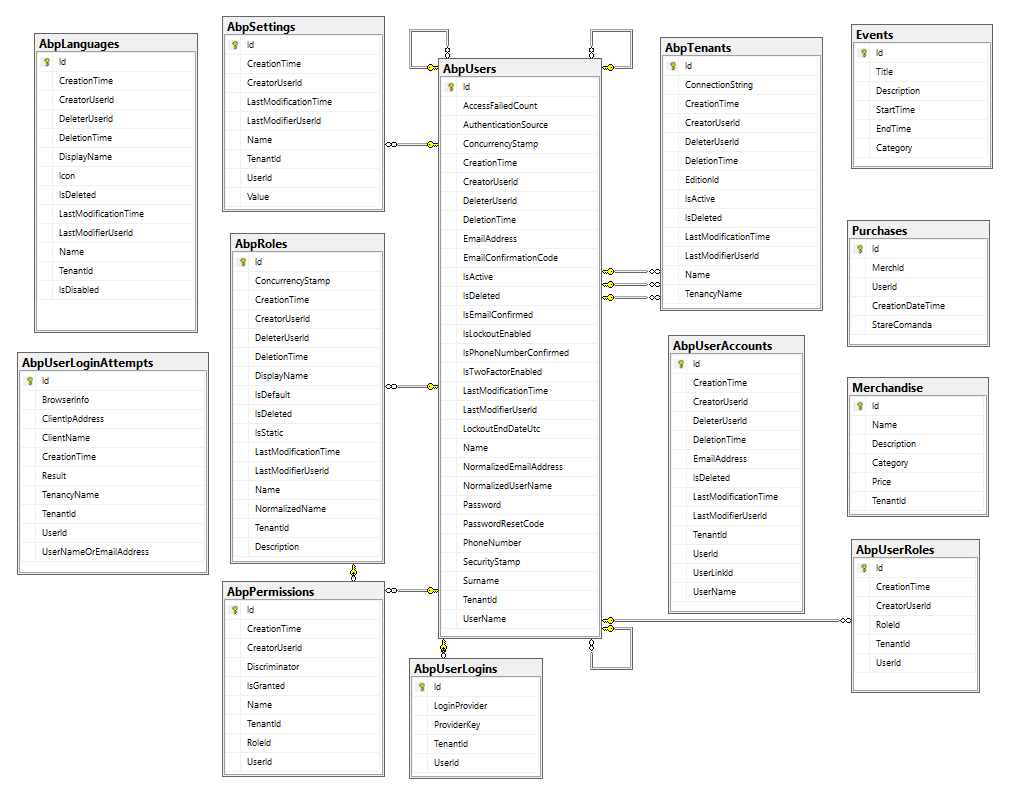
 For the code complexity to be reduced and to allow the Application Services not to work with Entities directly, DTOs are used as a situational middle-man between them. In order to allow mapping between them and entities where needed, a Map Profile needs to be provided, inheriting from Profile, in which the Mappings should be created. As stated before, communication between the application and the database is done through Dependency Injection and, by extension, Repositories. These come with basic CRUD operations, managed in units of work, that provide a simple development flow. The database the application uses Microsoft SQL Server through Entity Framework Core. It is and should always be managed programmatically, through code, since changes in the structure could harm the workflow of the framework and prevent it from efficiently performing its tasks. The Database architecture features the central piece of the puzzle in the application, the User, along with different adjacent entities and individual components such as Languages, used for Localization.

Figure 12 - Standard layout for each Application Service

Figure 13 - Database architecture

Structurally, the solution attempts to follow Domain Driver Design practices and ensure scalability and ease of use. Readability and understandability were also key aspects which were kept in mind when developing this application, as most of the relationships between components are designed to be as direct and simple as possible, allowing for a stranger-friendly approach.

3.4 Implementation

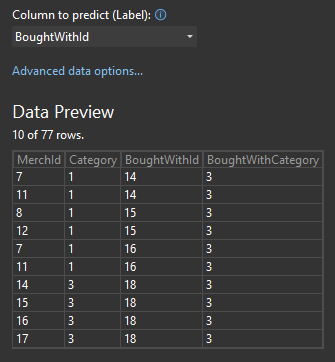
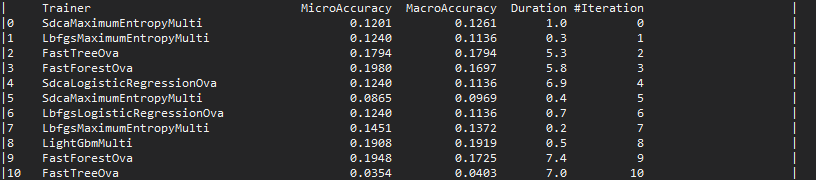
In implementing the features this application uses as a basis for researching modern machine learning framework ML.NET, making use of model builders, complex queries and data rendering strategies was necessary. The ML.NET model builder itself provides a smooth experience, featuring a friendly design. After selecting the dataset for the experiment, the column for the prediction must be chosen, as well as choosing whether or not to ignore certain columns of the dataset. In our case, no column was ignored and the column for the prediction was BoughtWithId. Moreover, a complete overview throughout the training process is visible through the Output window. Each iteration provides an insight into the trainer used, the duration of the iteration and the accuracies the trainer had with certain settings. However, it is not possible to observe the settings AutoML uses to adjust the trainers, which might feel simplistic in some concerns. However, observing the model trainers throughout the experiment provides an insight into how AutoML works. That is, since different settings for certain algorithms provide slightly different accuracy results, providing an insight into the way that Microsoft have created a solution to the never-ending problem that was lengthy and exhausting model training.

Figure 14 - Model training output

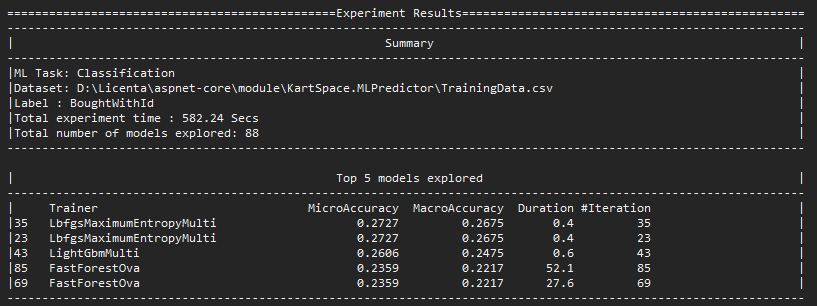
At the end of the experiment, a summary is generated in the model builder interface, as well as in the output window, the latter also featuring the top 5 models explored throughout the training period. If certain model trainers appear multiple times as it happened in this case, it means that slightly different settings were used for the same trainer which resulted in them having the same accuracy. This only happens in cases where smaller datasets are used since there are limited classifications to be made. Testing the trained model can be done manually in the model builder as well.

Figure 15 - Model training end summary and best models

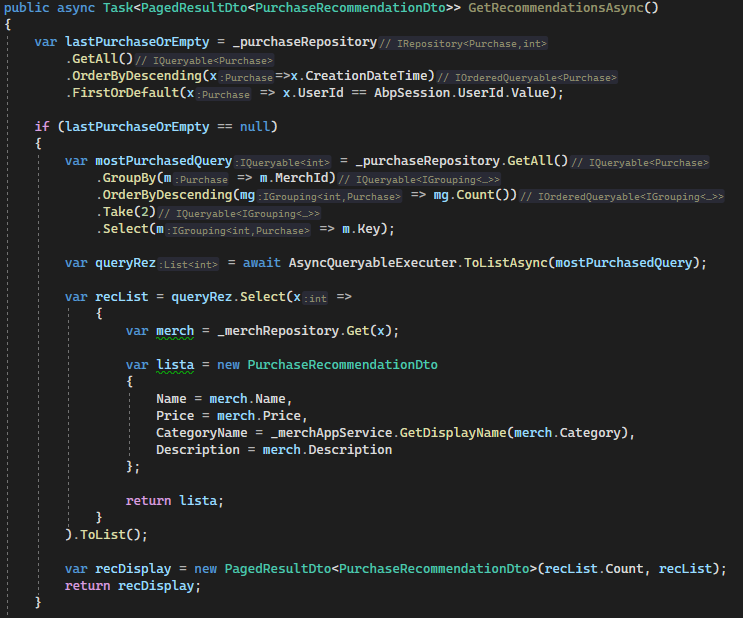
Model consummation is featured in the *PurchaseAppService*, along with a mechanism which provides the user with two recommendations based on overall user preference.

Figure 16 - Product recommendation for users with no purchase history

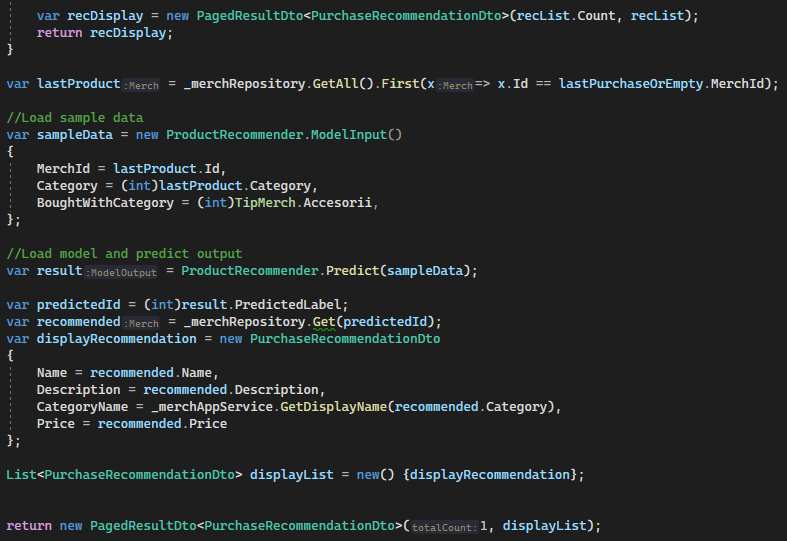
The method searches for any purchases the user might have and recommends a product based on the last one. Should he be a new user that has not yet purchased anything, it provides two products that are preferred by other users instead. The model input is created with the last purchased product and is given a product category chosen by the developer. The model is then used to predict a product Id based on the information provided and, after retrieving the product information from the database, the method returns the recommended item in a paginated list format.

Figure 17 - Product recommendation for users with purchase history, using the trained model

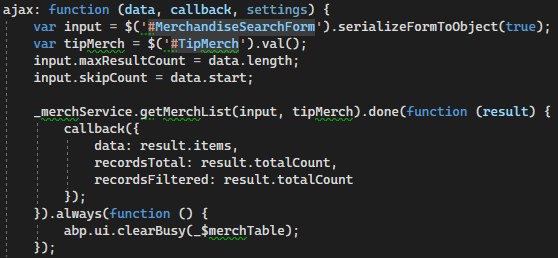
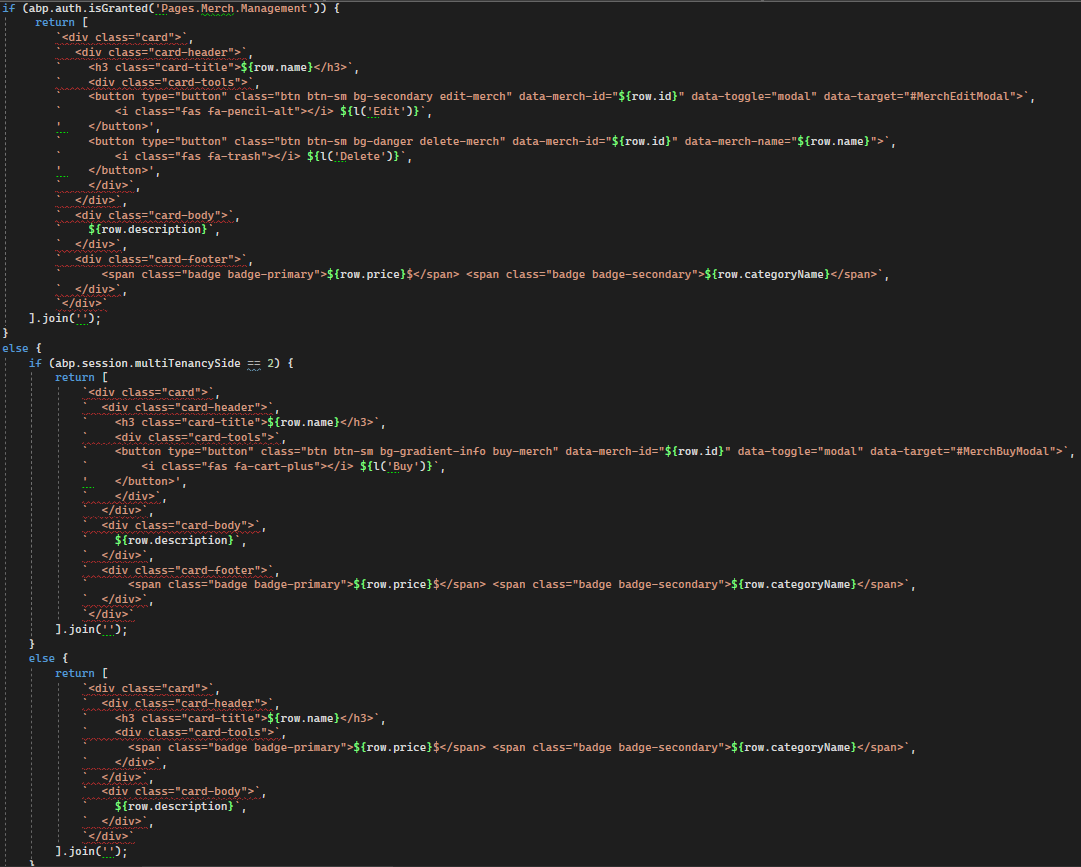
 In order to receive data from the application services under DataTables, AJAX calls were used, according to server-side data sourcing, that retrieved information which allowed the table to also have pagination usable. Furthermore, as the data had to be rendered differently for each type of user, the table was user more as a container that offered pagination rather than a table, with only one column being used and card elements being created through JavaScript code, rather than Cshtml.

Figure 18 - AJAX call for DataTable data retrieval

Figure 19 - DataTable rendering of merchandise depending on user type and permissions

# 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. |
| [5] | e. a. L. Quintanilla, “How to use the ML.NET automated ML API - ML.NET | Microsoft Docs,” 12 10 2021. [Online]. Available: https://docs.microsoft.com/en-us/dotnet/machine-learning/how-to-guides/how-to-use-the-automl-api. |