UNIVERSITATEA DIN CRAIOVA
FACULTATEA DE
AUTOMATICA, CALCULATOARE SI EELECTRONICA

DEPARTAMENTUL DE CALCULATOARE SI TEHNOLOGIA INFORMATIEI

PROIECT DE DIPLOMA

Perisoara Mihai-Cezar

COORDONATOR STIINTIFIC
Asist. univ. dr. ing. Mihai-Catalin SBORA

EASY LEARNING

CONTENTS

INTRODUCTION

PURPOSE

The sole purpose of this documentation is to provide all of our customers with all the necessary information that they need or want to know about our app.

Everything will be explained in detail as well as some crucial stats or features will be pointed out. *Easy Learning* will be built on ASP.NET Core and Ocelot using microservices dedicated to a learning management system.

SCOPE

The scope of *Easy Learning* is to build a webspace where teachers and students can interact somewhere else other than school. In this way, both parties can keep track of all that has been done and that needs to be done in the near future making it perfect to be used by any higher educational institute, being most suitable for faculties. That being said, it doesn't mean it can't be used other educational institutes or by any other organizations.

OVERVIEW

Within this application, there are be some essential features that a Learning Management System needs to have and, also, some unique features to ease some processes a little bit more, each feature being part of a module.

The modules are as follows:

- Authentication/Authorization Module
- Activities Management Module
- Groups Management Module

- Schedule Management Module
- Teaching/Learning Module
- Rooms Management Module

The Authentication/Authorization Module, as its name states, is the module dedicated to authenticate and authorize users, or to put it shortly, to login and register users. A normal user can only use the login system. The registration is done by a super user given to a school representative that has the job to register students and teachers that are within the faculty's institute.

The Activities Management Module contains all the details about activities that take place within an educational institute. These activities can be added with specific details, if some changes occur, any information can be edited, they can even be removed or disabled if the activity has ended or suspended from various reasons, and the activities can be linked, meaning a laboratory can be linked to its course.

The Groups Management Module refers to the groups that each specialization consists of, and those groups have as well subgroups. The groups have a name and the details related to that group including subgroups. The subgroups have a name and they contain the students assigned to that subgroup.

The Schedule Management Module consists of the schedule of the activities. For each activity, the teacher groups and subgroups will be specified with a time interval as well. For courses, it is allowed to overlap activities.

With the Teaching/Learning Module, each user that is a teacher should be able to create classes and assign groups/subgroups/students to each class. Also, classes can be automatically created from the schedule management module with teachers and students being automatically associated with the corresponding class. Here, teachers can add didactic materials, create assignments and mark students with a grade for each assignment, students can upload files or links for an assignment and add comments to an assignment or directly on the class messages.

The Room Management Module represents the module that administrates the situation on each room where classes take part. Each room has a capacity, a name and classes assigned to it, all depending on the schedule, so 2 or more classes cannot be assigned to the same room at the same time.

SUMMARY

The topics of the following chapters will include a presentation of Learning Management Systems (LMS), the existing solutions and what are the requirements for a LMS, a presentation of Microservices, with some information about their architectures and some of the most important advantages and disadvantages that they have, a description of application's architecture, containing all the information about microservices, about the data model and about user interface, a description about how the application was implemented, the main information being about the microservices that were implemented, the mechanisms that were used for communications, the implementation of the user interface and the technologies that were used to get to the final point of the application, and, finally, the conclusion.

LEARNING MANAGEMENT SYSTEMS

WHY ARE THEY USE FOR

A Learning Management System, shortly known as LMS, is an online system or software which is mainly used for planning, executing and assessing a specific learning process. To put it in other words, it is a software widely used in e-Learning for the administration, documentation, tracking, reporting, automation and delivery of educational courses, training programs, or learning and development programs. Learning Management Systems are mostly used to maintain an online collaboration between students and teachers, all while being over the internet.

They are most commonly used in colleges and companies to deliver online training. Corporates use them for training purposes, as well as maintaining employee records. Some of them are used to offer courses that provide students access to education and some others are used as an online system which staff use to support the delivery of courses and to provide opportunities for students and employees using online learning and blended learning, which is an approach to education that combines online educational materials and opportunities for interaction online with traditional place-based classroom methods.

The main purpose of Learning Management Systems is to enhance the learning process of a student or an employee. A Learning Management System, besides the fact that it delivers content, it handles registering courses, course administration, skill gap analysis, tracking and reporting. Most of the Learning Management Systems are web based and are used in various educational institutes and companies to improve classroom teaching, learning methodology and company records. In education markets, an LMS will include a variety of functionalities that are similar to the ones used in corporates, but will have features such as rubrics, teacher facilitated learning, and a discussion board.

Learning Management Systems come with a large number of benefits, which is the reason why they are used worldwide.

One of the benefits, which is one of the most important ones, if not the most important, is that the Learning Management Systems make things easier for every user, as it helps plan the training activities calendar which can be shared with learners, trainers, and co-administrators. This means that the training process can be easily maintained and, also, can be vastly improved. A LMS also helps a lot in getting reports, which is a well-known time-consuming process when it is done manually.

Another benefit is the ability to deliver a training that is engaging and motivating. Each individual has its own way of learning and comprehending new information. If different learning methods are approached, an LMS can greatly increase an individual’s success rate.

The technology leverage also represents a benefit of Learning Management Systems. In this digital world, the working system has drastically changed. The employee of today is engaged with a laptop or a desktop, and with the help of LMS, it is possible to make the training future ready.

The learning process is centralized. These days, it becomes easy to offer a centralized source of learning to multiple users and the training, performance and content can be accessed from the same source.

Another benefit that an LMS has is that performance can be enhanced through tracking and recording tools. A users' progress can be tracked, records can be reviewed and users can register for more than one courses. Which means that learning can be done through web based training. The management part can access records and can calculate which area needs to be improved. With this, learners also become aware of the areas that need improvement and additional efforts, as the weak performance areas, can be identified with ease.

Another benefit is the evaluation capability. This means that users can evaluate courses, and employees can keep track of real time performance by periodically scheduling assignments.

Content and information are easily upgradable, as Learning Management Systems offer a centralized location of information making it simple to implement changes.

The learning process is simplified as an LMS refines it. A first time user can figure out everything with ease as these type of systems are easy to use.

An LMS can reduce the cost of the training and teaching processes in many ways, as there are no charges involved or any major costs involved. This means that Learning Management Systems save valuable time and money through web based training.

The environment is more interactive. With the help of new online tools, the interaction and communication improve. Learners are able to get answers in real time and the engagement is more geared toward being interactive.

And one last benefit is the fact that an LMS can pe used at any time wherever a user may be as long as they have the proper device and a connection to the internet.

All these benefits come from all the key features that a Learning Management System has. Some common features that can be found in the majority of Learning Management Systems are:

- Managing of users, courses, roles and generating reports. This helps with the upload of courses, assigning roles and generating various reports.
- Making a course calendar, which improves the managing of courses activities.
- Messaging and notifications. This means that any user can send reminders and notifications.
- Assessments that can handle pre/post testing.
- Certification and display score and transcripts. This helps in maintaining training records of the learners, performance of the individual and providing certificates to those who have completed successfully.
- Instructor-led course management.
- Administration.
- Competency management.

EXISTING SOLUTIONS

As online learning is gradually becoming the main method of learning, choosing the proper Learning Management System is more important than ever, as there are tons of things to consider when it comes to opting for the right LMS. Educators and administrators generally want a LMS that they can rely on when it comes to develop and assign course content, track student progress, and measure and report student outcomes.

For the past few years, the educational LMS space has evolved considerably and the market has consolidated around a handful of major vendors, and with that concentration there is greater uniformity in design and features. The Learning Management Systems of today are responsive, highly interoperable and cloud hosted. With that in mind, many companies came with various LMSs, each one being unique in their own way.

There are 2 types of Learning Management Systems:

- Cloud-Based
- Open Source

Cloud-based LMSs are hosted on the web on the vendor’s server and are accessible from anywhere on any device by logging in on a dedicated page. Because the solution is developed, hosted and supported by the LMS provider, users don’t have to worry about enhancing or maintaining the software, and also personal data is secured. Such an LMS is considered user-friendly and helps the user to avoid IT hustle. A few of the most known and used cloud-based LMSs are:

- Docebo
- Adobe Captivate Prime
- TalentLMS
- LearnUpon LMS
- Inquisiq LMS
- LearningPool
- Google Classroom

and these are just some of them.

Docebo, mainly used by enterprises, is known for applying new technologies to the traditional cloud LMS, including Artificial Intelligence, and for supporting the ways people learn. This Learning Management System is praised for its simplicity to use, high configurability, robust learning management, delivery functionalities and affordable monthly active user pricing model. It includes over 35 native integrations and APIs, like CRM (Salesforce), eCommerce (Stripe, Shopify), collaboration and productivity tools (Slack, G Suite), web conferencing (GoToMeeting, Zoom).

Adobe Captivate Prime delivers and enhanced and enjoyable learning experience for learners. Users can take advantage of the AI-based social learning capabilities to encourage learning in the flow of work and allow peers to learn from each other. Through this LMS, enterprises can manage end-to-end training effectively across the value chain, employees can develop their skills, customers can be educated and, at the same time, their satisfaction can be improved, and the sales enablement become possible. This makes Adobe Captivate Prime a valuable LMS as it is also being used by numerous prestigious companies.

TalentLMS works as an all-in-one solution for businesses looking to build training programs that will result in stronger teams. This makes TalentLMS the leading training platform that allows anyone to build courses in minutes and launch them by the end of the day, no matter the familiarity with LMSs. Within this TalentLMS, users can set up training without manuals and without spending too much time figuring out how it works, training is always available on the cloud, even if it's on desktop, tablet, or phone, and it's free with the option to upgrade to a premium account.

LearnUpon LMS is built to efficiently manage, deliver, and track training for every audience. By combining their powerful, user-friendly platform with an excellent support team, this LMS delivers high impact training that fuels long-term growth. The ups of LearnUpon LMS is that it is easy to use and quick to set up, with a simple, intuitive UI for admins and learners, backed by extensive User Experience testing, it was built with multi-portal functionality, giving the platform the ability to scale and handle complex workflows at a large capacity, the strategy is customer-driven by adding new features and enhancements to the roadmap, those being demanded by customers and potential customers, and it has a world class support.

Inquisiq LMS seamlessly blends the most advanced eLearning technology and time-tested training methods within a rich learning ecosystem that fosters collaboration. It is a web-based, SCORM-compliant LMS for small to medium sized businesses. It has a configurable interface with full mobile compatibility, meaning that the user interface is configurable and customizable via the admin control panel, it is a great tool to implement blended learning, which allows to combine in-person learning with eLearning tools to allow a cohesive learning experience. With this LMS, learning automation takes the administrative task load burden off of the administrator. This LMS helps users feel connected with other groups and communities, making social sharing spontaneous, less formal and it can foster greater collaboration and a sense of community.

LearningPool is one of the most feature-rich LMS on the market. It is fully hosted and supported to deliver great learning, easy compliance, and clear reporting. Easy compliance means that a business can be kept on track with statutory and mandatory training. The support is built for confidence and peace of mind as the team is led by a dedicated learning consultant, backed up by telephone and online support and underpinned by their learning academy. The management of information is improved and keeps learner up to date with critical learning and reports progress in real time. LearningPool gives freedom to transform one’s learning and create a solution that can innovate, integrate, and scale with one’s needs, making also this LMS an opened-source one. There also have been created more than 70 add-ons to make this platform as user-friendly as possible.

Google Classroom is a cloud-based Learning Management System that is a part of Google Apps for Education. It is suitable for academic institutions. It enables students to access the platform from any device. Users can create Google Docs to manage assignments, port YouTube videos and attach files from google drive. It enables users to provide feedback through comments on the documents. Teachers can create online classrooms for sharing the learning materials that students can download and view. There also can be created online assignments to keep track of students’ progress. The connectivity of Google products such as Google Docs, Google Spreadsheets and Google Slides allows students to submit their assignments easily.

For open source LMSs things are different. An open source LMS requires installation and setup. One of the advantages of an open sourced LMS is that it is budget friendly since it can be found at a low price, but also there are free solutions as well. Since a setup is required for this kind of LMSs, they can be installed on a local server, as a result, making a user to take care of its maintenance and data. The downside is the updates need to be done manually and support can be found through forums and user communities. Now, some of the most known open source LMSs are:

- Moodle
- Chamilo
- Canvas

Moodle is widely known among open source LMS solutions. It features detailed guides on how to set up a LMS, tips on how to create online training courses and teach with Moodle. It features extensive tutorials for various aspects Learning Management System use, including installation, teacher and administrator quick guides, course setup, and learner progress tracking. It comes with a forum that covers tips and best practices on various topics. Its download sections offers plenty of additional plugins for activities and themes. All these are available and free, as Moodle does not require any subscription fee.

Chamilo offers easy-to-use authoring tools for creating online training that meets all learning preferences. It is mobile friendly as it offers mobile learning support, it allows users to learn at their own pace and access the online training material whenever they find it convenient. It has a great community as users can turn to it to get the support they need, as the community is available 24/7. Its interface is customizable and offers intuitive language and media embedding settings with plenty of options.

Canvas is an open source LMS that is free to use and makes teaching and learning easier in terms of implementation, adaption, customer support, and success. Its interface and features are crafted to save time and effort, resulting in getting adopted faster and deeper than many other Learning Management Systems. It has a powerful community with over 300,000 users and it is up to the standards of online learners and instructors. It is easily accessible from mobile devices as well. It is an easy plug-in for third party tools because its extensive API paves the way for those apps.

LMS REQUIREMENTS

As any other system there is, when a Learning Management System is being built, it has to meet two types of requirements: functional requirements and technical requirements.

The functional requirements of a LMS, which are the actual LMS features that learners and administration will be using, are:

- User and course management – in a LMS it is needed to decide how to work with users and groups, meaning how will new users be registered, who will manage the LMS and how will new users be added;
- Learning models – the learning model will be chosen to suit best for a one's LMS; here we have a few different models:
  - Pure learning – users are taught only online by taking courses, tests and reading the provided materials;

    - Blended learning – only a part of the learning process is online;
    - Instructor-led training – this allows learners and instructors/teachers to interact and discuss the training materials.
- Support and creation of learning content – the learning content is the most important part of a LMS, which means that it is highly important to decide how the learning materials will be distributed through the LMS;
- Mobile Learning – it is important for many students and learners that are on the move to have content that is available and looks great on all mobile devices.

Technical requirements, that are the technical issues that must be considered to successfully complete the project, are:

- Cloud-based – the LMS is built to be hosted on the web, so it doesn't require deployment and users can start working with it right away;
- Security – the security is a vital point of the LMS. The user's data needs to be stored safely and to be accessed only by themselves;
- Integration with other systems – this part must be considered if a LMS is needed to be used in different areas that require integration with their own system.

MICROSERVICES

GENERAL PRESENTATION

Microservices architecture arranges an application as a collection of loosely coupled services. In these kind of architectures, services are fine-grained and the protocols are lightweight.

In a microservice architecture, services are often processes that communicate over a network to fulfill a goal and are also organized around business capabilities. They can be implemented using different programming languages, databases, and hardware and software environments. Services, in general, are small in size, bounded by context, autonomously developed, independently deployable and built and released with automated processes.

A microservice is a self-contained piece of functionality that has clear interfaces, and which may implement a layered architecture by using its own components.

Microservices are most common in applications that are cloud-native, in serverless computing and applications using lightweight deployment. As the number of services is large, decentralized continuous delivery and DevOps are necessary to effectively develop, maintain, and operate applications that use microservices architecture.

MICROSERVICES ARCHITECTURES

When microservices architecture is involved in a project, it is crucial to choose what are the best suited services for a project, as each of these services is responsible for discrete tasks and can communicate at the same time with other services through simple APIs to solve a larger complex problem.

First thing is the definition of services corresponding to business capabilities, which are something that a business does in order to provide value to its users. Identifying business capabilities requires a high level of understanding and once they are identified, the required services can be built corresponding to each of these identified business capabilities.

After the service boundaries of these small services have been decided, they can be developed using the proper technologies which are best suited for each purpose. In this stage, the CI/CD pipelines can be setup with any of the available CI servers like Jenkins, Team City etc. to run automated test cases and deploy services independently to different environments.

Individual services need to be designed carefully. When services are in this phase, they need to be carefully defined, which also implies the need of thinking about what will be exposed, what protocols will be used to interact with the service, etc. It also means that it is very important to hide any complexity and implementation details that a service consists of and expose only what’s essential to service’s clients. If that happens, the task of changing the service will be a lot harder as it is hard and involves a lot of work to determine who is relying on various parts of the service.

One of the common mistakes in designing microservices is when a service stores all the information that it needs to a database and another service, that is created and needs the same data, accesses that data directly from the database. With this approach, control is immediately lost in determining what is hidden and what is not. Which means, that in order to tackle this, the second service should access the first service and avoid going directly to the database, therefore preserving utmost flexibility for various schema changes that may be required. The protocols for communication between services must be chosen carefully.

One more approach is to decentralize things, meaning that the teams who build the services don’t have to be anymore the ones who develop, deploy, maintain and support, following the path of open source model. With this approach, the developer who needs some changes to the service can check and work on a feature by himself, without waiting for the one who owns the service to work on the needed changes. One way to decentralize things is to have a collection of services that are communicating via a central message bus. This bus handles the routing of messages from different services. The more logic is put into this central bus, the more intelligent it becomes, which may be an issue as it makes it harder to do changes.

When it comes to deploying microservices, there are two common models. First, multiple microservices per operating system can be deployed. With this model, there is a time save when automating certain things. The downside when using this approach is that it limits the ability to change and scale services independently and also creates difficulty in managing dependencies. The independent services can produce unwanted side effects for other services that are running at that point in time, making it very difficult to reproduce and solve. The second model uses one microservice per operating system, which makes it the preferable choice. With this model, the service is more isolated and hence it’s easier to manage dependencies and scales services independently.

A common problem that is typically faced with a microservice is determining how to make changes in existing microservice APIs when other microservices are using it in production. By making changes in a microservice API, it might break another microservice that is dependent on it. There are a few ways to solve this issue.

One is to version the API. By using this method, a new version of the API can be deployed while keeping the previous version up. The dependent services can be upgraded afterwards at their own pace in order to use the newer version of the API. Once all services are migrated to the newer version, of the changed microservice, the old API can be brought down. One problem this approach brings up is that it becomes difficult to maintain various versions, as any new changes or bug fixes must be done in both versions. For this reason, an alternative approach is one where another end point is implemented in the same service when changes are needed. Once the new end point is fully utilized by all services, then the old one can be deleted.

In a microservice architecture, over time, each service starts depending on more and more services, which can introduce more problems as the services grow. The number of service instances and their locations might change dynamically. Also, the protocols and the format in which data is shared might vary from service to service. Here, API Gateways and Service Discovery become very helpful. Implementing an API Gateway becomes a single entry point for all clients. These can expose a different API for each client. The API Gateway might also implement security like verifying if the client is authorized to perform the request.

An important point is that microservices aren't resilient by default, as there will be failures on services, which can happen because of failures in dependent services. That means it is critical in microservices architecture to ensure that the whole system isn't impacted or goes down when there are errors in an individual part of the system. For this particular reason, there are patterns which can help in achieving better resiliency.

One of which is Bulkhead. This pattern isolates elements of an application into pool so that if one fails, the other will continue to function. Another pattern is the Circuit Breaker. It wraps a protected function call in a circuit breaker object, which monitors the failures. Once a failure crosses the threshold, the circuit breaker trips and all further calls to the circuit breaker returns an error. After the timeout period expires and if some calls are allowed to pass through and also succeed, then the circuit breaker return to its normal state.

Monitoring and logging individual services can be a challenge making it hard to figure out individual errors. To solve such problems, a preferred approach is to take advantage of a centralized logging service that aggregate logs from each service instance. Users are able to search through these logs from one centralized spot and configure alerts when certain messages appear. Similar to log aggregation, stats aggregation can also be leveraged and stored centrally. When one of the downstream services is incapable of handling requests, the solution is to implement health check APIs in each service, as they return information on the health of the system. A health check client invokes the endpoint to check the health of the service instance periodically in a certain time interval.

ADVANTAGES AND DISADVANTAGES

Like any other architecture, microservices also have their advantages and disadvantages. Depending on those, an enterprise is able to decide if it’s more suited for them to use microservices architecture or not.

To begin with the **advantages** of microservices architecture. Microservices work well with agile development processes and satisfy the increasing need for more fluid flow of information.

Microservices are independently deployable and allow for more team autonomy, meaning that each microservice can be deployed independently, as needed, enabling continuous improvement and faster app updates. This allows assigning specific development teams helping them focus solely on one service or feature. This means teams can work autonomously without worrying what’s going on with the rest of the app.

Microservices are also independently scalable, meaning that as demand for an app increases, it’s easier to scale using microservices. The resources can be increased to the most needed microservices rather than scaling an entire app, meaning the scaling process is faster and more cost-efficient as well.

Microservices reduce downtime through fault isolation, meaning that if a specific microservice fails, the failure can be isolated to that single service and prevent cascading failures that would cause the app to crash. This fault isolation means that even when one of the modules fails, the critical application can stay up and running.

The smaller codebase enables developers to understand more easily the code, making it simpler to maintain and deploy. It’s also much easier to keep the code clean and for teams to be wholly responsible for specific services.

With microservices, vendor or technology lock-in can be eliminated, as microservices provide the flexibility to try out new technology stack on an individual service as needed. There won’t be as many dependency concerns and rolling back changes becomes much easier. Also, with less code in play, there is more flexibility.

As for **disadvantages**, while much of the development process is simplified with microservices, there are still a few areas where microservices can actually cause new complexity.

Microservices create different types of complexity than monolithic applications for development teams. Firstly, communication between services can be complex, as application may include dozens or even hundreds of different services. All those services need to communicate securely. Secondly, the debug process becomes more challenging when using microservices. When an application has multiple microservices, each microservice having its own set of logs, tracing the source of the problem becomes difficult. Thirdly, while unit testing is easier with microservices, integration testing is not, as the components are distributed, and developers are unable to test an entire system from their individual machines.

The control over interface becomes more critical. Each microservice has its own API, which apps rely on to be consistent. While a developer can easily make changes to a microservice without affecting the external systems interacting with it, when changes occur to the API, any application using that microservice will be affected if the change is not backwards compatible, as a microservice architecture model results in a large number of APIs.

When using microservices, costs may be higher than usual. For microservices architecture to work, it is needed sufficient hosting infrastructure with security and maintenance support. For that, skilled development teams have to understand and manage all services. If those things are already in place, the costs involved in moving to microservices may be lower.

Clearly there are many advantages and disadvantages of microservices architecture to consider when building an application based on that.

APPLICATION ARCHITECTURE

MICROSERVICES

As *Easy Learning* is a Learning Management System based on *Microservices*, this application consists of multiple microservices every single one running independently of each other. Those microservices are:

1. Authentication and Authorization Microservice
2. Activities Management Microservice
3. Groups Management Microservice
4. Rooms Management Microservice
5. Schedule Management Microservice
6. Teaching and Learning Management Microservice

The *Authentication and Authorization Microservice,* as its name states, is the main authentication and authorization service for any user. It stores and manages all the users and their data in its own database and provides each of the users with their respective user-roles. Those roles are necessary for users in order to access the other services that are available to use based on their respective role. This microservice can not be accessed by a normal user, only by an administrator or a super user, the role of super user being given by the administrator.

The *Activities Management Microservice* manages all the activities that are needed in order to create the classes for those activities. The information regarding any activity is stored in the database build for this microservice. The database consists of different types of activities as there are more than one kind of activities. An activity can be modified in any way by a user with the role of administrator or super user, as any activity might need small changes.

The *Groups Management Microservice* is the microservice that is focused on the groups and subgroups that take part in the classes built for each activity. The groups are the main part of the microservice, while subgroups are to be created and attached afterwards to their respective groups. The groups contain the primary information that needs to be specified about it, and the subgroups contain the list of students that are part of that subgroups. Like the other microservices, this microservice also has its own database to store the groups and subgroups.

The *Rooms Management Microservice* is the microservice that manages the rooms in which the activities take place. With this microservice, a super user can keep track of the all the rooms that their institute has in order to ease the process of assigning them properly to each activity based on how many students take part.

The *Schedule Management Microservice* is a microservice used for managing the schedule for all types of activities that take place, making the Schedule Microservice closely related to the Activities, Groups and Rooms Microservices as those need to be specified in order to create a proper schedule for all activities.

The *Teaching and Learning Management Microservice* is the main microservice where the classes which contain the information from previous microservices are created. Here teachers can post their didactic materials and students are able to interact with them in different ways. The classes are created starting from activities followed by the addition of groups/subgroups/students and teachers.

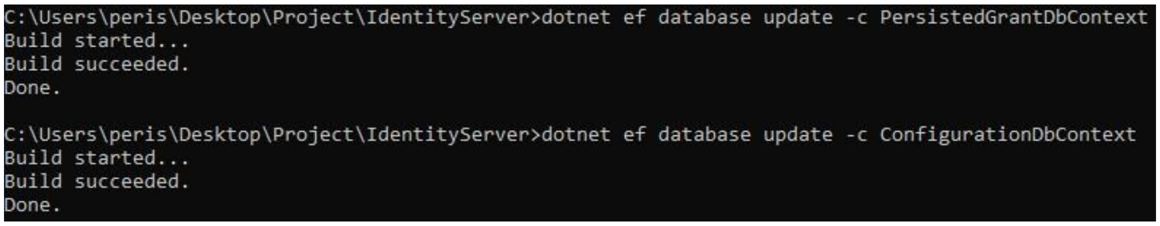
All these microservices are integrated in the final client application *EasyLearning*. A normal user, meaning a student or a teacher, can only access the main page and what the *Teaching and Learning Microservice* has available for those two types of users.

DATA MODEL

As there are multiple microservices and each microservice runs independently, respecting the microservices architecture, every single one of the microservices have their own unique data models and databases.

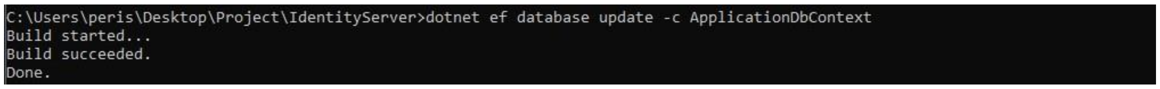
Authentication and Authorization Microservice

The *Authentication and Authorization Microservice* is a microservice built and developed around *IdentityServer 4* and *ASP.NET Core Identity*, so this microservice's data model comes with IDS4 and .NET Core Identity Frameworks. The database is built by running the migrations required for IDS4 (Fig. 1) and .NET Core Identity (Fig. 2).



```
C:\Users\peris\Desktop\Project\IdentityServer>dotnet ef database update -c PersistedGrantDbContext
Build started...
Build succeeded.
Done.

C:\Users\peris\Desktop\Project\IdentityServer>dotnet ef database update -c ConfigurationDbContext
Build started...
Build succeeded.
Done.
```

FIG. 1



```
C:\Users\peris\Desktop\Project\IdentityServer>dotnet ef database update -c ApplicationDbContext
Build started...
Build succeeded.
Done.
```

FIG. 2

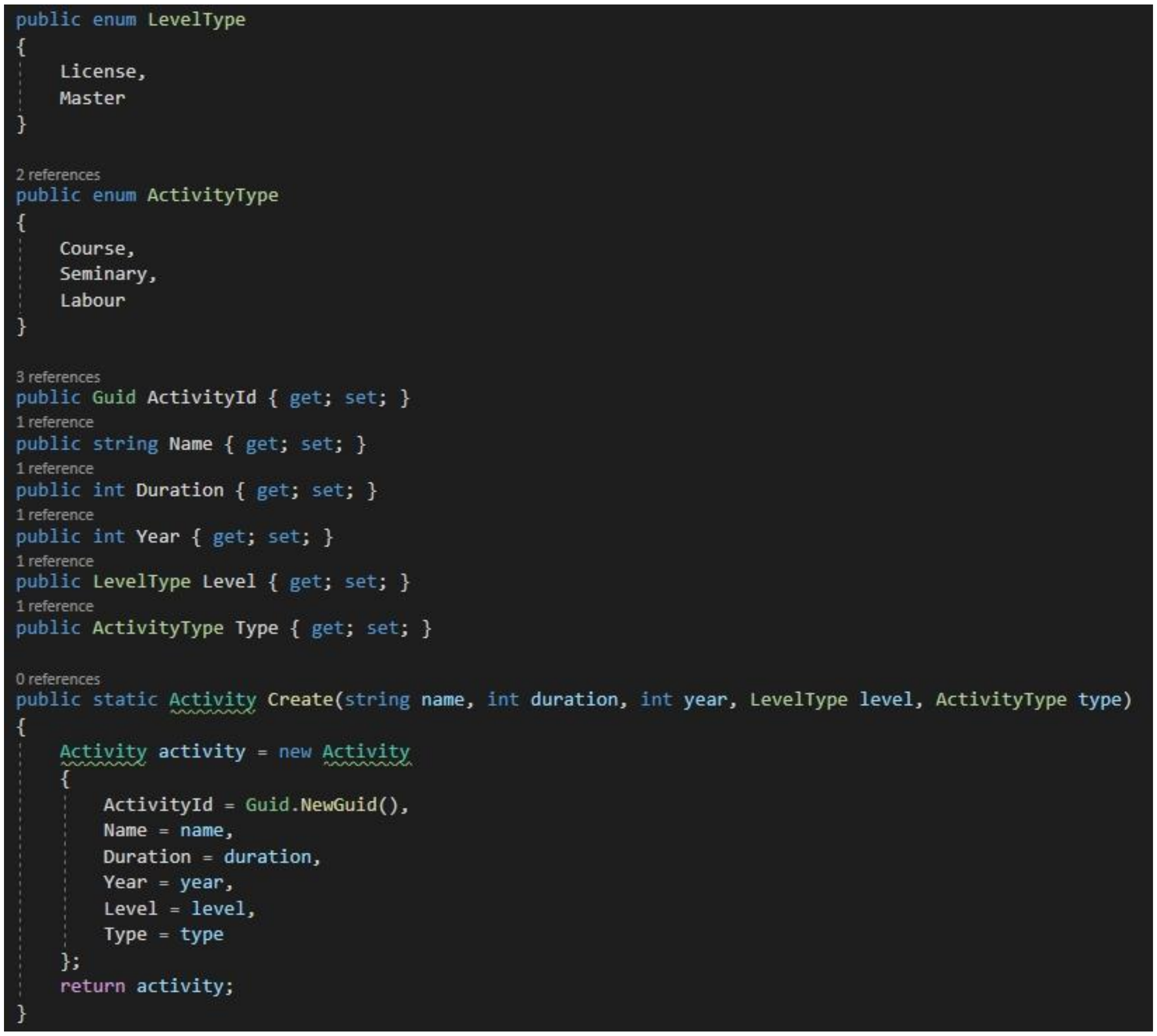
The two DbContexts, *PersistedGrantDbContext* and *ConfigurationDbContext,* are contexts required by IDS4.

The context *PersistedGrantDbContext* has the role to store for a short period of time operational data like authorization codes and refresh tokens. The context *ConfigurationDbContext* is used to for configuration data, meaning clients, resources and scopes.

The context *ApplicationDbContext* is required to run the migrations for ASP.NET Identity which is responsible with the users that are involved with it.

Activities Management Microservice

The *Activities Management Microservice* data model (Fig. 3) consists of one table that covers all the required data that is needed in order to store the activities.



```
public enum LevelType
{
    License,
    Master
}

2 references
public enum ActivityType
{
    Course,
    Seminary,
    Labour
}

3 references
public Guid ActivityId { get; set; }
1 reference
public string Name { get; set; }
1 reference
public int Duration { get; set; }
1 reference
public int Year { get; set; }
1 reference
public LevelType Level { get; set; }
1 reference
public ActivityType Type { get; set; }

0 references
public static Activity Create(string name, int duration, int year, LevelType level, ActivityType type)
{
    Activity activity = new Activity
    {
        ActivityId = Guid.NewGuid(),
        Name = name,
        Duration = duration,
        Year = year,
        Level = level,
        Type = type
    };
    return activity;
}
```

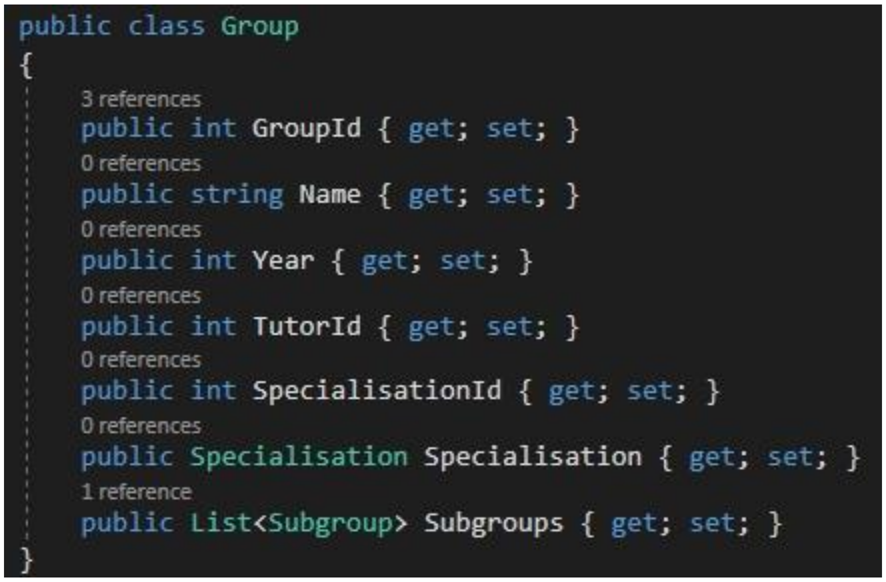
FIG. 3

Every activity that is added has an ID, a Name, a Duration and a Year, which refers to the year of the level type in which the activity takes place. LevelType refers to the educational level of the activity and ActivityType refers to the one of the three types of activities.

Groups Management Microservice

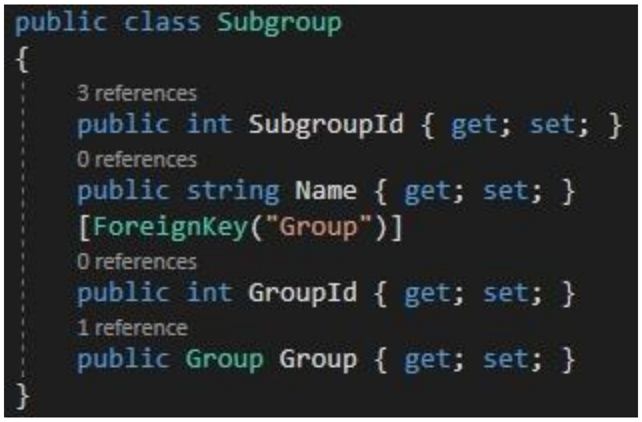
The *Groups Management Microservice* is formed of 3 data models:

- Groups data model – contains the entities needed to describe a group (Fig. 4);
- Subgroups data model – contains the entities needed for a subgroup (Fig. 5);
- Specialization data model – used in case there is a need to add more specific specializations that a educational institute has (Fig. 6).



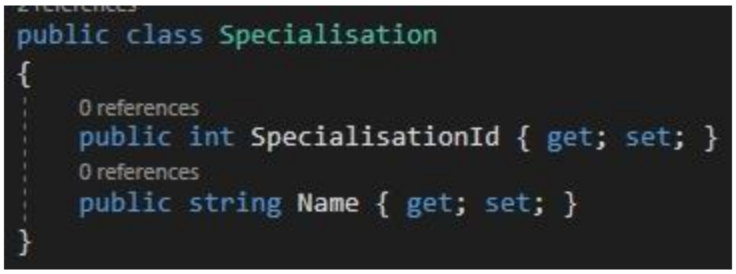
```
public class Group
{
    3 references
    public int GroupId { get; set; }
    0 references
    public string Name { get; set; }
    0 references
    public int Year { get; set; }
    0 references
    public int TutorId { get; set; }
    0 references
    public int SpecialisationId { get; set; }
    0 references
    public Specialisation Specialisation { get; set; }
    1 reference
    public List<Subgroup> Subgroups { get; set; }
}
```

FIG. 4



```
public class Subgroup
{
    3 references
    public int SubgroupId { get; set; }
    0 references
    public string Name { get; set; }
    [ForeignKey("Group")]
    0 references
    public int GroupId { get; set; }
    1 reference
    public Group Group { get; set; }
}
```

FIG. 5

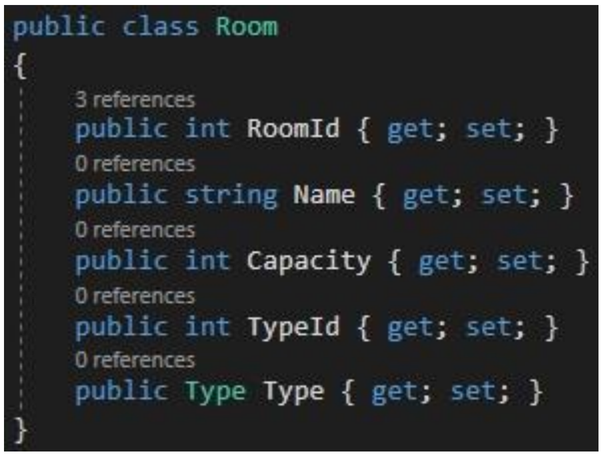


```
public class Specialisation
{
    0 references
    public int SpecialisationId { get; set; }
    0 references
    public string Name { get; set; }
}
```

FIG. 6

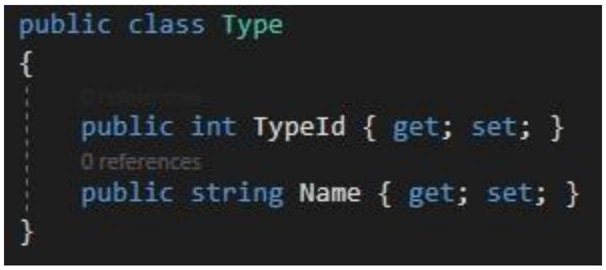
Rooms Management Microservice

The *Rooms Management Microservice* data model is composed of the entities necessary to describe a room within the institute (Fig. 7). A room has a Name, a Capacity and based on what activities rooms are meant to hold, there are different types of rooms described by the Type attribute. This Type attribute has its own model (Fig. 8) containing the names for each type of room in which activities take place.



```
public class Room
{
    3 references
    public int RoomId { get; set; }
    0 references
    public string Name { get; set; }
    0 references
    public int Capacity { get; set; }
    0 references
    public int TypeId { get; set; }
    0 references
    public Type Type { get; set; }
}
```

FIG. 7

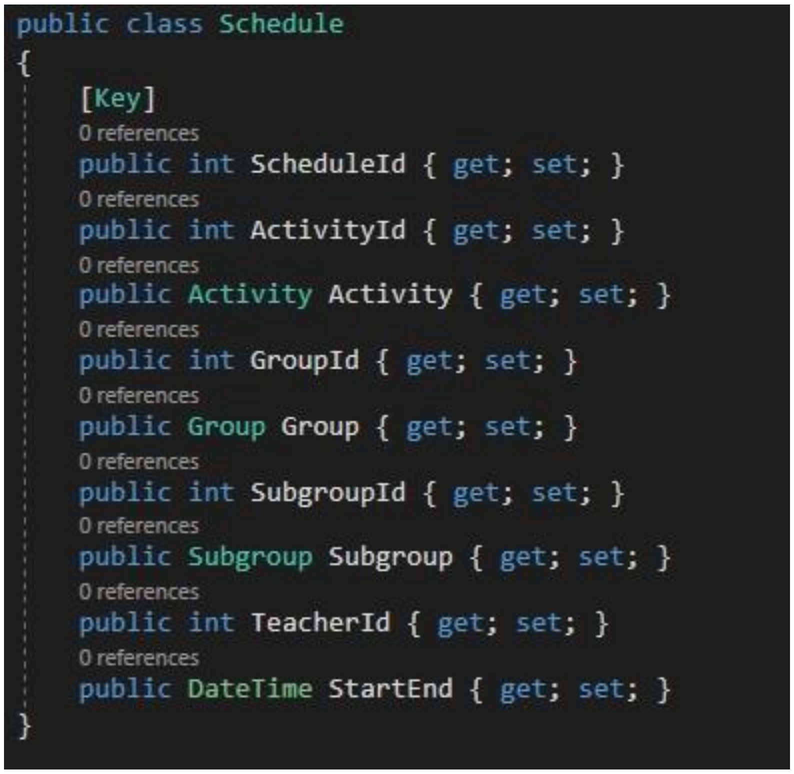


```
public class Type
{
    0 references
    public int TypeId { get; set; }
    0 references
    public string Name { get; set; }
}
```

FIG. 8

Schedule Management Microservice

The *Schedule Management Microservice* data model (Fig. 9) depends on the Activities and Groups Microservice's data model, as, for a entry in the schedule, the activity, group and subgroup need to be specified. Beside those 3 attributes, a Teacher needs to be specified and a start and point.



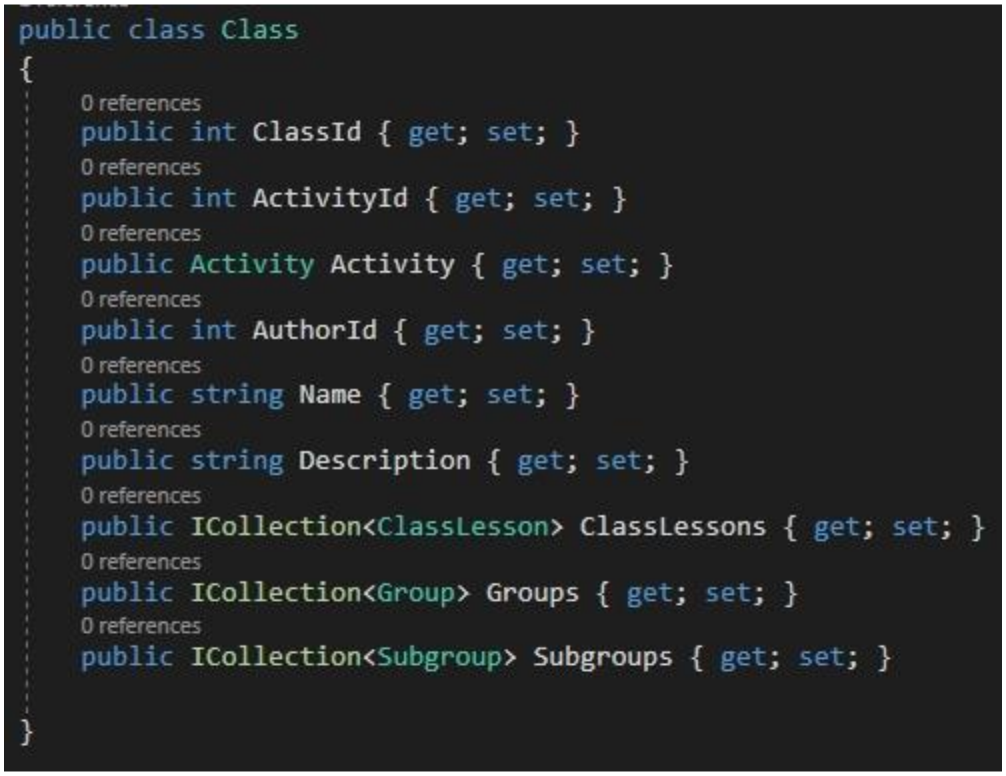
```
public class Schedule
{
    [Key]
    0 references
    public int ScheduleId { get; set; }
    0 references
    public int ActivityId { get; set; }
    0 references
    public Activity Activity { get; set; }
    0 references
    public int GroupId { get; set; }
    0 references
    public Group Group { get; set; }
    0 references
    public int SubgroupId { get; set; }
    0 references
    public Subgroup Subgroup { get; set; }
    0 references
    public int TeacherId { get; set; }
    0 references
    public DateTime StartEnd { get; set; }
}
```

FIG. 9

Teaching and Learning Management Microservice

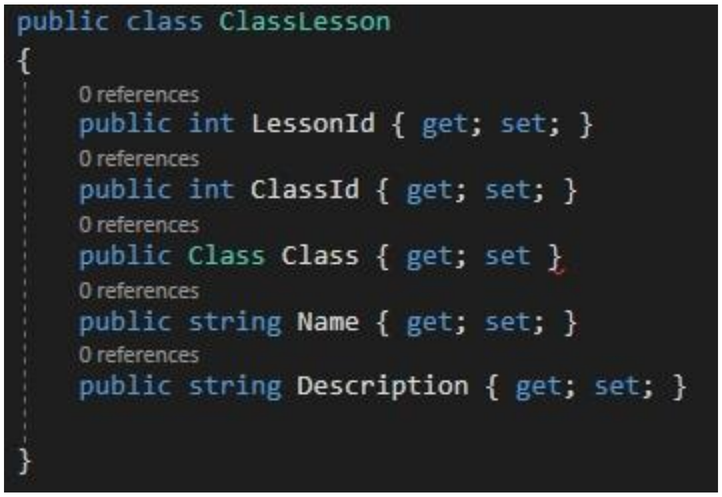
The *Teaching and Learning Management Microservice* data model is built based on two models:

- Class model (Fig. 10) – contains the attributes required to create a class, such as a Name, a Description, the Groups and Subgroups that are taking part in the class and the Class Lessons;
- ClassLesson model (Fig. 11) – has the attributes of the lessons that teachers post in that respective class.



```
public class Class
{
    0 references
    public int ClassId { get; set; }
    0 references
    public int ActivityId { get; set; }
    0 references
    public Activity Activity { get; set; }
    0 references
    public int AuthorId { get; set; }
    0 references
    public string Name { get; set; }
    0 references
    public string Description { get; set; }
    0 references
    public ICollection<ClassLesson> ClassLessons { get; set; }
    0 references
    public ICollection<Group> Groups { get; set; }
    0 references
    public ICollection<Subgroup> Subgroups { get; set; }

}
```

FIG. 10



```
public class ClassLesson
{
    0 references
    public int LessonId { get; set; }
    0 references
    public int ClassId { get; set; }
    0 references
    public Class Class { get; set }
    0 references
    public string Name { get; set; }
    0 references
    public string Description { get; set; }

}
```

FIG. 11

USER INTERFACE

The user interface is simple and easy to use as for a Learning Management System there needs to be available to use only the necessary microservices that a user must have access to in order to get the most benefit out of it.

For every teacher and student there will be available, when using our client application, only a couple of the microservices presented previously. Those microservices are:

- The *Authentication and Authorization Microservice* needed for every user with no exception in order to access the other microservices;
- The *Teaching and Learning Management Microservice* needed for the teacher-student interaction with the lessons that are posted throughout the classes.

The user interface for the other two types of users, the administrator and super user, has all the microservices included:

- *Authentication and Authorization Microservice;*
- *Activities Management Microservice;*
- *Groups Management Microservice;*
- *Rooms Management Microservice;*
- *Schedule Management Microservice;*
- *Teaching and Learning Management Microservice.*

It means that a user, which has one of the two roles mentioned above, has access to view, modify or delete any of the entries available or add new entries.

For authentication, users can only use the login service. The users will be registered by a super user. The role of super user was created to give a user almost the same kind of privileges that an administrator has. This role is actually given to at least one of the representatives of the educational institutes.

APPLICATION IMPLEMENTATION

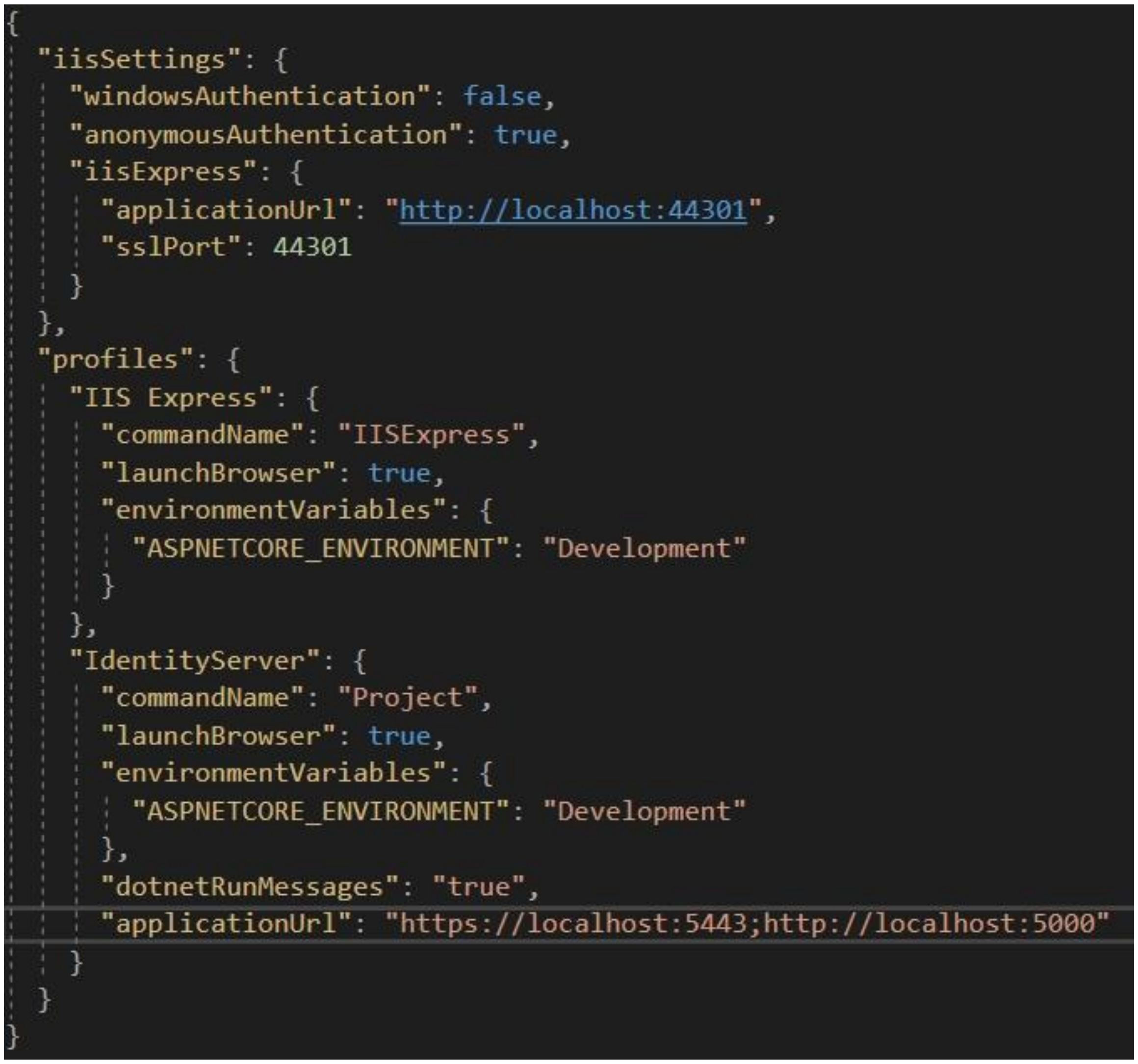
IMPLEMENTED MICROSERVICES

As presented in the previous chapters, there are a total of six microservices, each built around the requirements for a Learning Management System.

The *Authentication and Authorization Microservice* is built around Identity Server 4 and ASP.NET Identity APIs. Identity Server 4 (IDS4) is a critical API for this microservice as we are using its authentication and authorization services. The authorization protocol, entitled OAuth 2.0, provides our web application with specific authorization flows, while the authentication called OpenID Connect is a simple identity layer that works on top of OAuth 2. Those two are protocols that work around using access tokens. Those unique access token are given to users after their identity was verified during the authentication process. With those tokens, users are able to access the web application that the token was issued for.

As there are quite a few types of access tokens, for our application we chose to use the Bearer token as it uses the HTTPS security and a user that possesses it is considered authenticated.

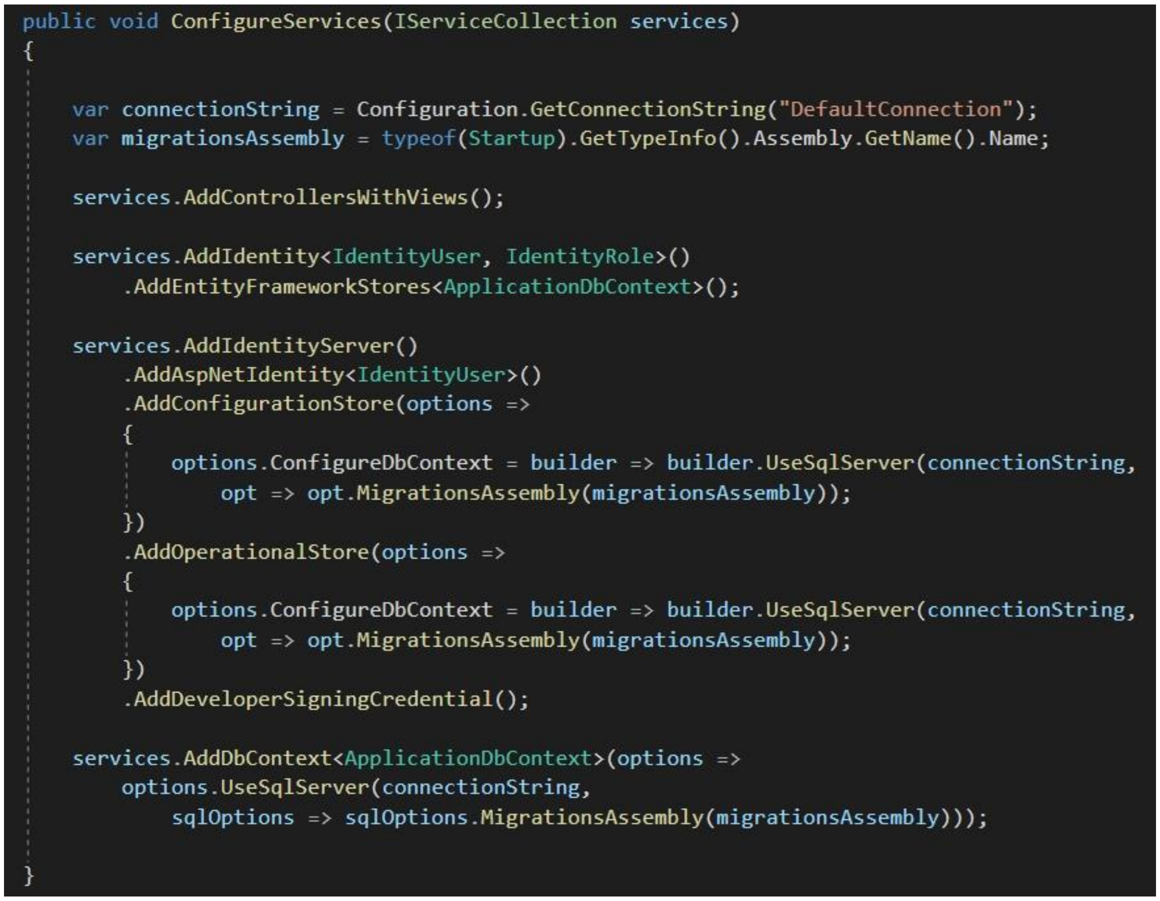
Besides the IDS4 API, this microservice also uses the ASP.NET Identity API which manages users, passwords, user roles, and many more. It works hand in hand with IDS4 as ASP.NET Identity serves to manage only users and IDS4 serves to provide users with the necessary access for the other microservices. In the following screenshots you can see the configuration of this microservice.



```
{
  "iisSettings": {
    "windowsAuthentication": false,
    "anonymousAuthentication": true,
    "iisExpress": {
      "applicationUrl": "http://localhost:44301",
      "sslPort": 44301
    }
  },
  "profiles": {
    "IIS Express": {
      "commandName": "IISExpress",
      "launchBrowser": true,
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development"
      }
    },
    "IdentityServer": {
      "commandName": "Project",
      "launchBrowser": true,
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development"
      },
      "dotnetRunMessages": "true",
      "applicationUrl": "https://localhost:5443;http://localhost:5000"
    }
  }
}
```

FIG. 12

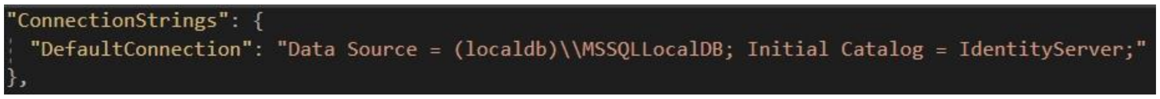
This microservice runs on the launch setting entitled Identity Server and on the URL *https://localhost:5443*. The other microservices that are running can be accessed only based on the authority that is requested from the URL on which this microservice runs.



```csharp
public void ConfigureServices(IServiceCollection services)
{

    var connectionString = Configuration.GetConnectionString("DefaultConnection");
    var migrationsAssembly = typeof(Startup).GetTypeInfo().Assembly.GetName().Name;

    services.AddControllersWithViews();

    services.AddIdentity<IdentityUser, IdentityRole>()
        .AddEntityFrameworkStores<ApplicationDbContext>();

    services.AddIdentityServer()
        .AddAspNetIdentity<IdentityUser>()
        .AddConfigurationStore(options =>
        {
            options.ConfigureDbContext = builder => builder.UseSqlServer(connectionString,
                opt => opt.MigrationsAssembly(migrationsAssembly));
        })
        .AddOperationalStore(options =>
        {
            options.ConfigureDbContext = builder => builder.UseSqlServer(connectionString,
                opt => opt.MigrationsAssembly(migrationsAssembly));
        })
        .AddDeveloperSigningCredential();

    services.AddDbContext<ApplicationDbContext>(options =>
        options.UseSqlServer(connectionString,
            sqlOptions => sqlOptions.MigrationsAssembly(migrationsAssembly)));

}
```

FIG. 13

In the above screenshot all of the services that this microservice uses are configured. The variable *connectionString* represents the string used to connect to the database that this microservice uses, string which is provided by the attribute *DefaultConnection* and which can be visualized in the Fig. 14.

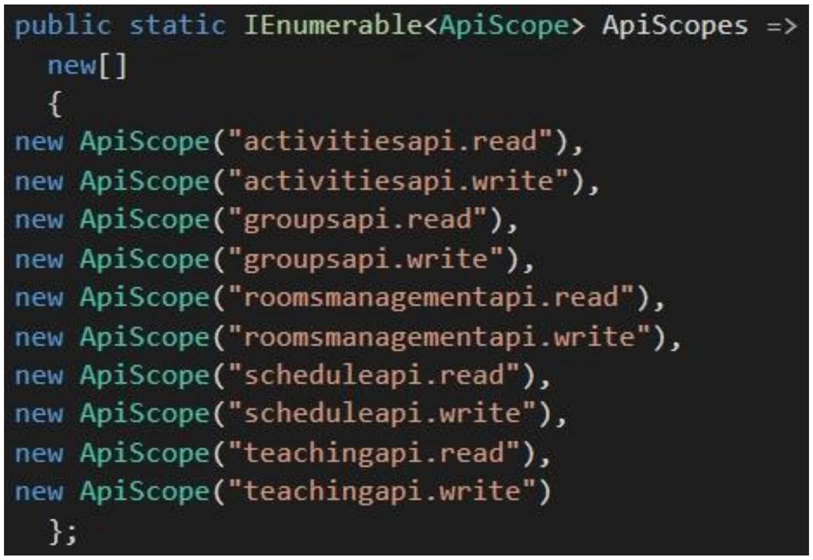


```json
"ConnectionStrings": {
  "DefaultConnection": "Data Source = (localdb)\\MSSQLLocalDB; Initial Catalog = IdentityServer;"
},
```

FIG. 14

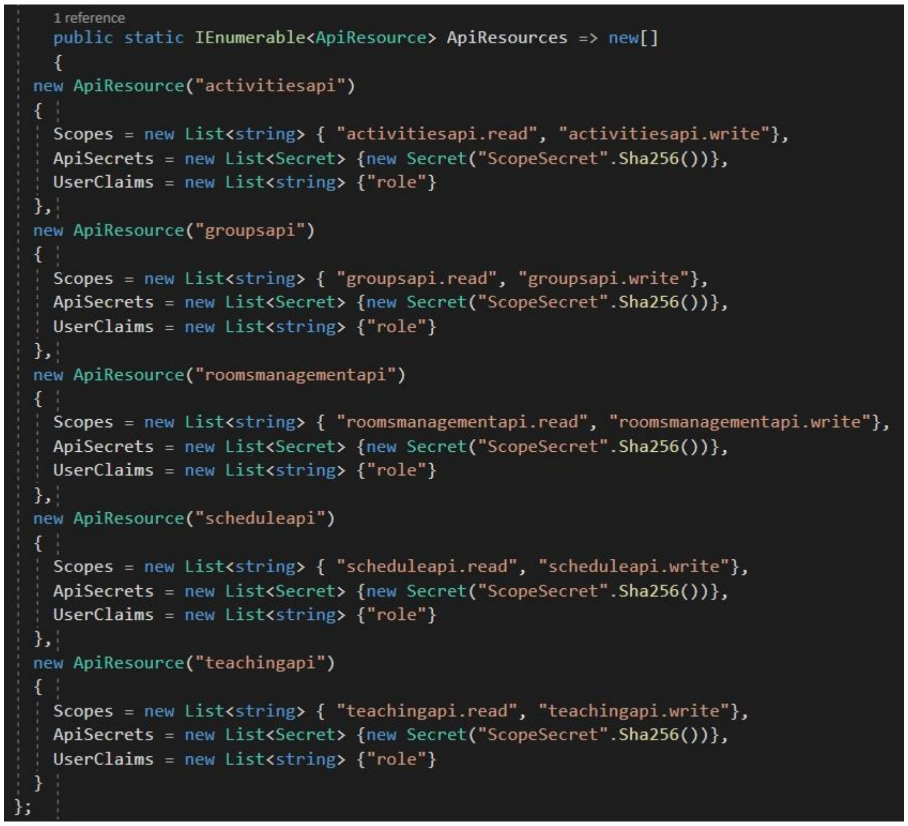
The addition of the *Configuration Store* is useful to load the API resources of the other microservices from the database instead of loading them from memory, while the addition of the *Operational Store* helps to load the persisted grants, which include the tokens and the user consents, directly from the database.

The API resources and scopes can be seen in the following two screenshots. Those scopes and resources are needed in order for the IDS4 to recognize and authorize the implemented microservices.



```
public static IEnumerable<ApiScope> ApiScopes =>
  new[]
  {
new ApiScope("activitiesapi.read"),
new ApiScope("activitiesapi.write"),
new ApiScope("groupsapi.read"),
new ApiScope("groupsapi.write"),
new ApiScope("roomsmanagementapi.read"),
new ApiScope("roomsmanagementapi.write"),
new ApiScope("scheduleapi.read"),
new ApiScope("scheduleapi.write"),
new ApiScope("teachingapi.read"),
new ApiScope("teachingapi.write")
  };
```

FIG. 15

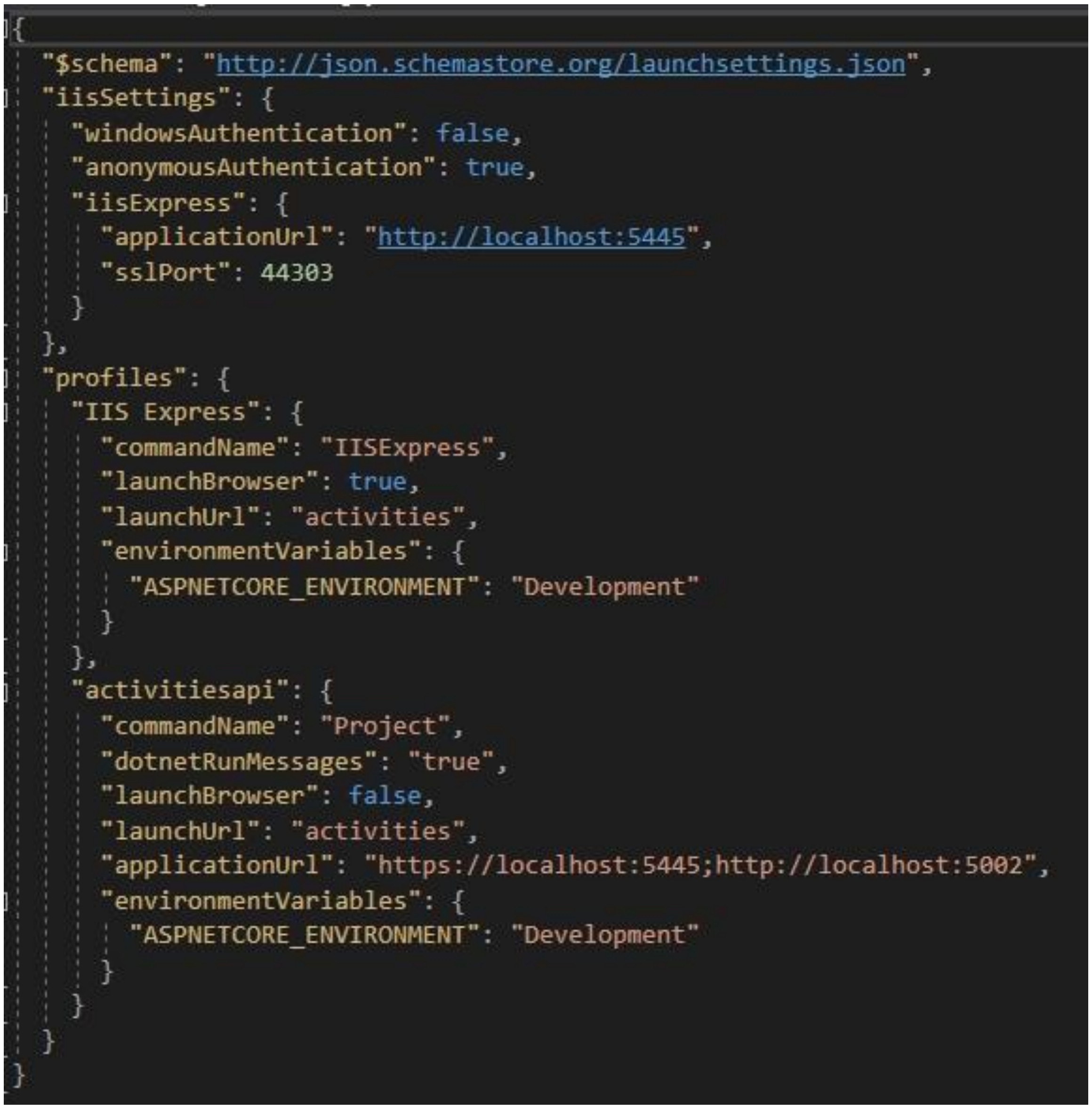


```
   1 reference
   public static IEnumerable<ApiResource> ApiResources => new[]
   {
 new ApiResource("activitiesapi")
 {
   Scopes = new List<string> { "activitiesapi.read", "activitiesapi.write"},
   ApiSecrets = new List<Secret> {new Secret("ScopeSecret".Sha256())},
   UserClaims = new List<string> {"role"}
 },
 new ApiResource("groupsapi")
 {
   Scopes = new List<string> { "groupsapi.read", "groupsapi.write"},
   ApiSecrets = new List<Secret> {new Secret("ScopeSecret".Sha256())},
   UserClaims = new List<string> {"role"}
 },
 new ApiResource("roomsmanagementapi")
 {
   Scopes = new List<string> { "roomsmanagementapi.read", "roomsmanagementapi.write"},
   ApiSecrets = new List<Secret> {new Secret("ScopeSecret".Sha256())},
   UserClaims = new List<string> {"role"}
 },
 new ApiResource("scheduleapi")
 {
   Scopes = new List<string> { "scheduleapi.read", "scheduleapi.write"},
   ApiSecrets = new List<Secret> {new Secret("ScopeSecret".Sha256())},
   UserClaims = new List<string> {"role"}
 },
 new ApiResource("teachingapi")
 {
   Scopes = new List<string> { "teachingapi.read", "teachingapi.write"},
   ApiSecrets = new List<Secret> {new Secret("ScopeSecret".Sha256())},
   UserClaims = new List<string> {"role"}
 }
};
```

FIG. 16

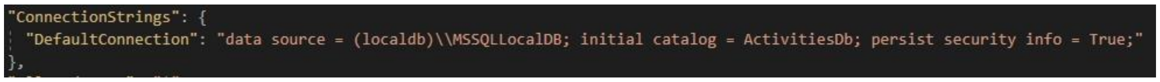
The *Activities Microservice,* as said before, is the microservice which supervises the curricular activities and has the ability to visualize, modify, remove or add them.

The launch and app settings are as follows:



```
{
  "$schema": "http://json.schemastore.org/launchsettings.json",
  "iisSettings": {
    "windowsAuthentication": false,
    "anonymousAuthentication": true,
    "iisExpress": {
      "applicationUrl": "http://localhost:5445",
      "sslPort": 44303
    }
  },
  "profiles": {
    "IIS Express": {
      "commandName": "IISExpress",
      "launchBrowser": true,
      "launchUrl": "activities",
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development"
      }
    },
    "activitiesapi": {
      "commandName": "Project",
      "dotnetRunMessages": "true",
      "launchBrowser": false,
      "launchUrl": "activities",
      "applicationUrl": "https://localhost:5445;http://localhost:5002",
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development"
      }
    }
  }
}
```

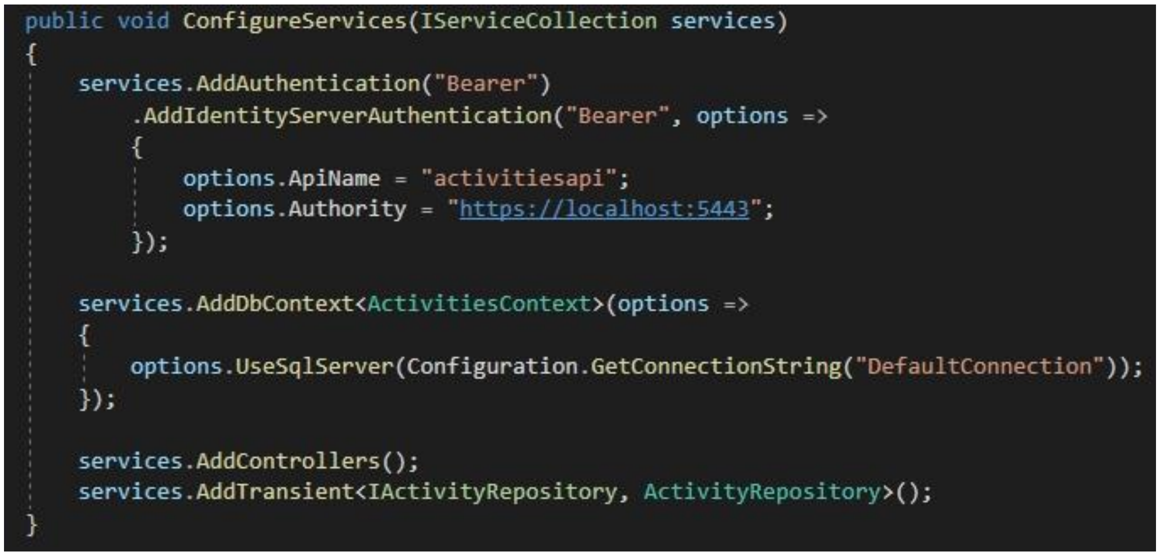
FIG. 17



```
"ConnectionStrings": {
  "DefaultConnection": "data source = (localdb)\\MSSQLLocalDB; initial catalog = ActivitiesDb; persist security info = True;"
},
```

FIG. 18

When the microservice is running, it runs on the launch setting *activitiesapi* and the URL *https://localhost:5445/activities*, but as it isn't necessary to actually run it in the browser, the attribute *launchBrowser* is set to false. The connection to the database is established with the with the *DefaultConnection* string as shown in the fig. 18.



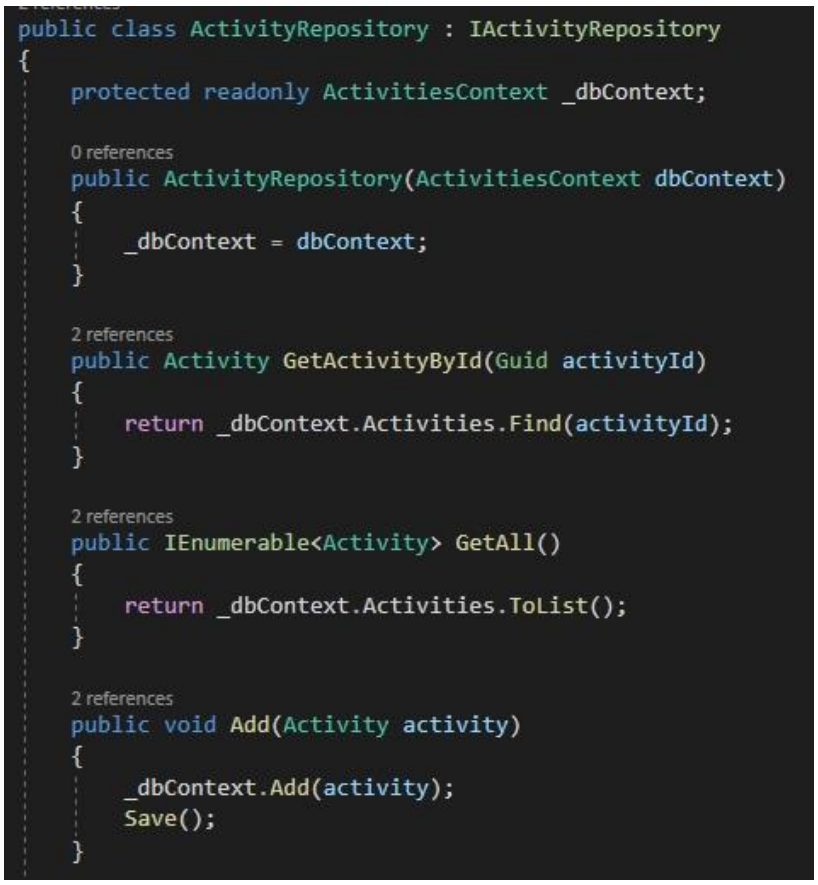
```
public void ConfigureServices(IServiceCollection services)
{
    services.AddAuthentication("Bearer")
        .AddIdentityServerAuthentication("Bearer", options =>
        {
            options.ApiName = "activitiesapi";
            options.Authority = "https://localhost:5443";
        });

    services.AddDbContext<ActivitiesContext>(options =>
    {
        options.UseSqlServer(Configuration.GetConnectionString("DefaultConnection"));
    });

    services.AddControllers();
    services.AddTransient<IActivityRepository, ActivityRepository>();
}
```

FIG. 19

In the fig. 19 we have the configuration of the services that this microservice uses. As mentioned previously, for authentication the Bearer token is used together with the addition of the API Name, which is the name of the current microservice, and with the URL of the authority *Authentication Microservice*.

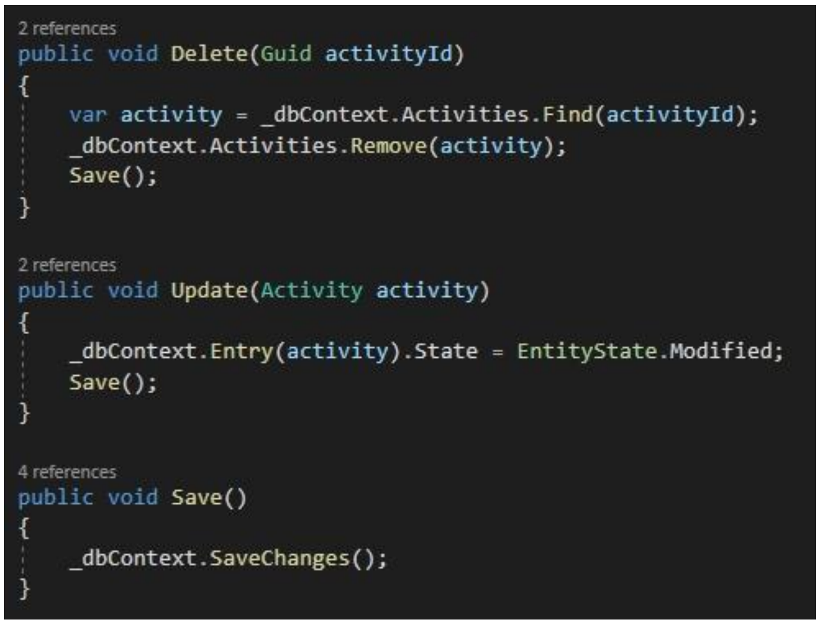
Next up, we have the *ActivityRepository (Fig. 20)* together with its interface (Fig. 21) and the controller of this microservice, *ActivitiesController* (Fig. 22).

In the *ActivityRepository* we have implemented the functions necessary to use in this microservice's controller. The *ActivitiesController* consists of CRUD operations and it is able to make modifications to the existent entities that are within the *Activity* database, or to add new entries to the database.



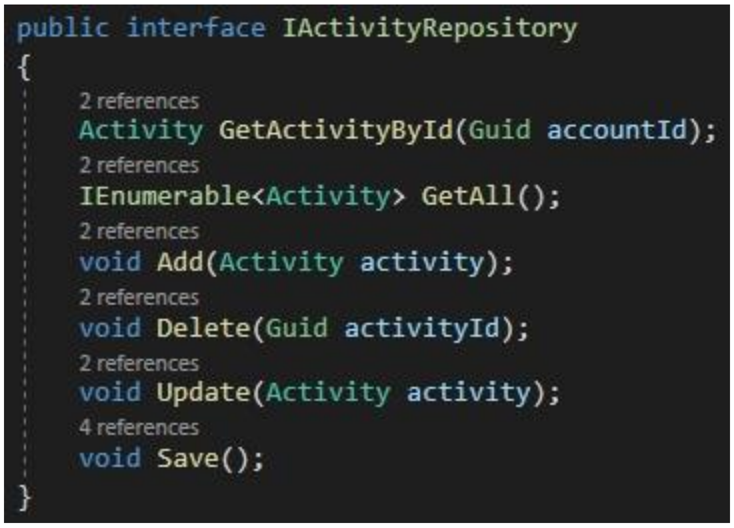
```
public class ActivityRepository : IActivityRepository
{
    protected readonly ActivitiesContext _dbContext;

    0 references
    public ActivityRepository(ActivitiesContext dbContext)
    {
        _dbContext = dbContext;
    }

    2 references
    public Activity GetActivityById(Guid activityId)
    {
        return _dbContext.Activities.Find(activityId);
    }

    2 references
    public IEnumerable<Activity> GetAll()
    {
        return _dbContext.Activities.ToList();
    }

    2 references
    public void Add(Activity activity)
    {
        _dbContext.Add(activity);
        Save();
    }
```

FIG. 20A



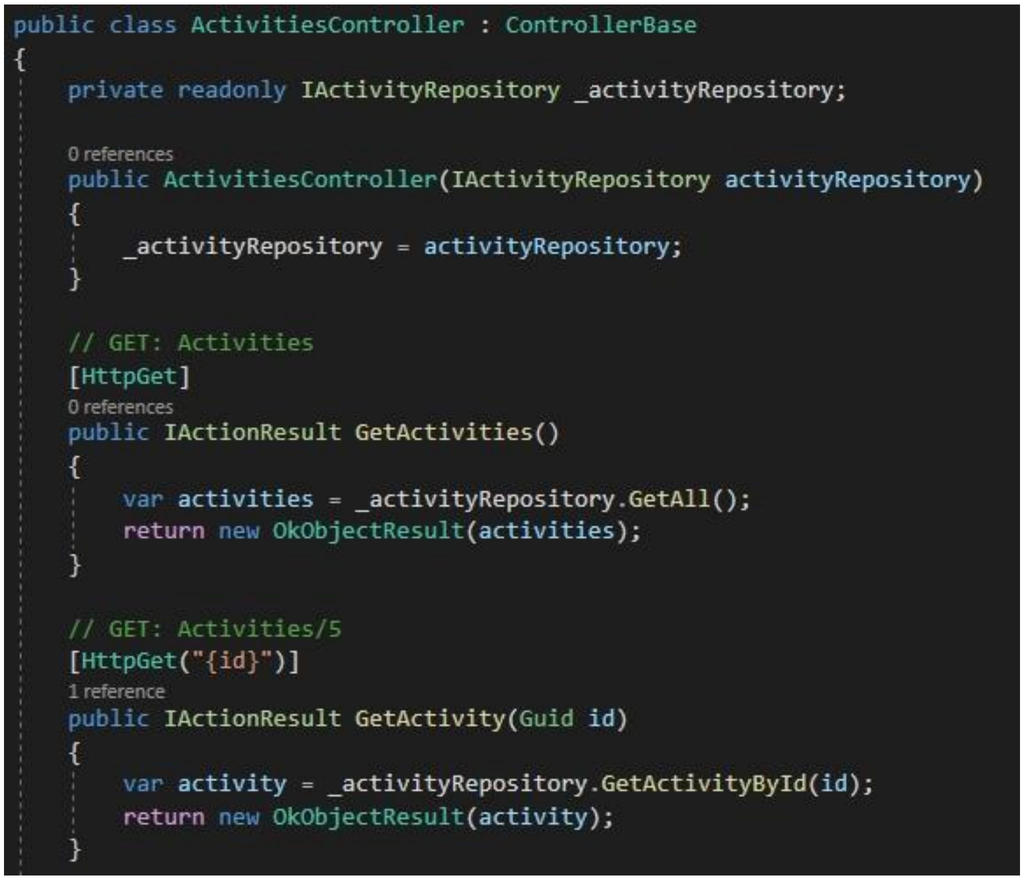
```
2 references
public void Delete(Guid activityId)
{
    var activity = _dbContext.Activities.Find(activityId);
    _dbContext.Activities.Remove(activity);
    Save();
}

2 references
public void Update(Activity activity)
{
    _dbContext.Entry(activity).State = EntityState.Modified;
    Save();
}

4 references
public void Save()
{
    _dbContext.SaveChanges();
}
```

FIG. 20B



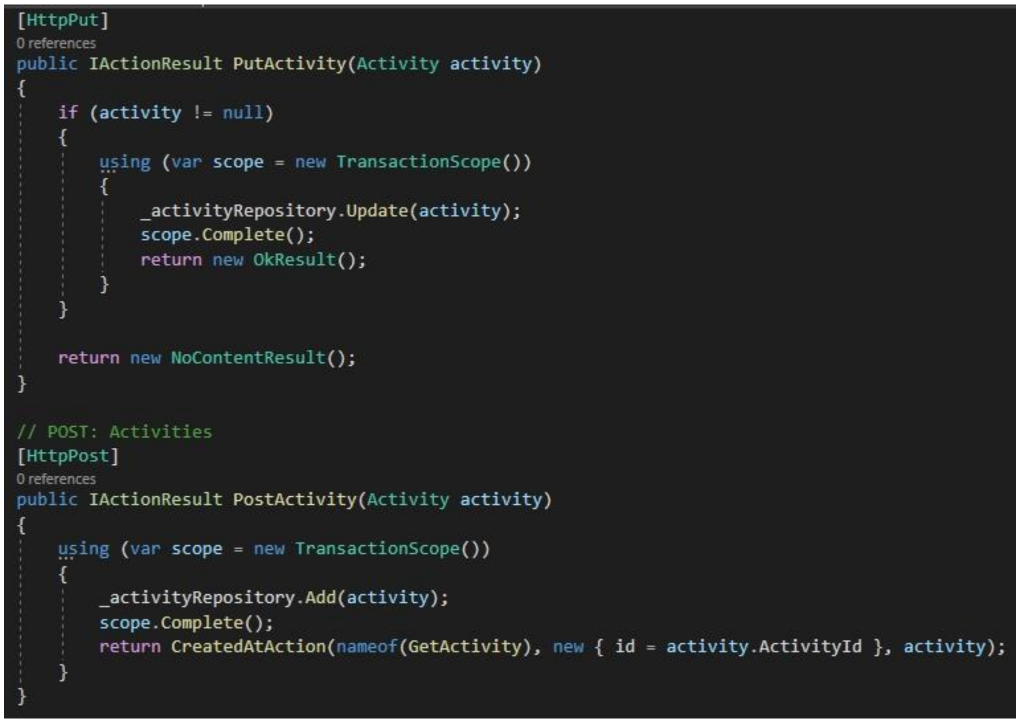
```
public interface IActivityRepository
{
    2 references
    Activity GetActivityById(Guid accountId);
    2 references
    IEnumerable<Activity> GetAll();
    2 references
    void Add(Activity activity);
    2 references
    void Delete(Guid activityId);
    2 references
    void Update(Activity activity);
    4 references
    void Save();
}
```

FIG. 21



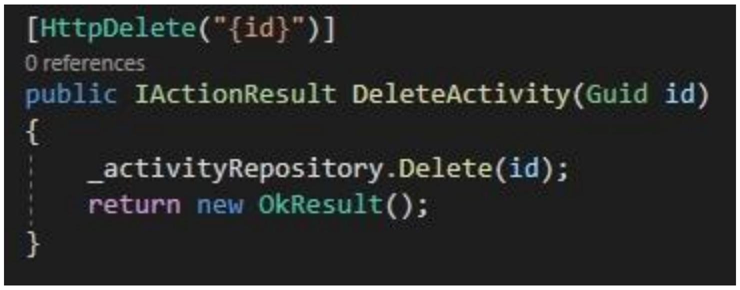
```
public class ActivitiesController : ControllerBase
{
    private readonly IActivityRepository _activityRepository;

    0 references
    public ActivitiesController(IActivityRepository activityRepository)
    {
        _activityRepository = activityRepository;
    }

    // GET: Activities
    [HttpGet]
    0 references
    public IActionResult GetActivities()
    {
        var activities = _activityRepository.GetAll();
        return new OkObjectResult(activities);
    }

    // GET: Activities/5
    [HttpGet("{id}")]
    1 reference
    public IActionResult GetActivity(Guid id)
    {
        var activity = _activityRepository.GetActivityById(id);
        return new OkObjectResult(activity);
    }
```

FIG. 22A



```
[HttpPut]
0 references
public IActionResult PutActivity(Activity activity)
{
    if (activity != null)
    {
        using (var scope = new TransactionScope())
        {
            _activityRepository.Update(activity);
            scope.Complete();
            return new OkResult();
        }
    }

    return new NoContentResult();
}

// POST: Activities
[HttpPost]
0 references
public IActionResult PostActivity(Activity activity)
{
    using (var scope = new TransactionScope())
    {
        _activityRepository.Add(activity);
        scope.Complete();
        return CreatedAtAction(nameof(GetActivity), new { id = activity.ActivityId }, activity);
    }
}
```

FIG. 22B

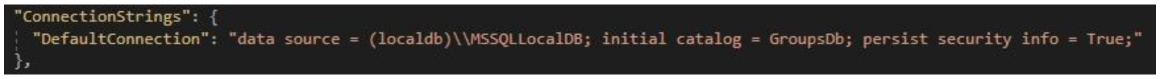


```
[HttpDelete("{id}")]
0 references
public IActionResult DeleteActivity(Guid id)
{
    _activityRepository.Delete(id);
    return new OkResult();
}
```

FIG. 22C

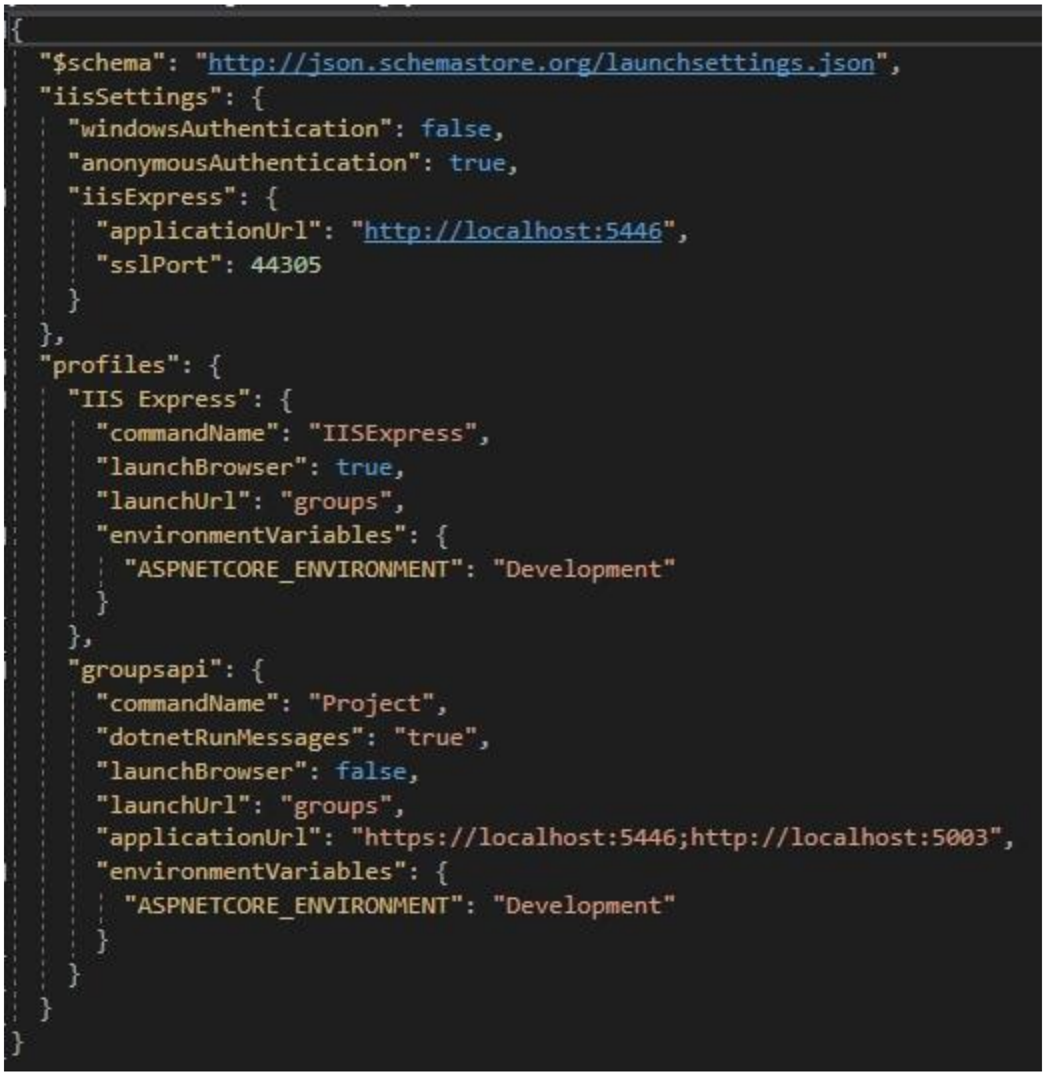
As the *Groups Microservice* is the microservice that manages the groups and subgroups, it is split into three parts, one part representing the groups, another representing the subgroups, and the last one representing the specializations.

In the next two figures you can see the configuration on which this microservice is set up. It runs on the URL *https://localhost:5446*, and the connection to the database is established with the help of the default *Connection String.*



```
"ConnectionStrings": {
  "DefaultConnection": "data source = (localdb)\\MSSQLLocalDB; initial catalog = GroupsDb; persist security info = True;"
},
```

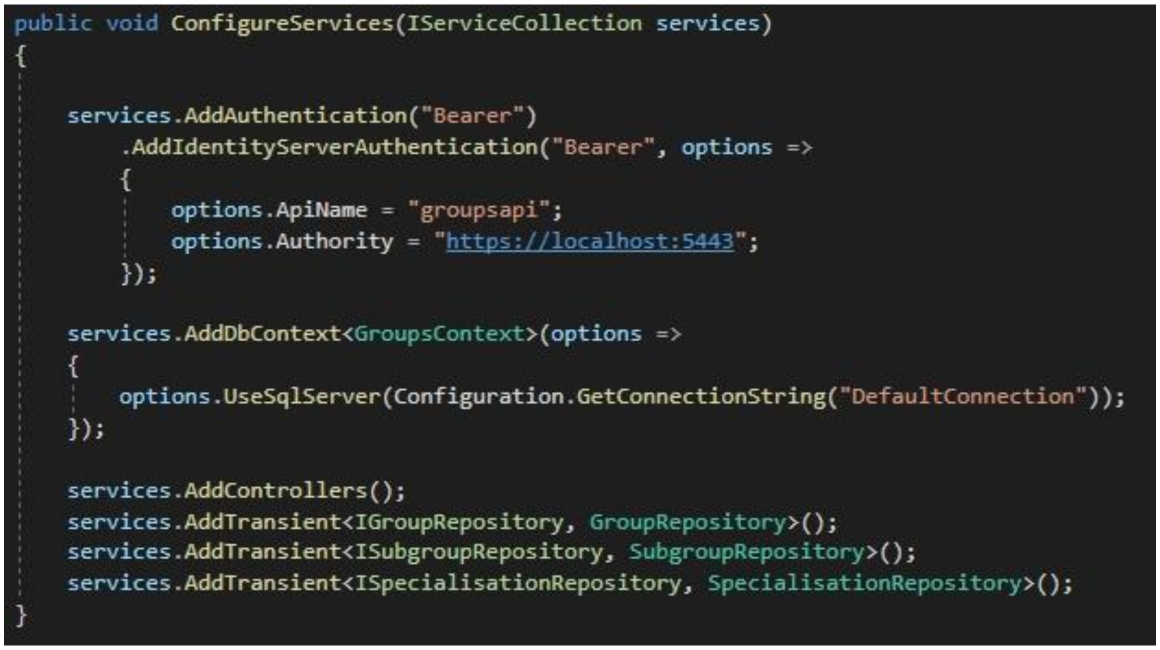
FIG. 23



```
{
  "$schema": "http://json.schemastore.org/launchsettings.json",
  "iisSettings": {
    "windowsAuthentication": false,
    "anonymousAuthentication": true,
    "iisExpress": {
      "applicationUrl": "http://localhost:5446",
      "sslPort": 44305
    }
  },
  "profiles": {
    "IIS Express": {
      "commandName": "IISExpress",
      "launchBrowser": true,
      "launchUrl": "groups",
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development"
      }
    },
    "groupsapi": {
      "commandName": "Project",
      "dotnetRunMessages": "true",
      "launchBrowser": false,
      "launchUrl": "groups",
      "applicationUrl": "https://localhost:5446;http://localhost:5003",
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development"
      }
    }
  }
}
```

FIG. 24

The configuration of the services that this microservice uses are:

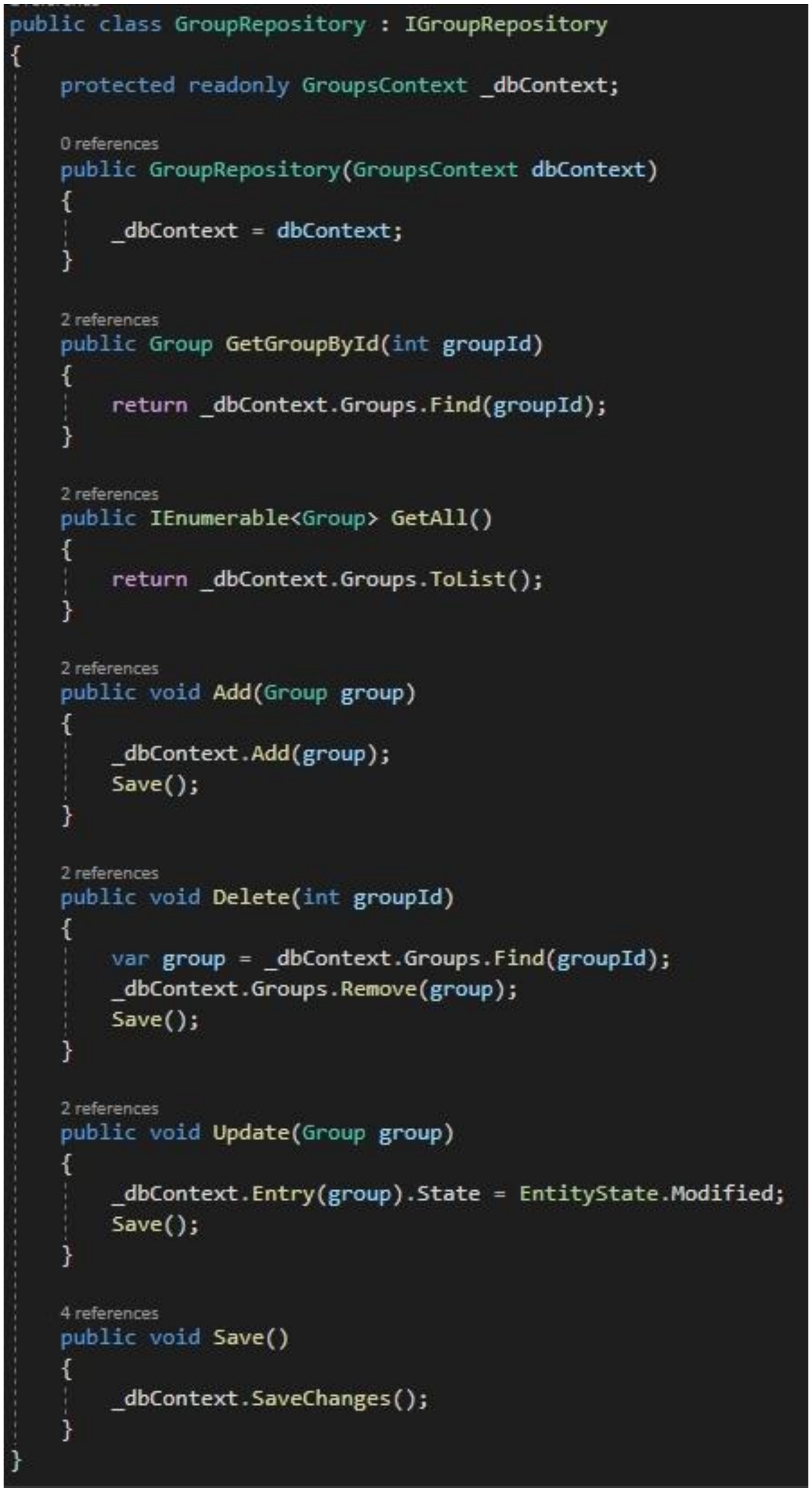


```
public void ConfigureServices(IServiceCollection services)
{

    services.AddAuthentication("Bearer")
        .AddIdentityServerAuthentication("Bearer", options =>
        {
            options.ApiName = "groupsapi";
            options.Authority = "https://localhost:5443";
        });

    services.AddDbContext<GroupsContext>(options =>
    {
        options.UseSqlServer(Configuration.GetConnectionString("DefaultConnection"));
    });

    services.AddControllers();
    services.AddTransient<IGroupRepository, GroupRepository>();
    services.AddTransient<ISubgroupRepository, SubgroupRepository>();
    services.AddTransient<ISpecialisationRepository, SpecialisationRepository>();
}
```

FIG. 25

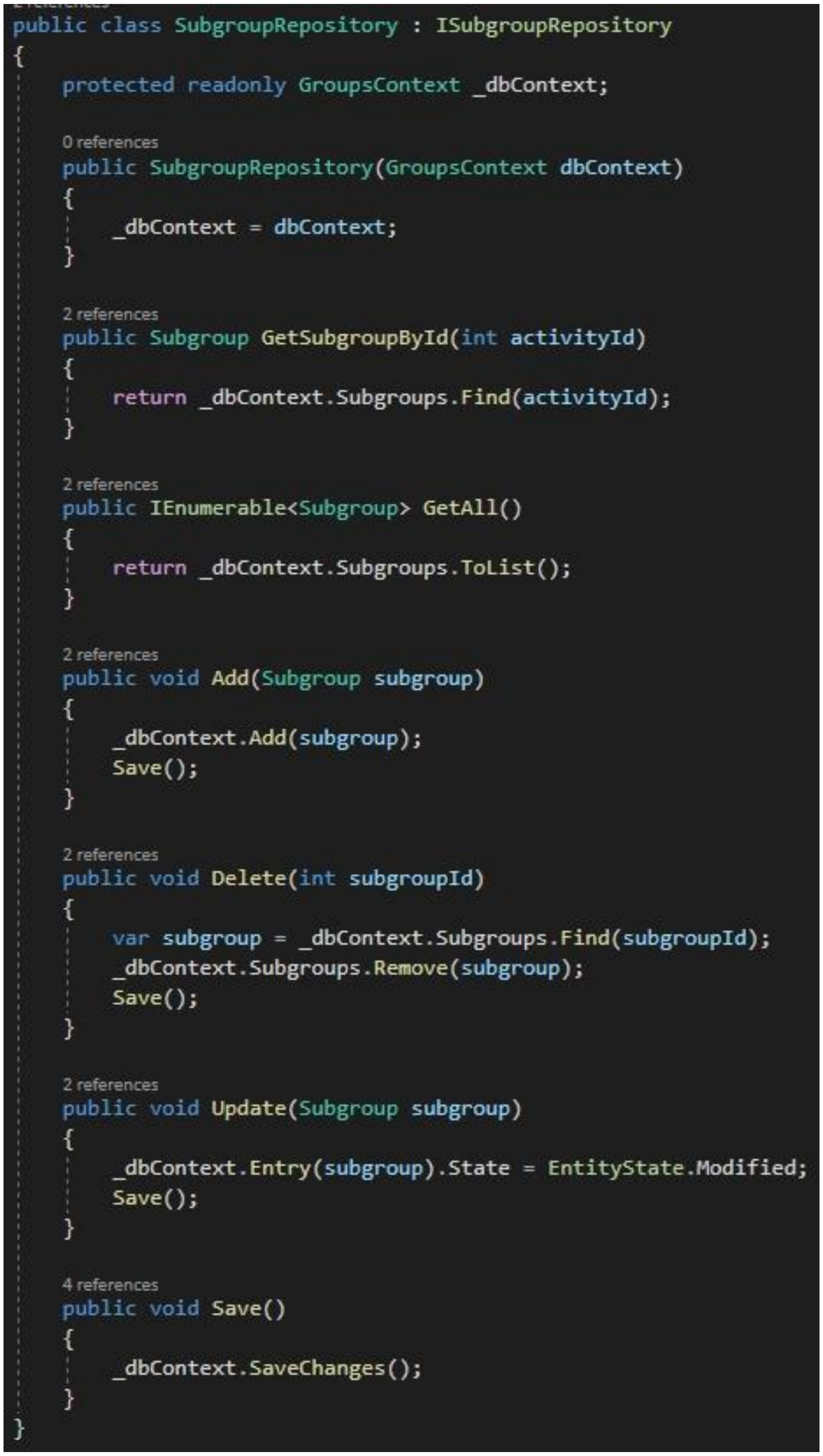
In the fig. 25 we have the configuration of the services that this microservice uses. As this is a basic setup for every microservice, for authentication the Bearer token is used together with the addition of the API Name, which is the name of the current microservice, and with the URL of the authority called *Authentication Microservice*.

For every of the three parts that are involved in this microservice, we have separate controllers and repositories. In the repositories have been implemented the functions necessary to use in this microservice’s controllers. The controllers consist of CRUD operations and are able to make modifications to the existent entities that are within each database, or to add new entries to the databases.



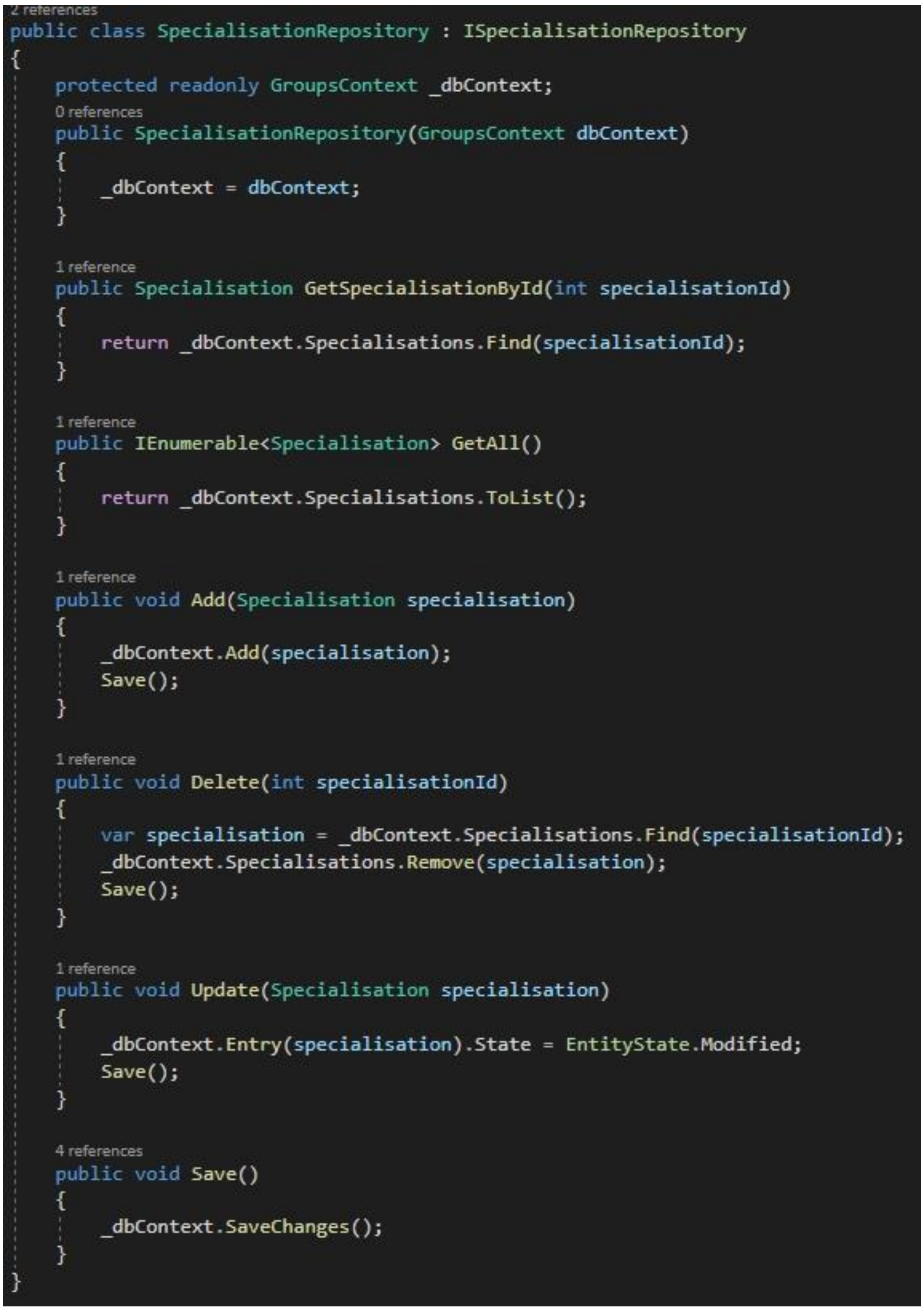
```
public class GroupRepository : IGroupRepository
{
    protected readonly GroupsContext _dbContext;

    0 references
    public GroupRepository(GroupsContext dbContext)
    {
        _dbContext = dbContext;
    }

    2 references
    public Group GetGroupById(int groupId)
    {
        return _dbContext.Groups.Find(groupId);
    }

    2 references
    public IEnumerable<Group> GetAll()
    {
        return _dbContext.Groups.ToList();
    }

    2 references
    public void Add(Group group)
    {
        _dbContext.Add(group);
        Save();
    }

    2 references
    public void Delete(int groupId)
    {
        var group = _dbContext.Groups.Find(groupId);
        _dbContext.Groups.Remove(group);
        Save();
    }

    2 references
    public void Update(Group group)
    {
        _dbContext.Entry(group).State = EntityState.Modified;
        Save();
    }

    4 references
    public void Save()
    {
        _dbContext.SaveChanges();
    }
}
```

FIG. 26



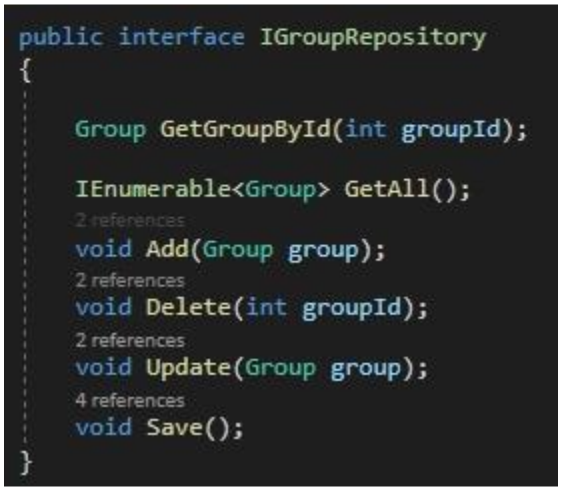
```
public class SubgroupRepository : ISubgroupRepository
{
    protected readonly GroupsContext _dbContext;

    0 references
    public SubgroupRepository(GroupsContext dbContext)
    {
        _dbContext = dbContext;
    }

    2 references
    public Subgroup GetSubgroupById(int activityId)
    {
        return _dbContext.Subgroups.Find(activityId);
    }

    2 references
    public IEnumerable<Subgroup> GetAll()
    {
        return _dbContext.Subgroups.ToList();
    }

    2 references
    public void Add(Subgroup subgroup)
    {
        _dbContext.Add(subgroup);
        Save();
    }

    2 references
    public void Delete(int subgroupId)
    {
        var subgroup = _dbContext.Subgroups.Find(subgroupId);
        _dbContext.Subgroups.Remove(subgroup);
        Save();
    }

    2 references
    public void Update(Subgroup subgroup)
    {
        _dbContext.Entry(subgroup).State = EntityState.Modified;
        Save();
    }

    4 references
    public void Save()
    {
        _dbContext.SaveChanges();
    }
}
```

FIG. 27



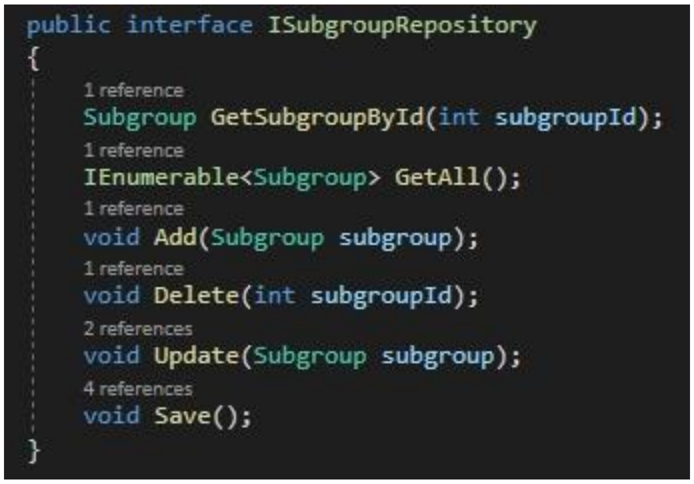
```
2 references
public class SpecialisationRepository : ISpecialisationRepository
{
    protected readonly GroupsContext _dbContext;
    0 references
    public SpecialisationRepository(GroupsContext dbContext)
    {
        _dbContext = dbContext;
    }

    1 reference
    public Specialisation GetSpecialisationById(int specialisationId)
    {
        return _dbContext.Specialisations.Find(specialisationId);
    }

    1 reference
    public IEnumerable<Specialisation> GetAll()
    {
        return _dbContext.Specialisations.ToList();
    }

    1 reference
    public void Add(Specialisation specialisation)
    {
        _dbContext.Add(specialisation);
        Save();
    }

    1 reference
    public void Delete(int specialisationId)
    {
        var specialisation = _dbContext.Specialisations.Find(specialisationId);
        _dbContext.Specialisations.Remove(specialisation);
        Save();
    }

    1 reference
    public void Update(Specialisation specialisation)
    {
        _dbContext.Entry(specialisation).State = EntityState.Modified;
        Save();
    }

    4 references
    public void Save()
    {
        _dbContext.SaveChanges();
    }
}
```

FIG. 28



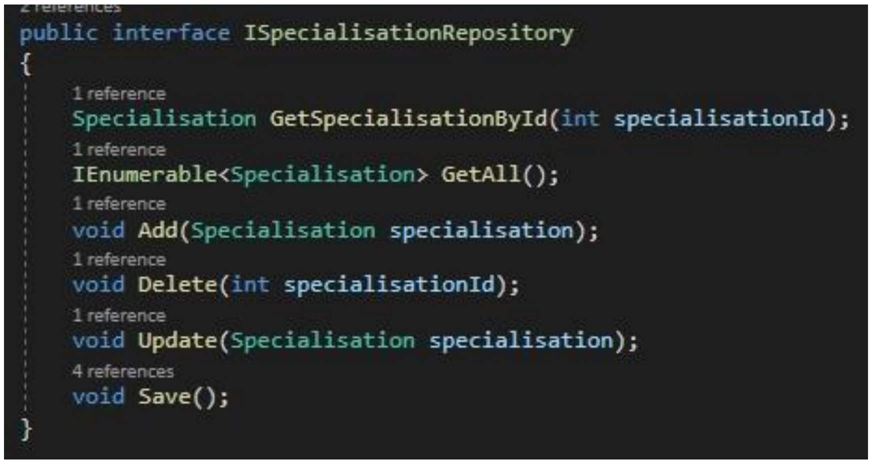
```
public interface IGroupRepository
{

    Group GetGroupById(int groupId);

    IEnumerable<Group> GetAll();
    2 references
    void Add(Group group);
    2 references
    void Delete(int groupId);
    2 references
    void Update(Group group);
    4 references
    void Save();
}
```

FIG. 29



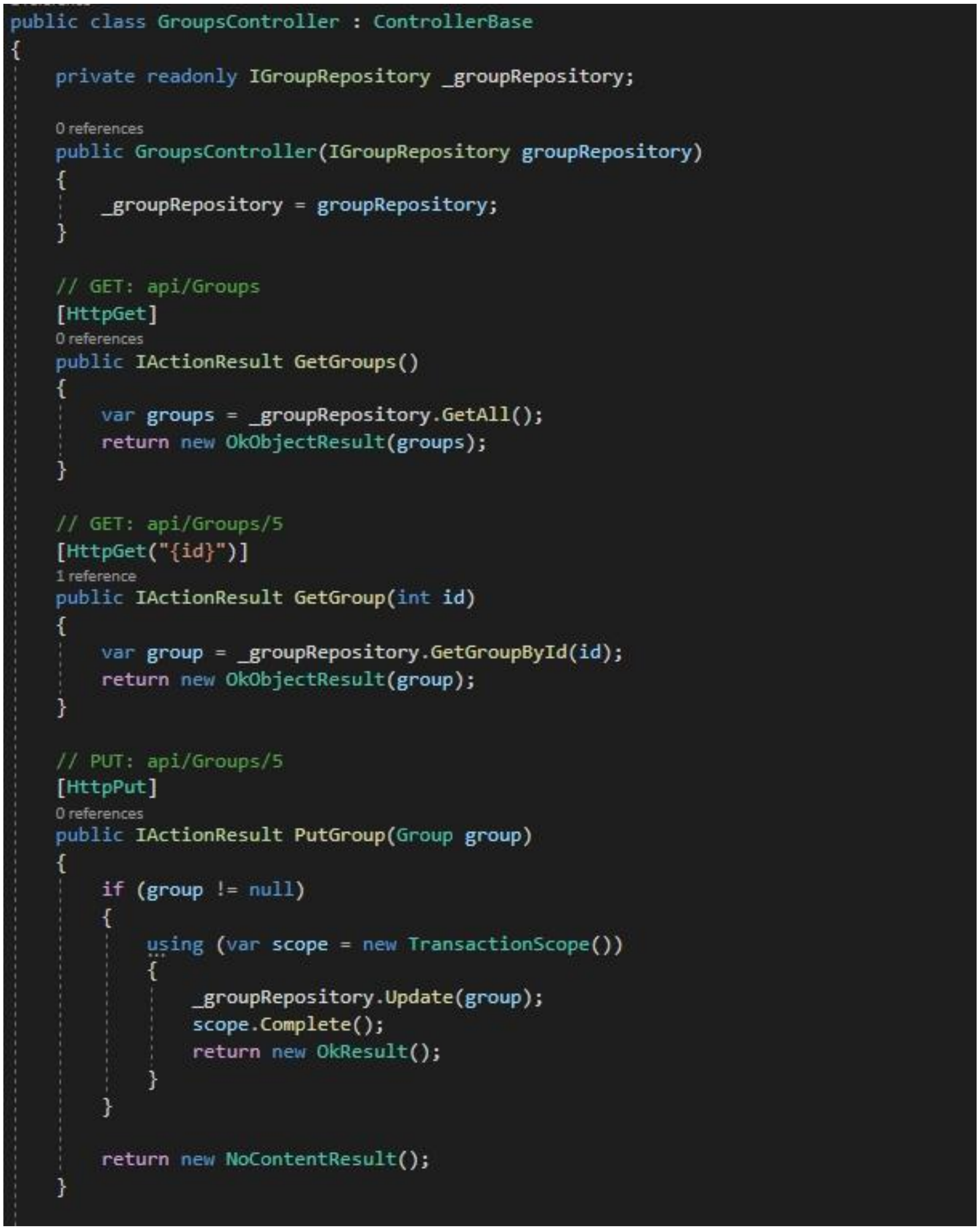
```
public interface ISubgroupRepository
{
    1 reference
    Subgroup GetSubgroupById(int subgroupId);
    1 reference
    IEnumerable<Subgroup> GetAll();
    1 reference
    void Add(Subgroup subgroup);
    1 reference
    void Delete(int subgroupId);
    2 references
    void Update(Subgroup subgroup);
    4 references
    void Save();
}
```

FIG. 30



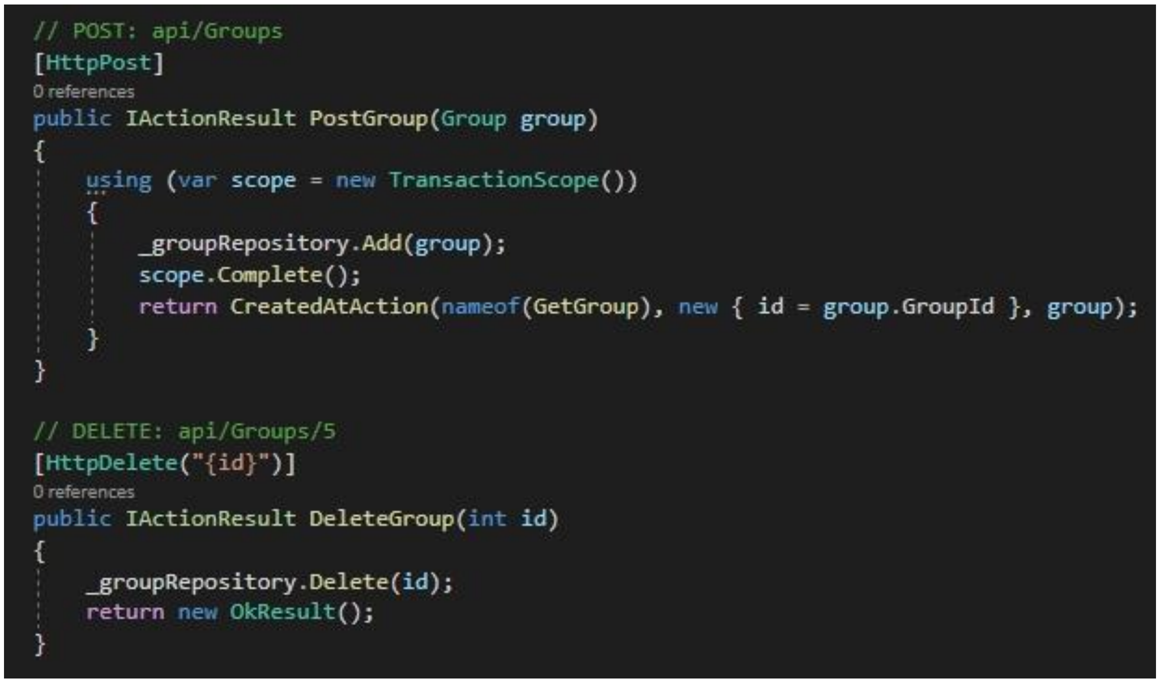
```
2 references
public interface ISpecialisationRepository
{
    1 reference
    Specialisation GetSpecialisationById(int specialisationId);
    1 reference
    IEnumerable<Specialisation> GetAll();
    1 reference
    void Add(Specialisation specialisation);
    1 reference
    void Delete(int specialisationId);
    1 reference
    void Update(Specialisation specialisation);
    4 references
    void Save();
}
```

FIG. 31



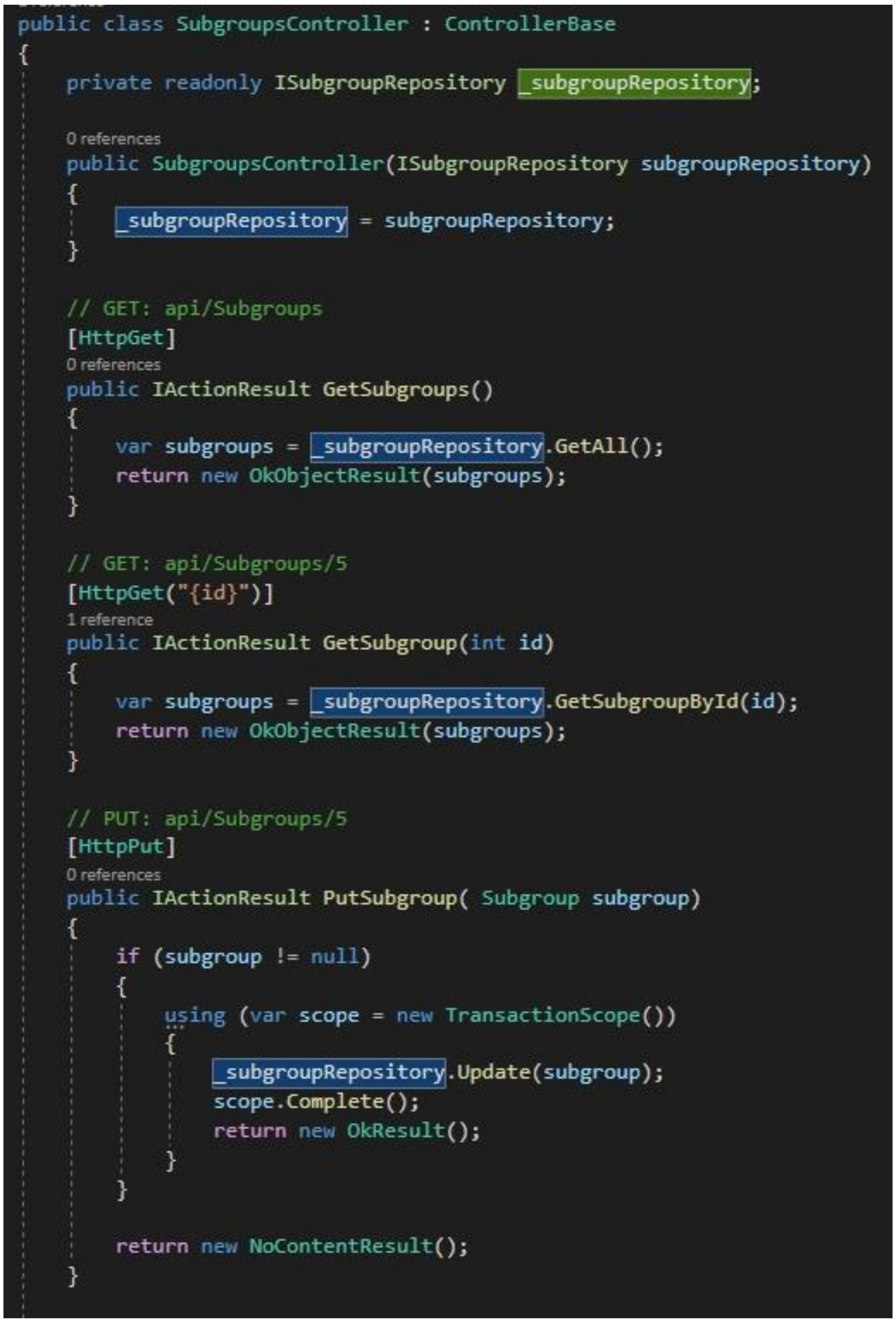
```
public class GroupsController : ControllerBase
{
    private readonly IGroupRepository _groupRepository;

    0 references
    public GroupsController(IGroupRepository groupRepository)
    {
        _groupRepository = groupRepository;
    }

    // GET: api/Groups
    [HttpGet]
    0 references
    public IActionResult GetGroups()
    {
        var groups = _groupRepository.GetAll();
        return new OkObjectResult(groups);
    }

    // GET: api/Groups/5
    [HttpGet("{id}")]
    1 reference
    public IActionResult GetGroup(int id)
    {
        var group = _groupRepository.GetGroupById(id);
        return new OkObjectResult(group);
    }

    // PUT: api/Groups/5
    [HttpPut]
    0 references
    public IActionResult PutGroup(Group group)
    {
        if (group != null)
        {
            using (var scope = new TransactionScope())
            {
                _groupRepository.Update(group);
                scope.Complete();
                return new OkResult();
            }
        }

        return new NoContentResult();
    }
```

FIG. 32



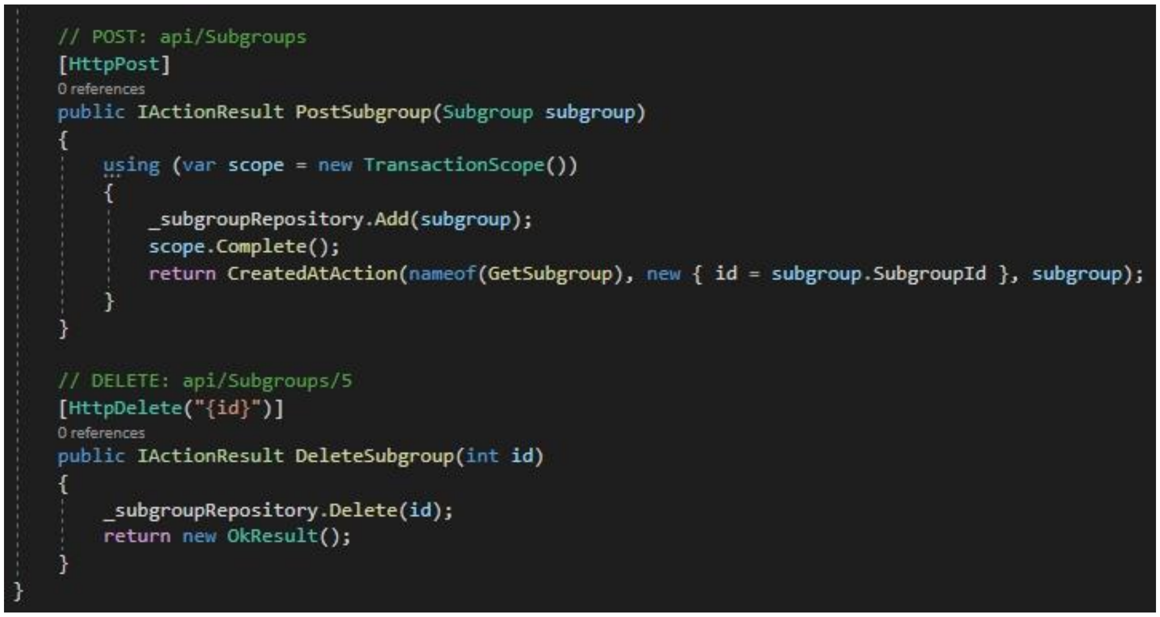
```
// POST: api/Groups
[HttpPost]
0 references
public IActionResult PostGroup(Group group)
{
    using (var scope = new TransactionScope())
    {
        _groupRepository.Add(group);
        scope.Complete();
        return CreatedAtAction(nameof(GetGroup), new { id = group.GroupId }, group);
    }
}

// DELETE: api/Groups/5
[HttpDelete("{id}")]
0 references
public IActionResult DeleteGroup(int id)
{
    _groupRepository.Delete(id);
    return new OkResult();
}
```

FIG. 33



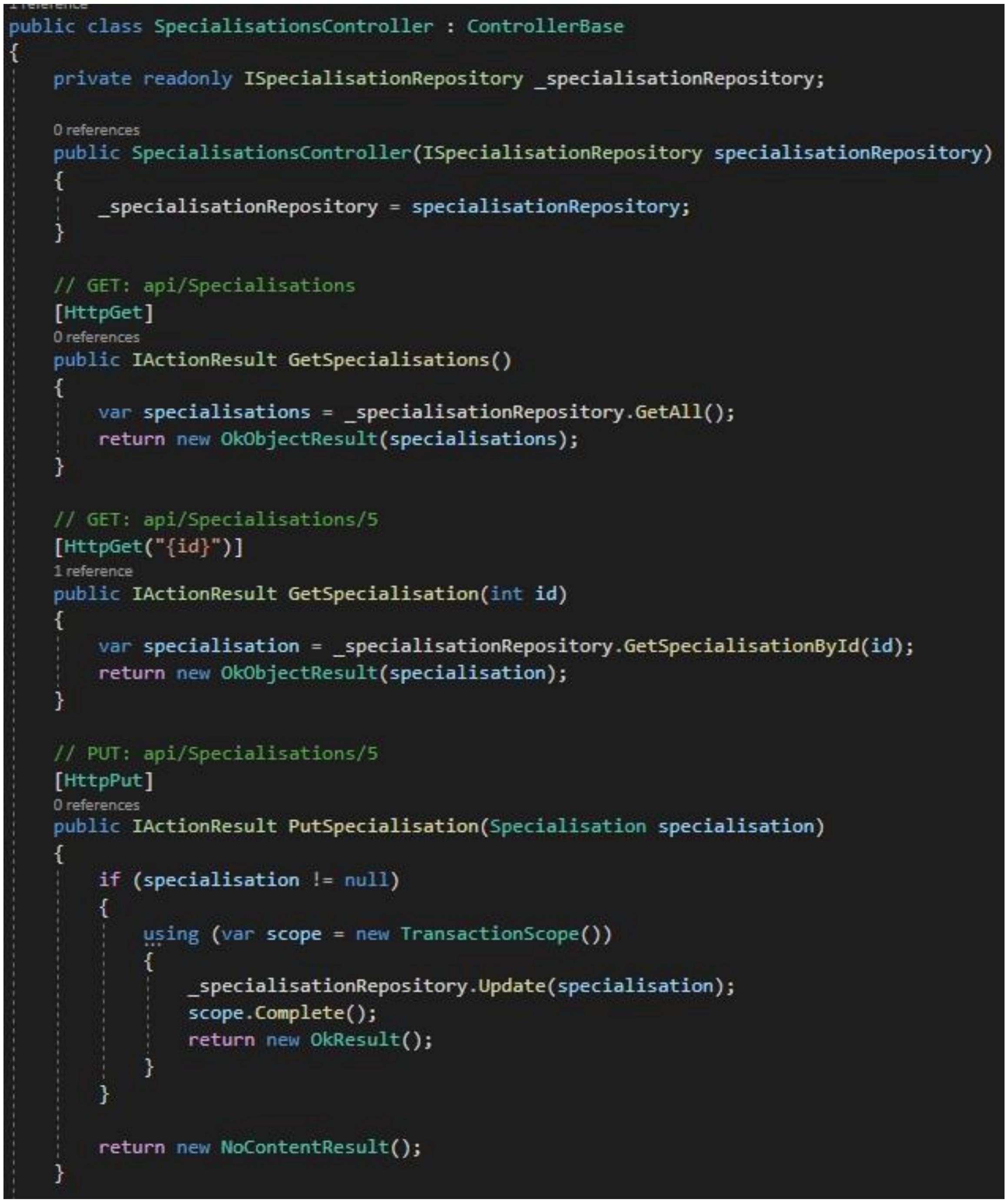
```
public class SubgroupsController : ControllerBase
{
    private readonly ISubgroupRepository _subgroupRepository;

    0 references
    public SubgroupsController(ISubgroupRepository subgroupRepository)
    {
        _subgroupRepository = subgroupRepository;
    }

    // GET: api/Subgroups
    [HttpGet]
    0 references
    public IActionResult GetSubgroups()
    {
        var subgroups = _subgroupRepository.GetAll();
        return new OkObjectResult(subgroups);
    }

    // GET: api/Subgroups/5
    [HttpGet("{id}")]
    1 reference
    public IActionResult GetSubgroup(int id)
    {
        var subgroups = _subgroupRepository.GetSubgroupById(id);
        return new OkObjectResult(subgroups);
    }

    // PUT: api/Subgroups/5
    [HttpPut]
    0 references
    public IActionResult PutSubgroup( Subgroup subgroup)
    {
        if (subgroup != null)
        {
            using (var scope = new TransactionScope())
            {
                _subgroupRepository.Update(subgroup);
                scope.Complete();
                return new OkResult();
            }
        }

        return new NoContentResult();
    }
```

FIG. 34



```
    // POST: api/Subgroups
    [HttpPost]
    0 references
    public IActionResult PostSubgroup(Subgroup subgroup)
    {
        using (var scope = new TransactionScope())
        {
            _subgroupRepository.Add(subgroup);
            scope.Complete();
            return CreatedAtAction(nameof(GetSubgroup), new { id = subgroup.SubgroupId }, subgroup);
        }
    }

    // DELETE: api/Subgroups/5
    [HttpDelete("{id}")]
    0 references
    public IActionResult DeleteSubgroup(int id)
    {
        _subgroupRepository.Delete(id);
        return new OkResult();
    }
}
```

FIG. 35



```
public class SpecialisationsController : ControllerBase
{
    private readonly ISpecialisationRepository _specialisationRepository;

    0 references
    public SpecialisationsController(ISpecialisationRepository specialisationRepository)
    {
        _specialisationRepository = specialisationRepository;
    }

    // GET: api/Specialisations
    [HttpGet]
    0 references
    public IActionResult GetSpecialisations()
    {
        var specialisations = _specialisationRepository.GetAll();
        return new OkObjectResult(specialisations);
    }

    // GET: api/Specialisations/5
    [HttpGet("{id}")]
    1 reference
    public IActionResult GetSpecialisation(int id)
    {
        var specialisation = _specialisationRepository.GetSpecialisationById(id);
        return new OkObjectResult(specialisation);
    }

    // PUT: api/Specialisations/5
    [HttpPut]
    0 references
    public IActionResult PutSpecialisation(Specialisation specialisation)
    {
        if (specialisation != null)
        {
            using (var scope = new TransactionScope())
            {
                _specialisationRepository.Update(specialisation);
                scope.Complete();
                return new OkResult();
            }
        }

        return new NoContentResult();
    }
```

FIG. 36

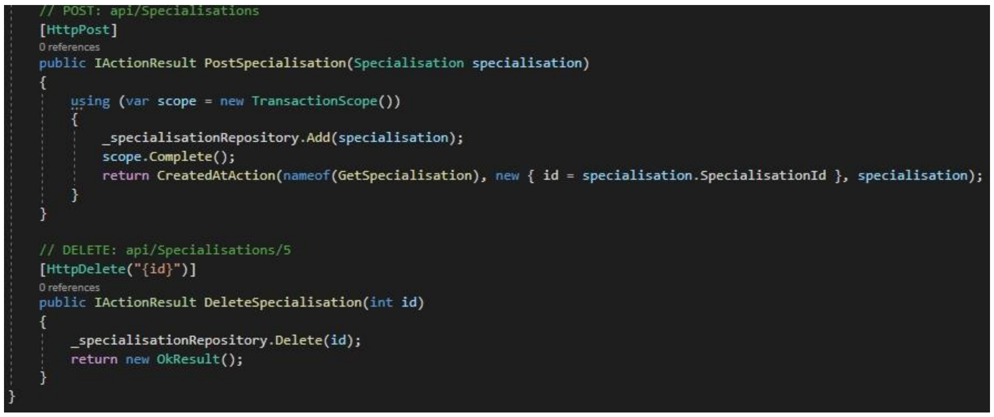

```
    // POST: api/Specialisations
    [HttpPost]
    0 references
    public IActionResult PostSpecialisation(Specialisation specialisation)
    {
        using (var scope = new TransactionScope())
        {
            _specialisationRepository.Add(specialisation);
            scope.Complete();
            return CreatedAtAction(nameof(GetSpecialisation), new { id = specialisation.SpecialisationId }, specialisation);
        }
    }

    // DELETE: api/Specialisations/5
    [HttpDelete("{id}")]
    0 references
    public IActionResult DeleteSpecialisation(int id)
    {
        _specialisationRepository.Delete(id);
        return new OkResult();
    }
}
```

FIG. 37

The *Rooms Microservice* is the microservice that manages the rooms that are within the institute, as its role is to ease process of visualizing and choosing the proper rooms for each activity. This microservice is set up to run on the URL *https://localhost:5447*, and the connection to the database is established with the help of the default *Connection String,* as you can see from the two pictures below.

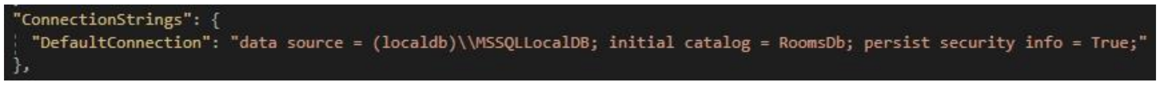

```
"ConnectionStrings": {
  "DefaultConnection": "data source = (localdb)\\MSSQLLocalDB; initial catalog = RoomsDb; persist security info = True;"
},
```

FIG. 38

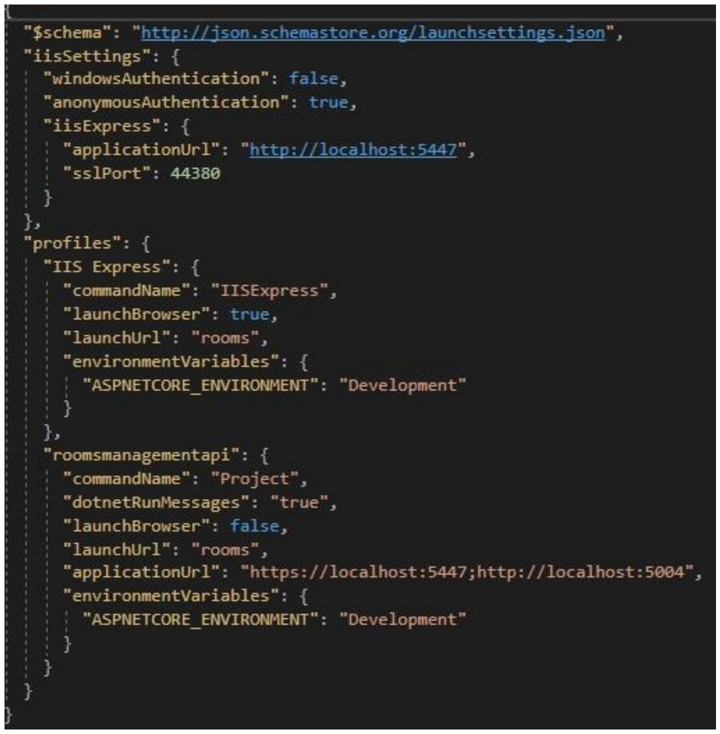
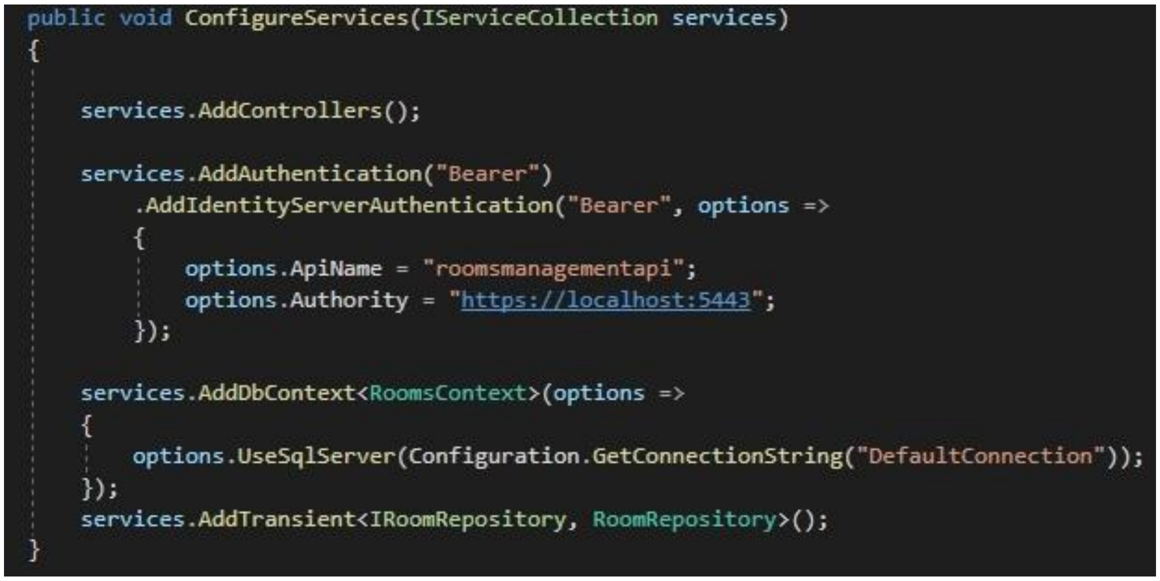

```
{
  "$schema": "http://json.schemastore.org/launchsettings.json",
  "iisSettings": {
    "windowsAuthentication": false,
    "anonymousAuthentication": true,
    "iisExpress": {
      "applicationUrl": "http://localhost:5447",
      "sslPort": 44380
    }
  },
  "profiles": {
    "IIS Express": {
      "commandName": "IISExpress",
      "launchBrowser": true,
      "launchUrl": "rooms",
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development"
      }
    },
    "roomsmanagementapi": {
      "commandName": "Project",
      "dotnetRunMessages": "true",
      "launchBrowser": false,
      "launchUrl": "rooms",
      "applicationUrl": "https://localhost:5447;http://localhost:5004",
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development"
      }
    }
  }
}
```

FIG. 39

In fig. 40 we have the configuration of the services that this microservice uses. As this is a basic setup for every microservice, for authentication the Bearer token is used together with the addition of the API Name, which is the name of the current microservice, and with the URL of the authority called *Authentication Microservice*.

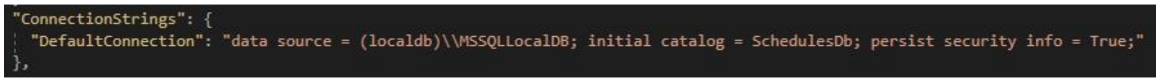


```
public void ConfigureServices(IServiceCollection services)
{

    services.AddControllers();

    services.AddAuthentication("Bearer")
        .AddIdentityServerAuthentication("Bearer", options =>
        {
            options.ApiName = "roomsmanagementapi";
            options.Authority = "https://localhost:5443";
        });

    services.AddDbContext<RoomsContext>(options =>
    {
        options.UseSqlServer(Configuration.GetConnectionString("DefaultConnection"));
    });
    services.AddTransient<IRoomRepository, RoomRepository>();
}
```

FIG. 40

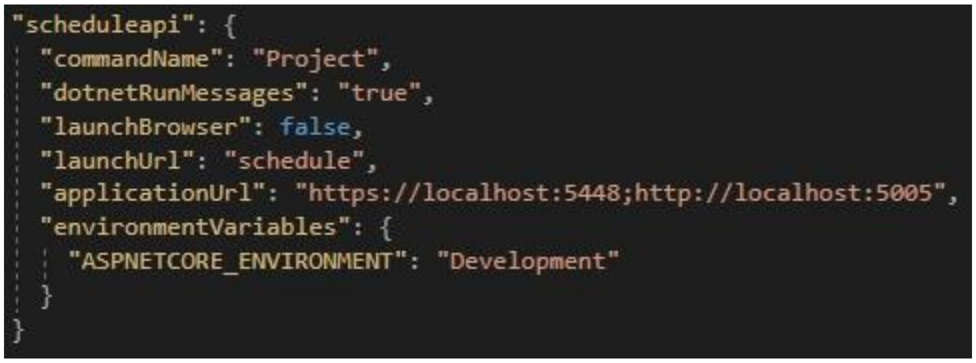
In the *RoomRepository* we have implemented the functions necessary to use in this microservice's controller. The *RoomsController* consists of CRUD operations and it is able to make modifications to the existent entities that are within the *Room* database, or to add new entries to the database.

The *Schedule Microservice* is the microservice that manages the schedule for each activity. In the following two screenshots you can see that this microservice is set up to run on the URL *https://localhost:5448*, and that the connection to the database is established with the help of the default *Connection String.*



```
"ConnectionStrings": {
  "DefaultConnection": "data source = (localdb)\\MSSQLLocalDB; initial catalog = SchedulesDb; persist security info = True;"
},
```

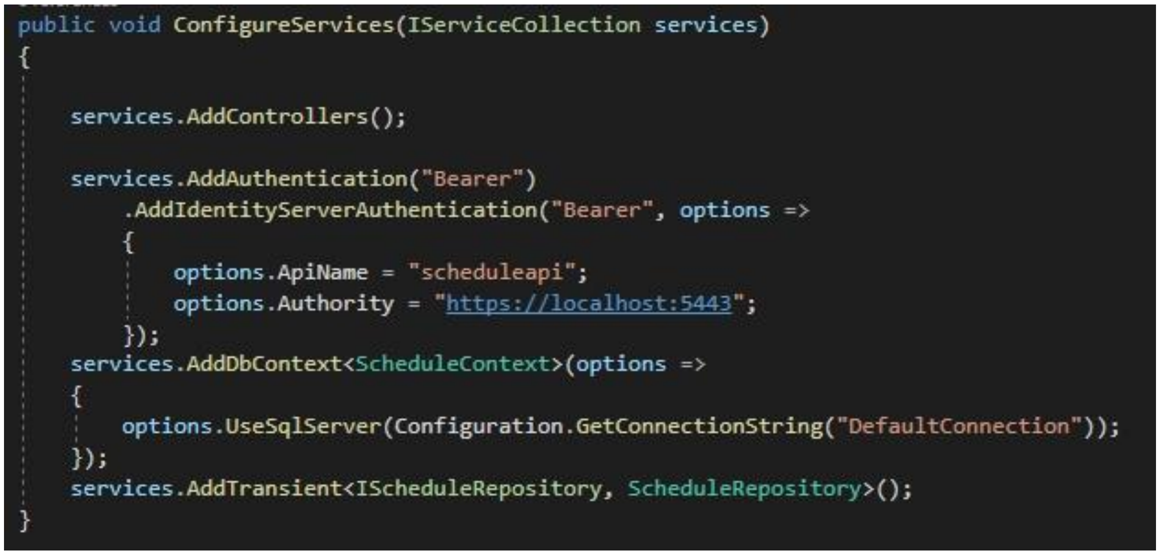
FIG. 41



```
"scheduleapi": {
  "commandName": "Project",
  "dotnetRunMessages": "true",
  "launchBrowser": false,
  "launchUrl": "schedule",
  "applicationUrl": "https://localhost:5448;http://localhost:5005",
  "environmentVariables": {
    "ASPNETCORE_ENVIRONMENT": "Development"
  }
}
```

FIG. 42

In fig. 43 we have the configuration of the services that this microservice uses. As this is a basic setup for every microservice, for authentication the Bearer token is used together with the addition of this API's Name and with the URL of the authority called *Authentication Microservice*.

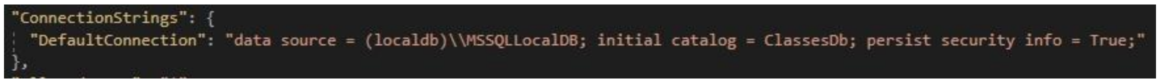


```
public void ConfigureServices(IServiceCollection services)
{

    services.AddControllers();

    services.AddAuthentication("Bearer")
        .AddIdentityServerAuthentication("Bearer", options =>
        {
            options.ApiName = "scheduleapi";
            options.Authority = "https://localhost:5443";
        });
    services.AddDbContext<ScheduleContext>(options =>
    {
        options.UseSqlServer(Configuration.GetConnectionString("DefaultConnection"));
    });
    services.AddTransient<IScheduleRepository, ScheduleRepository>();
}
```

FIG. 43

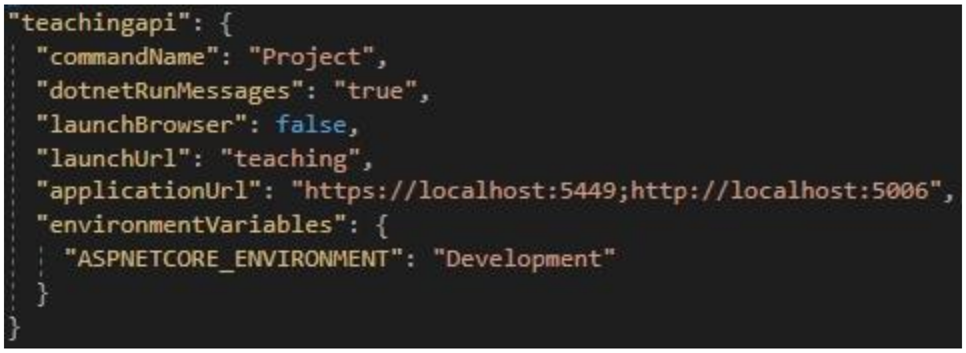
In the repository corresponding to this microservice we have implemented the functions necessary to use in its controller. The *SchedulesController* consists of CRUD operations and it is able to make modifications to the existent entities that are within the *Schedule* database, or to add new entries to the database.

The *Schedule Microservice* is the main microservice that will be used by teachers and students to teach and learn. In the following two screenshots you can see that this microservice is set up to run on the URL https://localhost:5449, and that the connection to the database is established with the help of the default *Connection String.*



```
"ConnectionStrings": {
  "DefaultConnection": "data source = (localdb)\\MSSQLLocalDB; initial catalog = ClassesDb; persist security info = True;"
},
```

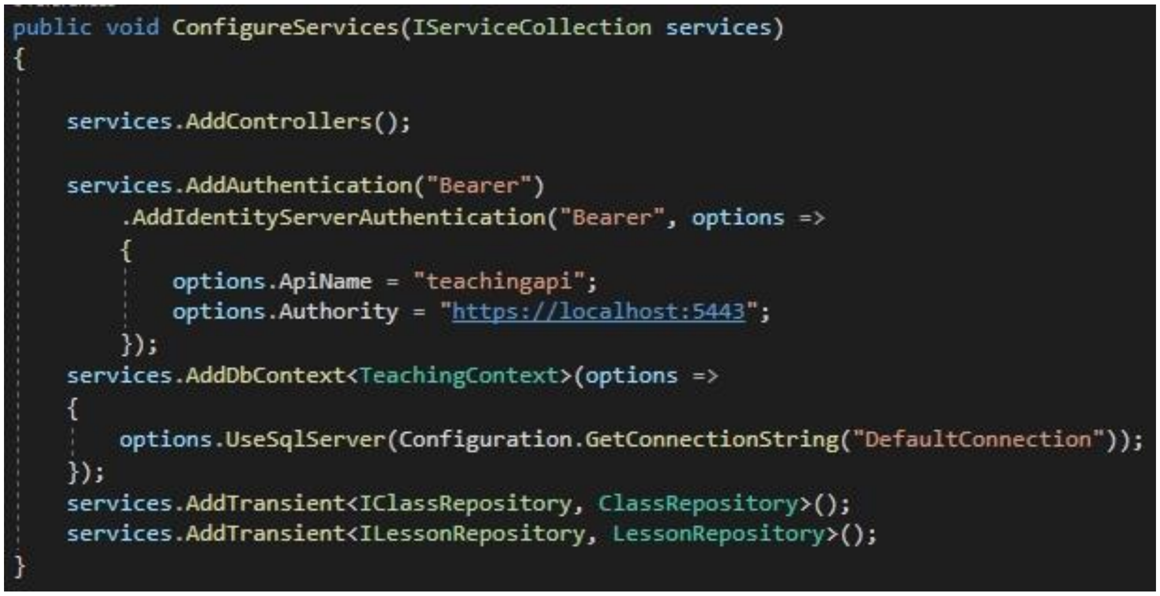
FIG. 44



```
"teachingapi": {
  "commandName": "Project",
  "dotnetRunMessages": "true",
  "launchBrowser": false,
  "launchUrl": "teaching",
  "applicationUrl": "https://localhost:5449;http://localhost:5006",
  "environmentVariables": {
    "ASPNETCORE_ENVIRONMENT": "Development"
  }
}
```

FIG. 45

In fig. 46 we have the configuration of the services needed for this microservice. As this is a basic setup for every microservice, for authentication the Bearer token is used together with the addition of this API’s Name and with the URL of the authority that we are using, the *Authentication Microservice*.



```
public void ConfigureServices(IServiceCollection services)
{

    services.AddControllers();

    services.AddAuthentication("Bearer")
        .AddIdentityServerAuthentication("Bearer", options =>
        {
            options.ApiName = "teachingapi";
            options.Authority = "https://localhost:5443";
        });
    services.AddDbContext<TeachingContext>(options =>
    {
        options.UseSqlServer(Configuration.GetConnectionString("DefaultConnection"));
    });
    services.AddTransient<IClassRepository, ClassRepository>();
    services.AddTransient<ILessonRepository, LessonRepository>();
}
```

FIG. 46

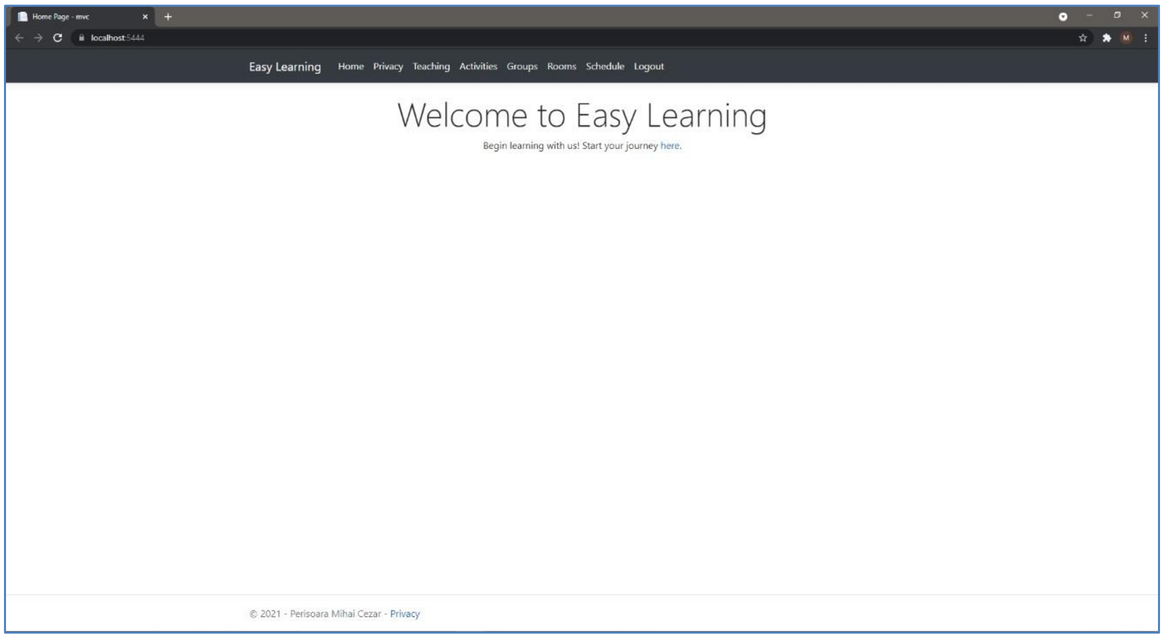
The actions of this microservice are split in half. One half takes care of the classes that are created, and the other half takes care of the content that is inside those classes. We have two separated repositories, one for each part. Those repositories contain the functions necessary to use in the two controllers that we have.

The *ClassesController and ClassLessonsController* contain CRUD operations and they are able to make modifications to the existent entities that are within the *Classes* database, or to add new entries to the database.

As we mentioned in the previous chapters, there are different user roles that have different limitations in terms of the actions that our users can do.

The two users that have access to all of the microservices implemented are the administrator and the super user. A user that has the role of a teacher has access to the *Authentication Microservice* and complete access to the *Teaching Microservice.* A user with the role of a student has the same access as a teacher when it comes to the *Authentication Microservice,* but only has limited access to the *Teaching Microservice,* as a student cannot modify in any shape or form the content that's within the class.

UI IMPLEMENTATION



Home Page - mvc

localhost:5444

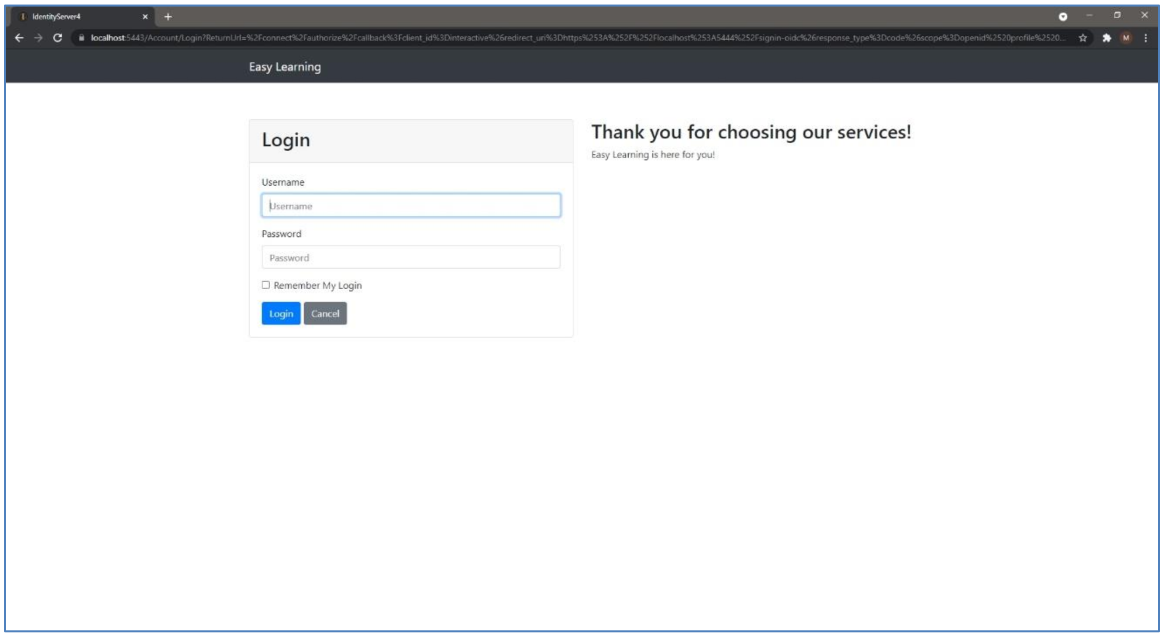
Easy Learning Home Privacy Teaching Activities Groups Rooms Schedule Logout

Welcome to Easy Learning

Begin learning with us! Start your journey here.

© 2021 - Perisoara Mihai Cezar - Privacy

FIG. 47



IdentityServer4

localhost:5443/Account/Login?ReturnUrl=%2Fconnect%2Fauthorize%2Fcallback%3Fclient_id%3Dinteractive%26redirect_uri%3Dhttps%253A%252F%252Flocalhost%253A5444%252Fsignin-oidc%26response_type%3Dcode%26scope%3Dopenid%2520profile%2520...

Easy Learning

Login

Username

Username

Password

Password

☐ Remember My Login

Login Cancel

Thank you for choosing our services!

Easy Learning is here for you!

FIG. 48

TECHNOLOGIES USED

To achieve the final result of this project, some technologies were used. All of the microservices mentioned up until this point and the main client application are running based on the ASP.NET framework. Besides the base of the ASP.NET framework, for each microservice there are multiple technologies that come in so called packages and are used in order to achieve the final result.

The *Authentication Microservice* uses the technologies offered by *Identity Server 4* together with *.NET Core Identity* and *Entity Framework,* while the database runs on the *SQL Server* database provider.

The other five microservices, *Activities Microservice, Groups Microservice, Rooms Microservice, Schedule Microservice and Teaching Microservice* are also build around quite a few technologies, such as *Entity Framework Core, SQL Server* database provider and the authentication handler from the *Identity Server 4 Access Token Validation.*

The main web application runs with the technology known as *ASP.NET Core MVC*, which stands for *Model-View-Controller* together with the necessary packages for running the *Authentication Microservice*: *ASP.NET Core OpenIdConnect* and *ASP.NET Core Identity Entity Framework Core*.

CONCLUSIONS

In conclusion, *Easy Learning* is an *Open-Source Learning Management System* based on *Microservices*. It is meant to ease the process of learning for students and to help teachers to provide their students with the proper content that will help the students during their learning.

BIBLIOGRAPHY

1. Identity Server 4
2. eLearning Industry - Best Learning Management Systems
3. Wikipedia – Learning Management System
4. Cloud Academy - Advantages and DIsadvantages of Microservices Architecture
5. ispring - Learning Management System Requirements
6. Skill Builder LMS - Open Source vs Cloud Based LMS
7. Medium - The What, Why and How of a Microservice Architecture
8. solace – Microservices Advantages and Disadvantages