|  |  |  |
| --- | --- | --- |
|  | UNIVERSITATEA DIN CRAIOVA  FACULTATEA DE AUTOMATICĂ, CALCULATOARE ȘI ELECTRONICĂ  DEPARTAMENTUL DE CALCULATOARE ȘI TEHNOLOGIA INFORMAȚIEI |  |

PROIECT DE DIPLOMĂ

Roman Valentin-Remus

COORDONATOR ȘTIINȚIFIC

**Conf. Dr. Ing. Marian Cristian Mihăescu**

Iulie, 2022

CRAIOVA

|  |  |  |
| --- | --- | --- |
|  | UNIVERSITATEA DIN CRAIOVA  FACULTATEA DE AUTOMATICĂ, CALCULATOARE ȘI ELECTRONICĂ  DEPARTAMENTUL DE CALCULATOARE ȘI TEHNOLOGIA INFORMAȚIEI |  |

**E-Learning platform based on microservices architecture**

Roman Valentin-Remus

COORDONATOR ȘTIINȚIFIC

**Conf. Dr. Ing. Marian Cristian Mihăescu**

Iulie, 2022

CRAIOVA

*„There are only two kinds of languages: the ones people complain about and the ones nobody uses.”*

Bjarne Stroustrup, The C++ Programming Language

**DECLARAȚIE DE ORIGINALITATE**

Subsemnatul Roman Valentin-Remus, student la specializarea Calculatoare si Tehnologia Informatiei (in limba engleza) din cadrul Facultății de Automatică, Calculatoare și Electronică a Universității din Craiova, certific prin prezenta că am luat la cunoştinţă de cele prezentate mai jos şi că îmi asum, în acest context, originalitatea proiectului meu de licenţă:

* cu titlul E-Learning platform based on microservices architecture,
* coordonată de Conf. Dr. Ing. Marian Cristian Mihăescu,
* prezentată în sesiunea Iulie, 2022.

La elaborarea proiectului de licenţă, se consideră plagiat una dintre următoarele acţiuni:

* reproducerea exactă a cuvintelor unui alt autor, dintr-o altă lucrare, în limba română sau prin traducere dintr-o altă limbă, dacă se omit ghilimele şi referinţa precisă,
* redarea cu alte cuvinte, reformularea prin cuvinte proprii sau rezumarea ideilor din alte lucrări, dacă nu se indică sursa bibliografică,
* prezentarea unor date experimentale obţinute sau a unor aplicaţii realizate de alţi autori fără menţionarea corectă a acestor surse,
* însuşirea totală sau parţială a unei lucrări în care regulile de mai sus sunt respectate, dar care are alt autor.

Pentru evitarea acestor situaţii neplăcute se recomandă:

* plasarea între ghilimele a citatelor directe şi indicarea referinţei într-o listă corespunzătoare la sfărşitul lucrării,
* indicarea în text a reformulării unei idei, opinii sau teorii şi corespunzător în lista de referinţe a sursei originale de la care s-a făcut preluarea,
* precizarea sursei de la care s-au preluat date experimentale, descrieri tehnice, figuri, imagini, statistici, tabele et caetera,
* precizarea referinţelor poate fi omisă dacă se folosesc informaţii sau teorii arhicunoscute, a căror paternitate este unanim cunoscută și acceptată.

Data, Semnătura candidatului,

|  |  |  |
| --- | --- | --- |
|  | UNIVERSITATEA DIN CRAIOVA  Facultatea de Automatică, Calculatoare şi Electronică  Departamentul de Calculatoare și Tehnologia Informației | Aprobat la data de  …………………  Şef de departament,  Prof. dr. ing.  Marius BREZOVAN |

**PROIECTUL DE DIPLOMĂ**

|  |  |
| --- | --- |
| Numele și prenumele studentului/-ei: |  |
| Enunțul temei: | E-Learning platform based on microservices architecture |
| Datele de pornire: | Datele de pornire au fost generate prin intermediul aplicației |
| Conținutul proiectului: | |  | | --- | | * Capitolul I acoperă introducerea * Capitolul II acoperă limbajele de programare tehnologiile și uneltele folosite * Capitolul III acoperă prezentarea arhitecturii sistemului, acestuia și detalii de implementare. * Capitolul IV acoperă funcționalitățile sistemului | |
| Material grafic obligatoriu: |  |
| Consultații: | Săptămânale |
| Conducătorul științific  (titlul, nume și prenume, semnătura): | Conf. Dr. Ing. Marian Cristian Mihăescu |
| Data eliberării temei: | 25.11.2021 |
| Termenul estimat de predare a proiectului: | 04.07.2022 |
| Data predării proiectului de către student și semnătura acestuia: |  |

|  |  |  |
| --- | --- | --- |
|  | UNIVERSITATEA DIN CRAIOVA  Facultatea de Automatică, Calculatoare şi Electronică  Departamentul de Calculatoare și Tehnologia Informației |  |

**REFERATUL CONDUCĂTORULUI ȘTIINȚIFIC**

|  |  |
| --- | --- |
| Numele și prenumele candidatului/-ei: | Roman Valentin-Remus |
| Specializarea: | Calculatoare și Tehnologia Informației (in limba engleza) |
| Titlul proiectului: | E-Learning platform based on microservices architecture |
| Locația în care s-a realizat practica de documentare (se bifează una sau mai multe din opțiunile din dreapta): | În facultate □ |
| În producție □ |
| În cercetare □ |
| Altă locație: [*se detaliază*] |

În urma analizei lucrării candidatului au fost constatate următoarele:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Nivelul documentării | | Insuficient  □ | Satisfăcător □ | Bine  □ | Foarte bine  □ |
| Tipul proiectului | | Cercetare  □ | Proiectare  □ | Realizare practică □ | Altul  [*se detaliază*] |
| Aparatul matematic utilizat | | Simplu  □ | Mediu  □ | Complex □ | Absent  □ |
| Utilitate | | Contract de cercetare □ | Cercetare internă □ | Utilare  □ | Altul  [*se detaliază*] |
| Redactarea lucrării | | Insuficient  □ | Satisfăcător □ | Bine  □ | Foarte bine  □ |
| Partea grafică, desene | | Insuficientă  □ | Satisfăcătoare □ | Bună  □ | Foarte bună  □ |
| Realizarea practică | Contribuția autorului | Insuficientă  □ | Satisfăcătoare □ | Mare  □ | Foarte mare  □ |
| Complexitatea  temei | Simplă  □ | Medie  □ | Mare  □ | Complexă  □ |
| Analiza cerințelor | Insuficient  □ | Satisfăcător □ | Bine  □ | Foarte bine  □ |
| Arhitectura | Simplă  □ | Medie  □ | Mare  □ | Complexă  □ |
| Întocmirea specificațiilor funcționale | Insuficientă  □ | Satisfăcătoare □ | Bună  □ | Foarte bună  □ |
| Implementarea | Insuficientă  □ | Satisfăcătoare □ | Bună  □ | Foarte bună  □ |
| Testarea | Insuficientă  □ | Satisfăcătoare □ | Bună  □ | Foarte bună  □ |
| Funcționarea | Da  □ | Parțială  □ | Nu  □ | |
| Rezultate experimentale | | Experiment propriu  □ | | Preluare din bibliografie  □ | |
| Bibliografie | | Cărți | Reviste | Articole | Referințe web |
| Comentarii  și  observații | |  | | | |

În concluzie, se propune:

|  |  |
| --- | --- |
| ADMITEREA PROIECTULUI  □ | RESPINGEREA PROIECTULUI  □ |

Data, Semnătura conducătorului științific,

**PROJECT SUMMARY**

This project’s purpose is to create an E-Learning web application based on microservices architecture, with a domain-driven design for each of the modules. The project consists of 8 modules that are hosted using docker containers, with a docker-compose to hold them together. The modules are the following: the main RecSys REST API that is the entrypoint for the whole system, the orders API that processes incoming orders and creates licenses, the Angular client application, the SQL Express Database Server that contains all the metadata needed for the system to work properly, the MongoDB Database that stores the actual videos and photos and the Grafana, Prometheus and cAdvisor tools that are used to monitor the health and the resource usage of the application. The REST API and the Orders API contain all the useful components like data entities, data transfer objects, repositories, services, controllers, unit of work, validators, etc.

The RecSys REST application is constructed using C# with .NET 5.0 framework, uses EF Core as the object-relational mapper, and follows the domain-driven design architecture, meaning that the whole application is split into multiple layers: the Presentation layer that sits on the front of the application and is the outermost layer, being the point of access, the Infrastructure layer that contains most of the concrete implementation of the interfaces and is tied directly to external dependencies, the Application layer that contains all the business logic and orchestrates all the services into producing useful results for the controllers and the Domain layer, that is the core of the application and that contains the entities and the relation between the code structure and the database. Orders API also follows the same pattern as RecSys, both of the programs sharing the Domain layer that sits at their core but having different Application and Infrastructure layers. The search engine of the project uses the Full-Text Search capability of the SQL Server, with access to semantic searches across the data.

To build up and host the entire system I have used Docker with a Docker Compose project, which is used to create the network between the containers and to attach each container to its own volumes. For the storage part, I have used a code-first approach using EF Core and SQL Express Server and a non-relational MongoDB database, each tied to their own volume in order to preserve the data in case the container or image gets rebuilt. The application is authorized using JWT tokens with policy-based auth, it has support for rotated refresh tokens and also supports token invalidation, to prevent security breaches from leaked tokens. As entrypoints, the Angular Client application is the only place a user has access to, but the RecSys REST API is public so that it can be accessed from everywhere with HTTPS requests.

***Keywords***: Microservices, E-Learning, API, Docker, .NET, Core, SQL, MongoDB, EF, JWT, Angular, database, cAdvisor, Grafana, Prometheus, Application, Domaine, Infrastructure, REST

**ACKNOWLEDGEMENTS**

I would like to thank my coordinator, conf. dr. ing. Cristian Mihaescu, for his guidance and dedication during my college years. He always was open to helping me with advice whenever I requested his help.

**SUMMARY**

[1. Introduction 13](#_Toc106891297)

[1.1 Purpose 13](#_Toc106891298)

[1.2 Motivation 13](#_Toc106891299)

[2. Tools and Technologies 14](#_Toc106891300)

[2.1 Programming Languages and Frameworks 14](#_Toc106891301)

[2.1.1 RecSys REST API 14](#_Toc106891302)

[2.1.2 RecSys Angular Client 15](#_Toc106891303)

[2.1.3 Entity Framework Core 15](#_Toc106891304)

[2.1.4 OrdersAPI Microservice 16](#_Toc106891305)

[2.1.5 Docker 17](#_Toc106891306)

[2.1.6 SQL Server Express 18](#_Toc106891307)

[2.1.7 MongoDB 20](#_Toc106891308)

[2.1.8 Visual Studio 2022 and Visual Studio Code 21](#_Toc106891309)

[2.1.9 Version control 21](#_Toc106891310)

[2.1.10 Grafana, Prometheus and cAdvisor 22](#_Toc106891311)

[2.2 Similar existing systems 23](#_Toc106891312)

[2.2.1 Udemy 23](#_Toc106891313)

[2.2.2 PluralSight 24](#_Toc106891314)

[3. System design and implementation 25](#_Toc106891315)

[3.1 Modules interaction 26](#_Toc106891316)

[3.2 Application Architecture 29](#_Toc106891317)

[3.2.1 RecSys REST API Architecture 31](#_Toc106891318)

[3.2.2 RecSys Angular Client Architecture 32](#_Toc106891319)

[3.2.3 SQL database 34](#_Toc106891320)

[3.2.4 MongoDB database 35](#_Toc106891321)

[3.3 Application Implementation details 36](#_Toc106891322)

[3.3.1 RecSys API 36](#_Toc106891323)

[3.3.2 OrdersAPI 40](#_Toc106891324)

[3.3.3 RecSys Client 40](#_Toc106891325)

[3.4 Hosting and containers orchestration 41](#_Toc106891326)

[3.4.1 Docker 41](#_Toc106891327)

[3.4.2 Docker compose 44](#_Toc106891328)

[4. System Functionalities 45](#_Toc106891329)

[4.1.1 Use cases 45](#_Toc106891330)

[4.1.2 System functionalities 47](#_Toc106891331)

[A. Bibliography 53](#_Toc106891332)

[B. Web references 54](#_Toc106891333)

[C. Source code 55](#_Toc106891334)

**LIST OF FIGURES**

[Figure 1- Docker engine components 17](#_Toc106881021)

[Figure 2 - MongoDB RecSys Database seen through MongoDB Compass 20](#_Toc106881022)

[Figure 3 - System overview schematic 26](#_Toc106881023)

[Figure 4 - RecSysAPI CPU usage seen in cAdvisor monitor 28](#_Toc106881024)

[Figure 5 - RecSysApi projects overview 31](file:///C:\Users\Remus\Documents\E-LearningRecSys\Roman-Valentin-Remus-Diploma-Project.docx#_Toc106881025)

[Figure 6 - Dependencies between layers in DDD 32](#_Toc106881026)

[Figure 7 - Client application structure 33](file:///C:\Users\Remus\Documents\E-LearningRecSys\Roman-Valentin-Remus-Diploma-Project.docx#_Toc106881027)

[Figure 8 - Database Overview 34](#_Toc106881028)

[Figure 9 - Video Files Objects 35](#_Toc106881029)

[Figure 10 - Video Chunk Objects 35](#_Toc106881030)

[Figure 11 - Dependency injection in SearchController 36](#_Toc106881031)

[Figure 12 - Search query action inside SearchController 37](#_Toc106881032)

[Figure 13 - Controller with Policy Authorize Attribute 37](#_Toc106881033)

[Figure 14 - Adding Auth settings to the API 38](#_Toc106881034)

[Figure 15 - Infrastructure Layer overview 38](file:///C:\Users\Remus\Documents\E-LearningRecSys\Roman-Valentin-Remus-Diploma-Project.docx#_Toc106881035)

[Figure 16 - Application Layer Overview 39](file:///C:\Users\Remus\Documents\E-LearningRecSys\Roman-Valentin-Remus-Diploma-Project.docx#_Toc106881036)

[Figure 17 - Domain Layer Overview 39](file:///C:\Users\Remus\Documents\E-LearningRecSys\Roman-Valentin-Remus-Diploma-Project.docx#_Toc106881037)

[Figure 18 - OrdersAPI Overview 40](file:///C:\Users\Remus\Documents\E-LearningRecSys\Roman-Valentin-Remus-Diploma-Project.docx#_Toc106881038)

[Figure 19 - RecSys API Dockerfile 42](#_Toc106881039)

[Figure 20 - Docker Compose file for the RecSys System 43](#_Toc106881040)

[Figure 21- Essential Use Cases 45](#_Toc106881041)

[Figure 22 - Home page 46](#_Toc106881042)

[Figure 23 - Search results page 46](#_Toc106881043)

[Figure 24 - Course overview 47](#_Toc106881044)

[Figure 25 - Cart page 47](#_Toc106881045)

[Figure 26 - Owned courses overview 48](#_Toc106881046)

[Figure 27 - Watch video page 48](#_Toc106881047)

[Figure 28 - Contact us page 49](#_Toc106881048)

[Figure 29 - Course details 49](#_Toc106881049)

[Figure 30 - Section details 50](#_Toc106881050)

[Figure 31 - Video details and content upload 50](#_Toc106881051)

# Introduction

## Purpose

The purpose of this project is to create an E-Learning web application, with a client module built with Angular framework and an API based on a domain driven design with microservices architecture. The main idea behind this project is to have a modern front end built with the latest technologies and with an attractive look and feel and a sturdy, easily extendable and reusable API system that processes all the complex logic hidden behind the system.

The application’s intent is to provide courses and videos that fits best the queries for user, at a good quality and with good loading speed. It searches all data for each course and video and provides the user with the best matches that fits the specified criteria. In addition to searching for data, a filtering panel is also available.

In the end, the main purpose of this application is to create an easily accessible and reliable virtual platform where content creators can post their work and users can easily access knowledge.

## Motivation

Online learning applications have provided access to knowledge for a wide range of people, continously evolving in order to provide the best so that people will not migrate to other platforms. The application is as important as the knowladge it holds, therefore allowing content creators to upload their work in accessible manner is a must for this kind of business.

A concrete problem with the educational resources available online is that they are sometimes erroneous, incomplete or non-existent. Another drawback is that there are not many online platforms that centralize educational resources, information or courses in multiple and diverse fields.

The motivation for choosing this topic came from the desire to solve this problem. I have chosen to develop an online platform where people specialized in various fields can create courses and educational resources to help other people to enrich their knowledge or to specialize in a certain field.

# Tools and Technologies

## Programming Languages and Frameworks

### RecSys REST API

The backend has been written in **C#** using **ASP.NET 5.0** framework due to the following reasons:

* It is an object-oriented, threaded and dynamic programming language, with which I am used to because I have worked with it for a long time period
* It has a wide range of libraries that can be used to accomplish every basic need like Jackson for json serialize/deserialize, EF Core for ORM and others
* It is a popular language and framework, so a lot of materials and support on forums are available with a quick internet search
* It has an automatic garbage collection system so memory management is no longer a problem for the programmer, therefore most of the time is spent improving the logic behind the application and not looking for memory leaks
* Async await support is very useful because it efficiently uses a single core and it does not spend time on expensive I/O operations
* Being an object oriented language, it helps alot with code organization and it is easier to follow the SOLID and DRY principles
* A lot of design patterns like singleton and dependency injection are already implemented by the framework, so the programmer can use them out of the box, without having to recreate the whole system.
* LINQ expressions are very useful for performing queries over internal models
* Being part of the C family of programming languages, it has been easy for me to transit from C++ to C#
* Cross-platform capabilities provided by the .NET platform – due to this, it can be run on Windows, Linux and MacOS without any modifications

A possible drawback of this language and framework is that it is the performance isn’t the absolute best, on benchmarks losing to C++ due to C++ being closer to the actual machine. Also memory management plays an important part in the performance comparison. Another drawback of this language is the hard learning curve, you have to learn alot about all different tools that are available inside the ecosystem.

### RecSys Angular Client

The client part has been completely separated from the API part, meaning that I have not used an MVC approach. This provides the benefit of decoupling the client application from the backend. In this case, the backend can be completely changed (even written in another programming language) without changing anything in the client app. I have chosen **Angular 13** as the framework for creating the client application due to the following reasons:

* Typescript – it is a superset of the javascript programming language with syntax for types, developed and maintained by Microsoft. It makes it easy to create large applications that are compiled into javascript
* Services and Dependency Injection pattern - a service or component may require other dependent services to complete a task. Dependency injection design patterns are used to satisfy these dependencies, dividing the task among different services that are reused across the application. The Angular Injector provides this feature out of the box.
* Enhanced Design Architecture – everything can be split up and arranged into modules, making it very easy to organize dependencies and to find easy bits and parts of the application that needs to be created or updated
* Directives – they are used to change the DOM according to the logic defined by the programmer, dynamically adding classes and styles and two way data-binding to HTML form elements
* State management – it is very important for the application to hold state because it allows us to manage the auth information, also it can be very useful when used in cart management and a lot of other important parts that have data that changes according to the users actions.

The drawbacks of using Angular are steep learning curve, because the array of topics and aspects to be covered is quite large and it takes some practice before being able to actually produce something usefull using it.

### Entity Framework Core

Entity Framework (EF) Core is a lightweight, extensible, open source and cross-platform version of the popular Entity Framework data access technology.1

EF Core serves as an ORM that enables .NET developers to use a database through entities, classes thare are defined in code, and removes most of the need to write SQL code for database operations. It uses entity attributes for creating table columns and has navigation properties that automatically creates relationships between models.

One nice feature of EF Core is that, through migrations, it allows to update the database to a particular state, while making it easy to downgrade in case something did not went well. This kind of database state management is particulary usefull because you can recreate a database state in a certain period by just using one command. You still have to seed the data if the database starts from scratch, but this can be easily done using scripts.

Another important feature is that you can opt to use a code-first approach in managing the database but you can also use a database-first approach. I have used the code-first approach because I wanted the database to follow the needs provided by the logic inside the application. Other developers might consider that database-first approach is the better approach because they have to work with already created database to build new projects.

Integrating EF Core, I have used the Dependency Injection, the Unit Of Work and the Repository patterns, by passing the DbContext to the Unit Of Work through its constructor, and the UOW instantiated the repositories by injecting the DbContext towards the repositories. This way, every repository shares the same context, and every service share the Unit Of Work. The saveChanges action sends the state of the context towards the database, making the changes permanent. This has the benefit that if some operation fails before the saveChanges is called, the database is not altered, allowing the services logic that interacts with the database to be atomic.

The only drawback that I could found about Entity Framework is that data migration can break easily, meaning that one small mistake can consume a lot of time to figure how to fix it. Also, working on multiple machines (my laptop and my desktop computer) I have noticed that sometimes the database get out of sync due to recreating parts of the migrations.

### OrdersAPI Microservice

This API is a microservice that runs on the same technologies like **RecSys REST API**, meaning that it was written using **C#** on **ASP.NET 5.0** framework. I have chosen to use the same technologies in order to share the domain between the two apps, but also because I am more used to working with those technologies.

### Docker



Figure 1- Docker engine components

<https://docs.docker.com/engine/images/engine-components-flow.png> 2

**Docker** is an open source containerization platform. It enables developers to package applications into containers—standardized executable components combining application source code with the operating system (OS) libraries and dependencies required to run that code in any environment. 3

I have used Docker to recreate the environment needed in order for each application to run as it was on its supported OS with all the packages and dependencies installed, but without having to create a virtual machine for each one. This leads to better returon on investment and cost savings in a real life scenario.

Another benefit of Docker is rapid deployment - It can decrease deployment to seconds. It is because of the fact that it can create a container for every process and even does not boot an OS. So, even without worrying about the cost to bring it up again, it would be higher than what is affordable, Data can be created as well as destroyed.

Security is also important when it comes to this tool, Docker makes sure that applications that are running on containers are completely segregated and isolated from each other, from a security point of view, by granting us complete control over traffic flow and management.

Simplicity and faster configurations is a key selling point of Docker, because It gives flexibility to users to take their own configuration, put that into the code, and further deploy it without any problems. However, the requirements of the infrastructure are no longer linked with the environment of the application, as Docker can be used in a wide variety of environments.

With the help of a Docker, we can build a container image and can further use that same image over every step of the deployment process. The advantage of it is the ability to separate non-dependent steps and also run them in parallel. In addition, the duration of time it takes from build to production may speed up notably.

Another key point about Docker containers is that with containers, you can run several times as many copies of an application on the same hardware as you can using VMs. This can reduce your cloud spending. Also, Docker can track versions of a container image, roll back to previous versions, and trace who built a version and how. It can even upload only the deltas between an existing version and a new one.

Today Docker containerization also works with Microsoft Windows server. And most cloud providers offer specific services to help developers build, ship and run applications containerized with Docker.

### SQL Server Express

For the database, I have used SQL Server Express, a free version of Microsoft’s primary relational database management system (RDBMS) – the SQL Server. Essentially, the SQL Server is a database management system that can be used to store and access the information stored in many different databases. SQL Server comes with an impressive range of features like business intelligence, reporting, and in-depth advanced analytics. Also one of the main features I have used is the Full Text support with semantics analysis. I have chosen this because of the following reasons:

**Free:** One huge advantage of SQL Server Express is that it is free. Your only outlay is the time investment you make downloading and setting up the system. It is perfect for students that want to learn how to use SQL Server because there is nothing to lose by downloading the system and getting used to how it works.

**Features:** While Express is the lite version of SQL Server, there is still an impressive range of features that you would have to pay for with other systems. Express supports Full-Text Search, native XML, and the SQL Common Language Runtime. Other key features include a reporting component and report designer enabling custom report creation.

**Scalability:** SQL Server Express is an ideal starting point for smaller independent software vendors (ISVs) since it can be used with any smaller application. The licensing allows Express to be included as part of an app or product. While there are limitations on memory and socket usage, they are not as restrictive. Express is not limited to a single user which is a commonly held misconception. There is a 10GB database restriction, but that is a maximum size per database meaning you can have multiple databases that store up to 10GB of data.

**Security:** Within SQL Server Express there is the option of free online backup that will help to protect your valuable data if anything goes wrong.

There are some limitations within SQL Server Express, including:

* SQL Agent is not included in Express. The SQL Agent is a background tool which enables administrators to automate tasks like backing up data, database replication setup, job scheduling, user permissions, and database monitoring.
* The limit on the buffer cache for each instance is 1MB of RAM.
* 1GB maximum memory used by the SQL Server Database Engine
* The maximum size of each relational database is 10GB
* The relational database engine is restricted to the lesser of 1 socket or 4 cores.

### MongoDB

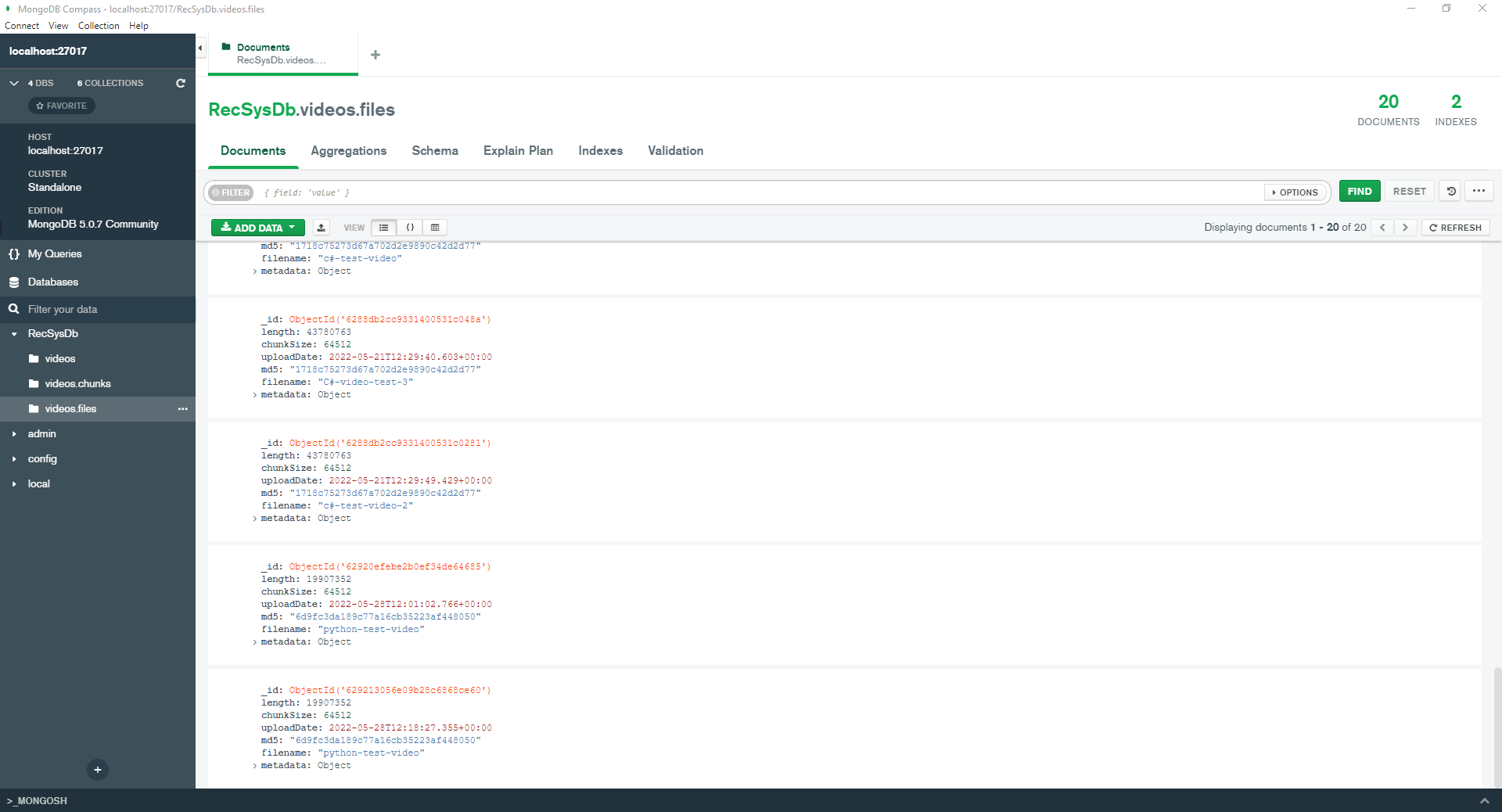


Figure 2 - MongoDB RecSys Database seen through MongoDB Compass

For storing videos I chose a non-relational database called **MongoDB** with a **GridFS** specification, that is used for storind and retrieving files that exceed the **BSON** document size limit of 16 MB.

Instead of storing data in arrays of rows or columns like SQL databases, each record in a MongoDB database is a document described in BSON, a binary representation of the data. Applications can then retrieve this information in JSON format. 4

MongoDB, being a document database, allows developers to easily store structured or unstructured data. It uses a JSON-like format to store documents. This format maps directly to native objects in most modern programming languages, making it a natural choice for developers since they don't need to think about normalizing the data. MongoDB can also handle large volumes and can scale up vertically or horizontally to accommodate large data loads on busy periods.

The reason that I have chosen **GridFS** is that instead of storing a file in a document, **GridFS** divides the file into parts or sections and stores each part in a separate document. By default, GridFS uses the default block size of 255 KB; i.e. GridFS splits a file into 255KB parts except for the last part. The last part is only as big as it needs to be. Likewise, files that are no larger than the block size have only one final block, using only the required space plus some additional metadata.5

As a tool for administrating the database, I have used **MongoDB Compass6** version **1.32.2** also seen in **figure 2**.

### Visual Studio 2022 and Visual Studio Code

For development, I have used **Visual Studio Comunity 2022** version **17.2.0** and **Visual Studio Code** because they fit best my development needs.

Microsoft Visual Studio is an integrated development environment (IDE) developed and maintained by Microsoft, used to develop web, desktop, mobile, web services, or games.

It includes a code editor that supports IntelliSense, as well as code refactoring. The built-in debugger works both as a source-level debugger and as a machine-level debugger. Microsoft Visual Studio also includes other components such as a code profile, a form designer, used to build graphical user interfaces (GUIs) for desktop or mobile applications, a class designer, and a database schema designer.

Visual Studio supports development in multiple programming languages, among which the most relevant for this work are: C#, C++, JavaScript, HTML, CSS.

In order to be able to use all the capabilities of the **ASP.NET CORE** framework, in the elaboration of this paper I used the version of **Visual Studio Comunity Edition 2022**, which is available free of charge.

For the Angular part I have used **Visual Studio Code** because it allows for easy development of the Angular application. It has support for code completion, IntelliSense and Prettier, which simplify the task of formatting and making sure that every code convention is respected.

### Version control

For versioning control, the tools used were **Git** and **Sourcetree**.

Git is a version control software for tracking changes in any set of files, usually used for coordinating work among programmers collaboratively developing source code during software development. Its goals include speed, data integrity, and support for distributed, non-linear workflows (thousands of parallel branches running on different systems).

The main advantages of **Git**, as presented on the official website7:

* It allows and encourages you to have multiple local branches that can be entirely independent of each other; the creation, merging, and deletion of those lines of development takes seconds
* Nearly all operations are performed locally, giving it a huge speed advantage on centralized systems that constantly have to communicate with a server somewhere
* It is distributed: this means that instead of doing a "checkout" of the current tip of the source code, you do a "clone" of the entire repository
* The data model that Git uses ensures the cryptographic integrity of every bit of the project; every file is check-summed and retrieved by its checksum when checked back out
* Unlike the other systems, Git has something called the "staging area" or "index"; this is an intermediate area where commits can be formatted and reviewed before completing this step

### Grafana, Prometheus and cAdvisor

For monitoring the state and the resources used by each container, I have used Grafana, Prometheus and cAdvisor because they are free, reliable, provide usefull information and were easily integrated within my docker-compose.

Designed to be an open-source monitoring and alerting system, **Prometheus** is a real-time, time-series database with a robust query language designed to provide aggregate insights from data series while they are collected.

The Prometheus server is where the data is stored, with many nodes to scale the influx of data received. Applications don’t send data to Prometheus; it is Prometheus that pulls data from the monitored systems. Of course, not all apps are well suited for a pull scheme, so it is possible to create a push gateway to serve as an intermediate service. The push service can also be used for a short-lived process like a serverless application, which is created and destroyed too fast to be discovered by the server without having its data pushed.

While Prometheus is all about how to store and query data, Grafana is about how to visualize this data so it is possible to identify issues quickly. Grafana describes itself as an “open-source metrics analytics & visualization suite.” It supports not only Prometheus but also Elasticsearch, InfluxDB, and many others.

**Grafana** works as a sandbox for graphic visualization, where you can create themed dashboards with several graphics by collecting data from different data sources and metrics. The definition of each dashboard varies from the data source, and, in the case of Prometheus.

Developed and maintained by Google, **cAdvisor** (container Advisor) is a running daemon that collects real-time monitoring data of containers. The project is an open-source monitoring tool that displays information in its own web interface. **cAdvisor** comes as a standalone deployment. It has native support for **Docker** containers and supports other container environments such as Kubernetes out of the box. This means that cAdvisor can collect valuable data analytics on any running container—including **CPU usage, memory, file system usage, physical and virtual network interfaces**, and more—while running on a single Docker image.

## Similar existing systems

### Udemy

**Udemy.com** is an online learning platform, founded in 2010, for professional adults who want to add new skills to their resume or for students who want to gain knowledge in a particular field. Unlike academia, Udemy offers a platform for experts in any field where they can create courses that can be offered to the general public, either for free or for a tuition fee. The platform offers users the opportunity to become instructors and create a course, promote it and earn money from tuition fees.

It has now reached 30 million students, 100,000 courses, 40,000 instructors and 22 million minutes of video content. All this makes the Udemy platform a world leader in the teaching and learning market, connecting students and instructors from all over the world.

Upon completion of a course, the student receives a graduation certificate, which is not currently recognized by any accredited educational institution.

Courses can be watched from any electronic device, from desktop, to tablets and mobile phones, in addition once purchased a course, the student has access to it forever and in most cases and the content added to it over time ( course updates).

The business model adopted by Udemy is a company-client, which sells the courses, created by the instructors, to the clients represented by the students, and the revenues resulting from the sale are divided between the instructor who created the course and the platform. Udemy prefers to keep the prices at an affordable level, $ 30- $ 50 so that they can be bought by as many people as possible.

Another service that Udemy offers is “Udemy for Business”, a separate platform where the employees of the partner companies can specialize in various fields, having access to all the top courses that exist on the platform and where the companies they work for can ask to be created special courses in a certain field for their employees.

### PluralSight

PluralSight is an online educational platform, founded in 2004, for IT professionals who want to gain new knowledge or for students who want to learn new concepts such as programming languages, data analysis, web application development or databases.

The platform offers the possibility for users to become authors on it and create new courses that users can use to gain new knowledge. Authors receive money for courses created based on the number of views of each course created.

PluralSight offers over 6,000 IT and technology courses, created by over 1,500 author trainers in this field. It also has partnerships with major companies such as Google, Microsoft, Unity, Oracle, Adobe and StackOverflow. All this makes the PluralSight platform one of the most important companies in the teaching and learning market in the field of IT and technology, managing to help students around the world.

The business model adopted by PluralSight is the monthly or annual subscription (subscription) model, which means that for a certain amount of money per month or per year, a user has access to all existing courses on the platform. In addition, there are premium tariff plans, which include certificates for completion of courses, examination tests or additional projects.

Another service that PluralSight offers is business access, which for a larger amount, allows the use of an account by several people, access to the platform API, data export and more.

Courses can be watched from any electronic device, from desktops to tablets and mobile phones as long as the user has a valid subscription.

# System design and implementation

Before the first single page applications frameworks appeared, the developed applications were multi-page applications built on the server side with server-side applications technologies such as ASP.NET, PHP or Java.

In production, many of the applications that are being developed are multi-page applications and a monolithic architecture, with quite a few cases where developing these applications is the best option. The advantages of multiple page applications are: the initial page load is usually faster than single page applications, search engine optimization (SEO) is easier to be made even more efficient because most crawlers were built for multi-page applications and most of the frameworks for developing multi-page applications (multiple page applications) are tested, secure and have a large developer community.

In recent years, frameworks for single page applications have grown significantly, with frameworks such as Angular, React and Vue being developed and popularized. Their main advantages are: the loading of the pages when the user browses through the application is faster, resulting in a more pleasant user experience and offers the possibility to build more complex systems that offer many functionalities, offering a more captivating user experience.

A single-page application **(SPA)** - based architecture typically includes a web API, which could also integrate with other web, desktop, or mobile applications. The Web API handles requests (HTTP requests or messages) by: running business logic, accessing the database, exchanging messages with other systems, and returning an HTML / JSON / XML response to client applications.

The advantage of this architecture is the presence of the principle of separation of concerns (separation of concerns), each of the applications having well-defined and separate roles.

The architecture of the system emphasizes the scalability, modularity, and separation of tasks of each module, also another important aspect in choosing this architecture was the ease of maintenance of the system and the possibility of updating it in the future.

## Modules interaction

The whole system is based on the modularity and intercommunication of the modules involved. Below, in figure 3, it is presented an overview of the whole system, and how each module interacts with the others:

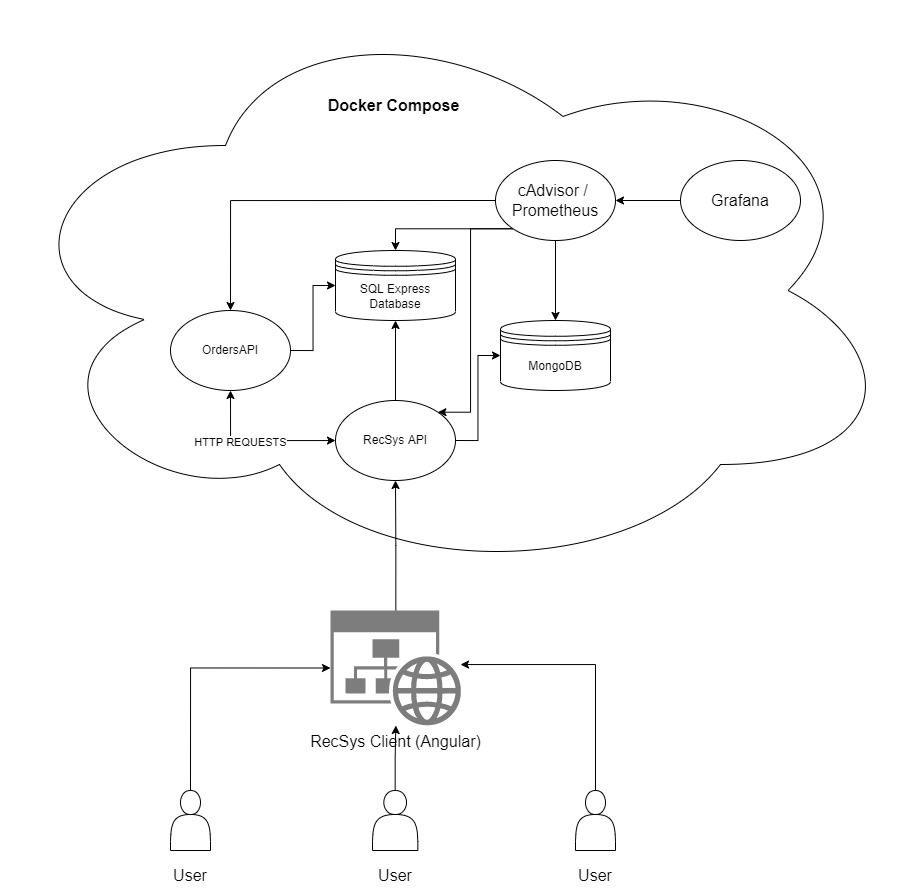


Figure 3 - System overview schematic

**RecSysAPI** is the entrypoint of the Docker cluster, all resources being exposed through it. It can be publicly accessed with https requests from anywhere, but most of the endpoints are protected with policy based restrictions. To access those resources, the request must contain an authorization header that embeds an JWT Bearer token that contains all the claims for that token, also signed with a secret key so that it can’t be tampered with without loosing the match with the generated signature. There are three types of users: Administrator, Publisher and User.

Each endpoint that is protected with the JWT token also checks for the required claims, meaning that a User can not access Administrator functions. Also, to provide additional entrypoint protection, the RecSys Client has guards that protect different routes from being access by users that don’t meet particular criterias.

The RecSys API communicates with the OrdersAPI when a checkout is created in the client application. The flow is that the user chooses some courses, presses the checkout button and that triggers a POST HTTPS request from the Client towards the RecSys API containing the state of the cart as the body and having an auth header gained when the user logged in. The RecSysAPI then forwards the request towards the OrdersAPI, that processes the request and creates licenses for each of the courses. Those licenses determine if the user has access or not to a particular course.

Another possible flow is for publishers or admins to create a course. This flow is triggered also from the client application, that creates an object with all the course metadata, including sections, videos, and video content for each, and sends multiple requests. One request contains all the metadata needed to provide quick access to courses and videos and others send the byte array contents for each video, which gets streamed towards the MongoDB database. Each video gets split into chunks and stored accordingly.

The communication between the APIs and the databases is done via a connection string that contains all the needed information required to manage the databases. The ORM handles the rest of the modeling and the logic needed to prepare the queries that get sent over the network. All of the containers are connected via a network created by the docker compose command. Also, each container has their own volume, which is used to permanently store information on the host machine even if the containers are restarted or removed and recreated. This ensures that the data they have stored is not lost during those events and can be easily recovered. Another key point is that those volumes can be either versioned or sent over to other machines.

The advantage of this architecture is that it can be easily scaled horizontally by putting each component on its own machine. This ensures that resources are allocated properly where they are needed, and not spent on containers that do not use them. It works best if vertical scaling is possible too, meaning that containers that have high resource usage can be allocated more CPU, memory or bandwitdth.

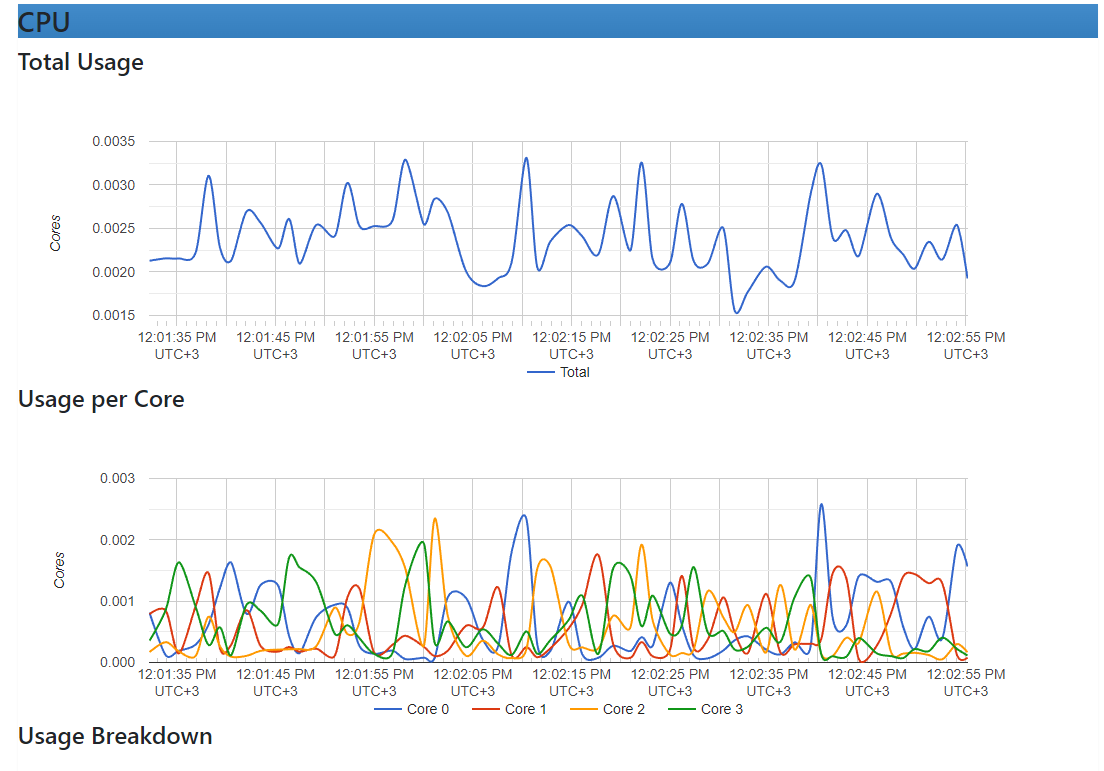


Figure 4 - RecSysAPI CPU usage seen in cAdvisor monitor

By observing **Figure 4**, we can see that even when idling, the system can have differnt spikes. This is particularly important on real applications when usage can go up at peak hours and down at night. Resources should be allocated only when the peak hours happen, to avoid crashes or low repsonse time.

The reverse of that is that at night, we should not allocate many resources because the application would not need them, and hosting costs would be high for no reason.

The **Prometheus, Grafana and cAdvisor** tools used to monitor the containers communicate with all the other containers on the local network inside the docker compose system. They poll usage data from each of the containers and output visible graphs towards the system administrator through a web page that can be accessed on the browser by navigationg towards a localhost port exposed via Docker.

## Application Architecture

The whole system is based on **microservices**8 architecture using **domain driven design**9. Microservices architecture (often abbreviated as **microservices**) refers to an architectural style of application development. Microservices allow for the separation of a large application into smaller independent parts, each with its own area of ​​responsibility. In response to a user request, a microservices-based application can call multiple internal microservices to compose its response.

Containers are an example of a microservices architecture that is great because they allow you to focus on service development without worrying about dependencies. Modern cloud-native applications are often built as microservices using containers that are tied together using **Kubernetes**.

Generally, microservices are used to speed up application development. Microservices architectures built with C# are popular, especially those integrated with the **.NET** framework. It is also common to compare microservices with a Service Oriented Architecture. Both have the same goal, which is to break down monolithic applications into smaller components, but they have different approaches. The main difference between the two approaches is scope. Simply put, Service Oriented Architecture (SOA) has enterprise scope while Microservices Architecture has application scope.

**Benefits of Microservices Architecture**

* It solves complexity by breaking down the application into a set of manageable services for faster development and much easier to understand and maintain.
* It reduces barriers to the adoption of new technologies because developers are free to choose the technologies that are appropriate for their services and are not limited by the choices made at the project beginning.
* Microservices architecture allows each microservice to be deployed independently. Thus, it enables continuous deployment for complex applications.
* Microservice architecture enables each service to be scaled independently.
* It enables each service to be developed independently by a team that is focused on that service.

**Drawbacks of Microservices Architecture**

* The microservices architecture adds complexity to the entire system because the microservices application is a distributed one. You must choose and implement a message-based or **Remote Procedure Call (RPC)**-based process communication mechanism and write code to handle partial failures and account for other failures in distributed computing.
* Microservices have a partitioned database architecture. Business transactions that update multiple business entities in a microservices-based application must update multiple databases held by different departments. Using distributed transactions is usually not an option and you will have to use a possible consistency-based approach, which is more difficult.
* Difficulty implementing changes that include multiple departments. In a monolithic application, you can simply modify the respective modules, integrate the changes and deploy them all at once. In a Microservices architecture, you must carefully plan and coordinate the implementation of changes to each service.
* Testing a microservices application is also much more complex then in case of monolithic web application. For a similar test for a service you would need to launch that service and any services that it depends upon (or at least configure stubs for those services).
* Deploying an application based on microservices is also more complex. A monolithic application is simply deployed on a set of identical servers behind a load balancer. In contrast, a microservice application usually consists of a large number of services. Each service will have multiple runtime instances. And each instance must be configured, deployed, scaled, and monitored. In addition, you will also need to implement a service discovery mechanism. Manual approaches to operations cannot scale to this level of complexity, and successful deployment of a microservices application requires a high degree of automation.

To summarize, a microservices architecture provides the benefit of adaptability to change and the power to change and add new modules in order to adapt to business needs, at the cost of complexity and development effort. The flow is usually to create a simple monolith application at first in order to accomplish the first business needs, deliver the first versions to the end-users, and collect feedback on the functionalities and the possible future of the application. After getting all that information, the best approach is to break down the monolith application into multiple microservices, in order to be able to adapt to more and more changes, to help the team grow and work together on the same project without breaking things. Some legacy modules can be replaced without breaking the whole system or affecting anything else. The flexibility microservices architecture offers is clearly worth the development effort in order to ensure the growth and the maintainability of the system. This results in better workflows and faster releases, accompanied by the satisfaction of the clients that get new functionalities and faster mechanisms.

### RecSys REST API Architecture

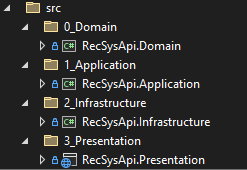
RecSys API architecture is designed in a decoupled, multi-level manner, each level having a clear and isolated task, respecting the principle of separation of concerns, such as database access or application logic management. I have chosen this architecture on several levels out of the desire to respect the principle of reversing the dependencies, so that the high-level modules, in the case of the Web controllers do not depend directly on low-level modules, for example, the domain services module, which gives access to the database, both depending on abstractions.

Figure 5 - RecSysApi projects overview

It strives to be a clean architecture, as Robert C. Martin states in his book, Clean Architecture, *“Good architecture makes the system easy to understand, easy to develop, easy to maintain, and easy to deploy. The ultimate goal is to minimize the lifetime cost of the system and to maximize programmer productivity.”10*because following this system leads to better organized code, decoupled modules that are easier to change and fix, and it helps promote **S.O.L.I.D.** principles.

The structure of the code, as seen in **Figure 5**, is composed of:

* **Presentation**, this is represented by an ASP.NET Core Web API project and contains the application start point (Startup class) and the controllers that expose the API endpoints;
* **Application**, it is represented by a .Net Core Library class library and contains the services that have the role of managing the application logic;
* **Infrastructure**, it is represented by a .Net Core Library class library and contains the implementations of some of the abstract Application and Domain services and repositories, and also the services that have external dependencies;
* **Domain**, it is represented by a .Net Core Library class library and contains the domain models of the application and the interfaces and services needed to manage the database.

The way dependencies flow follow the Domain-Driven Design, as seen in the **Figure 6**, meaning that the Application Layer depends both on the Domain Layer and the Infrastructure Layer in order to implement its interfaces, working with abstractions as fundamental basis. The Infrastructure depends both on the Domain Layer but also depends on external libraries, like packages imported with the **NuGet Package Manager**. The Domain Model Layer is free of any dependencies.

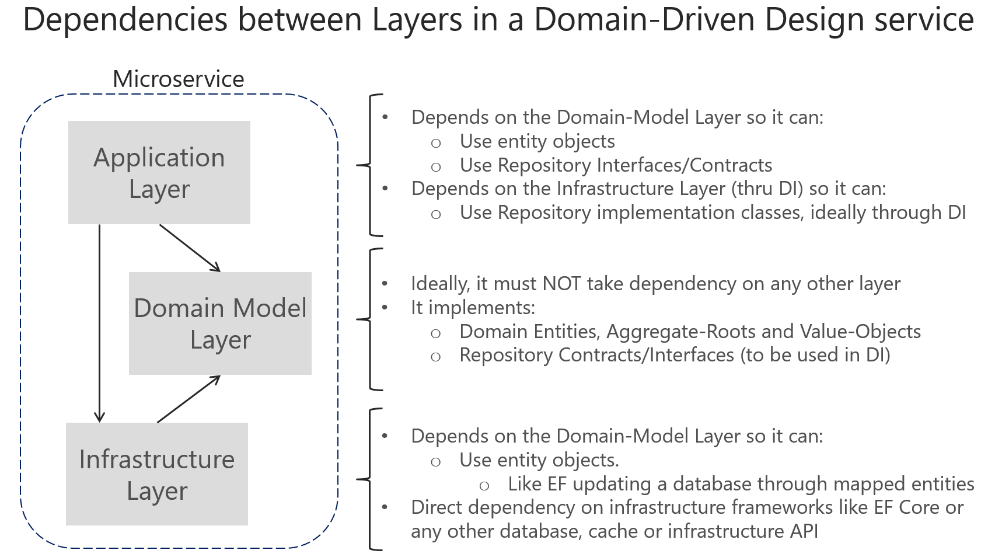


Figure 6 - Dependencies between layers in DDD

### RecSys Angular Client Architecture

The web application will give users the ability to register and authenticate directly to it. After this step, they will be able to search for a course, then if they wish they will be able to add it to cart and view and evaluate its content. Another feature that the web application will offer users is the ability to create courses each with their own defined sections, and add videos to each of the sections.

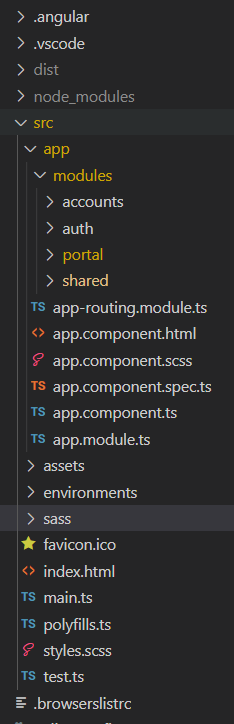
The structure of the client application is split into multple modules, alongside with the main app module and the main app routing module. Alongside the modules folder are the assets, containing all the static files included in the application, the environments folder which keeps all the definitions and configurations for development and production environments. The sass folder contains all the shared scss files that contain global scss variables, classes, mixins (that are style functions).

Figure 7 - Client application structure

Regarding the modules part, I have created the following, each with their own part to fullfill:

-Accounts module containing all the services, routing, pages and models needed to accomplish the login, account details and sign-up parts. It is not handling session or guards because those are handled in the auth part.

-Auth module contains all the logic that restricts the user to certain functionalities, regarding their role. It contains guards, which are small functions that get executed and return a Boolean telling if the user has access or not to a particular route.

-The portal module is the front part of the application, containing sections like search, admin, course, contact, home, owned-courses and services to manage admin functionalities. It also contains a shared part, that has partial components that are reused inside the portal module

-Shared module is a module that has parts that are reused throughout the entire application, not restricted only to the portal.

The shared module contains a navbar, a footer, web interceptors (important functions that get triggered before HTTP requests are sent or received, e.g. adding Auth token on outgoing requests) and most important services like the http service, the cart service, the search one and the notification service. Those have high potential for reusability; therefore, they were added to the Shared module. Also, the Shared module contains imports for Angular Material, used to style everything inside the application.

### SQL database

The SQL database is relational, contains 15 tables, listed below with all their properties:



Figure 8 - Database Overview

### MongoDB database

The SQL database is non-relational, contains 2 tables created with the GridFS framework, listed below with all their properties:

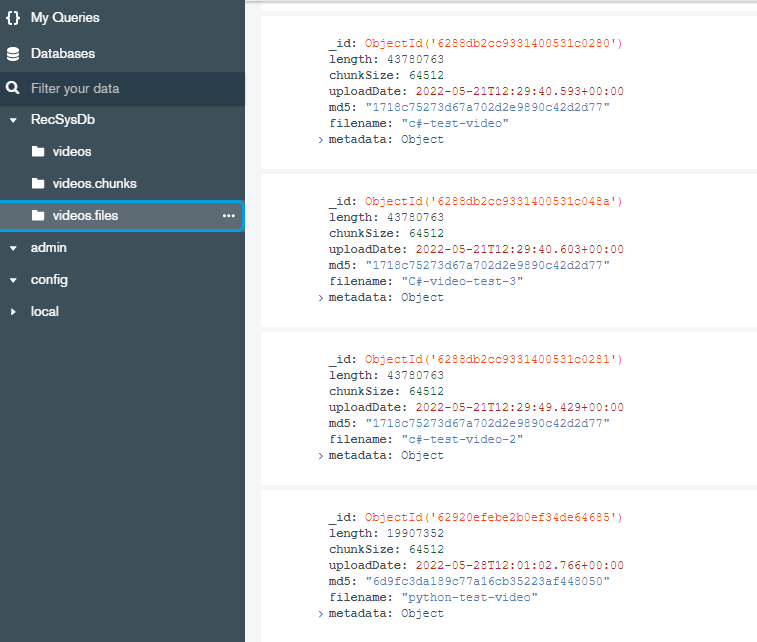


Figure 9 - Video Files Objects



Figure 10 - Video Chunk Objects

## Application Implementation details

### RecSys API

Starting wih the **Presentation** layer, the entrypoint of the application, containing the controllers responsible for handling user sessions, courses, videos and searches. Those controllers hold the logic of delegating work to the Application layer, which contains all the business logice needed to fulfill the goal of each action present on the controllers

A pattern that is consistently used in this project is the Dependency Injection pattern, an concrete example of this being shown in **Figure 11**.

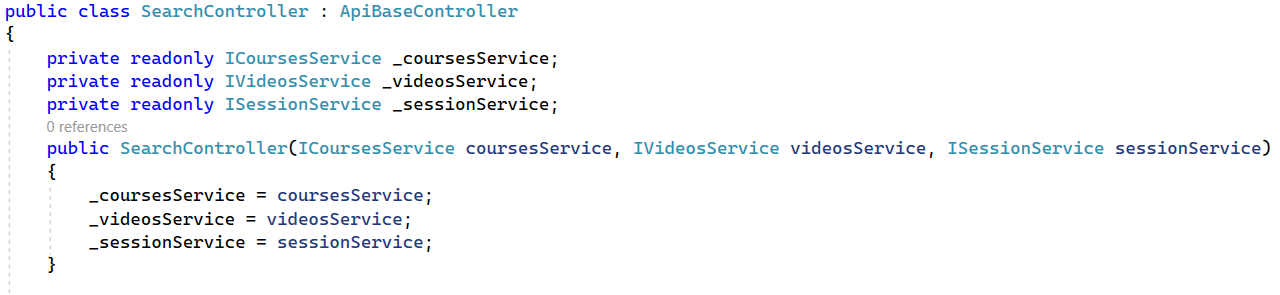


Figure 11 - Dependency injection in SearchController

**Dependency Injection (DI)** is a software design pattern that allows us to develop loosely coupled code, being a great way to reduce tight coupling between software components. Dependency Injection also allows us to better manage future changes and other complexities in our software. The goal of DI is to make the code maintainable, by allowing us to change only certain implementations, and not the methods above that use this functionality. By working with abstractions instead of concretions, we allow the logic to remain unchanged, even if we completely swap the implementation of the functions called.

In my case, SearchController needs courses, videos and session services in order to be able to process a search request and return proper results. The same thing applies to all other controllers. The controller also has the reponsability to handle pagination options, defaults if no values are provided, exception handling and formatting a proper response. It is also its responsability to define who has access to certain resources, what type of action it handles, the route it accepts requests to, from where to take the information needed to process the request, doing all of those by using attributes as seen in **Figure 12** and **Figure 13**.



Figure 12 - Search query action inside SearchController



Figure 13 - Controller with Policy Authorize Attribute

The **[Authorize(Policy = "AuthOnly")]** attribute tells the application that each of its child actions is protected from jwt tokens that don’t have this claim present. This is important because, for example, a user can not use a refresh token (which has a long life) as an auth token (that has a short life), therefore a refresh token can only be used to access the refresh action, which is protected against forgery using rotated refresh tokens and token invalidation. What that means is that the auth and refresh tokens are recreated each time a refresh token is requested. When that happens, the application searches for the latest generated token and sees if it is in the list of previously generated ones. If it is, then the current refresh and auth tokens are invalidated and the user is asked to log in again into the application. This whole system is intended to prevent session hijacking attack attempts and tries to be compliant with the **Oauth 2.0**[11] protocol.

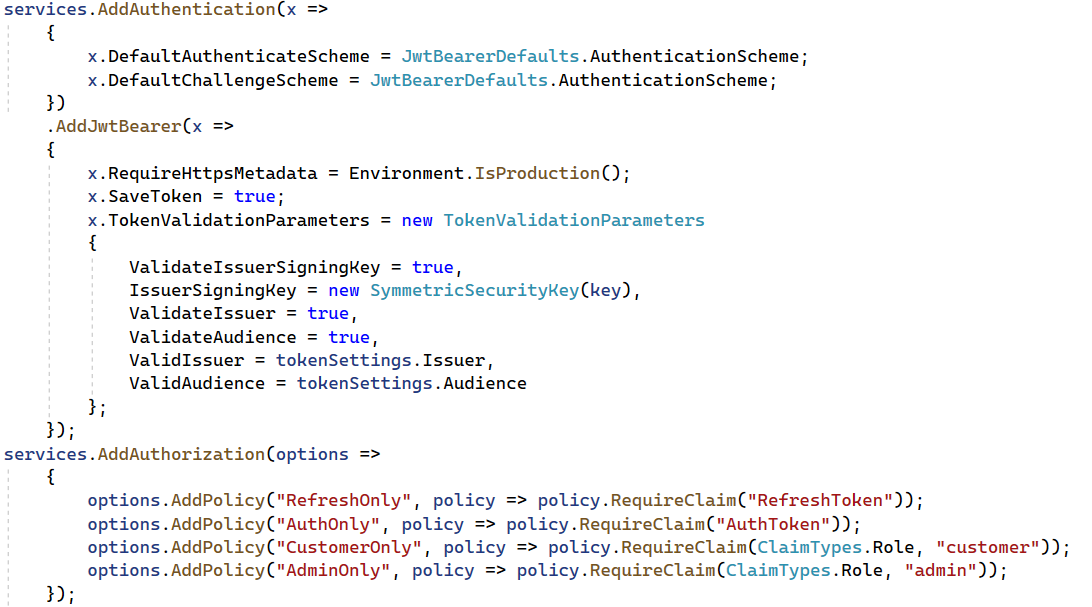


Figure 14 - Adding Auth settings to the API

**Figure 14** presents how Authentication and Authorization are set up, defining settings for the JWT and defining policies for the diferrent actions that may require Auth.

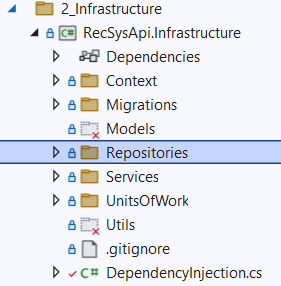
Going on with the **Infrastructure** Layer, we can oberve that here we have the database context, the EF Core Migrations, the repositories, the unit of work and the implementation for the http, session and videos storage services. What those services have in common is that they are tightly coupled with external dependencies, that is why they are found in the infrastructure layer.

Figure 15 - Infrastructure Layer overview

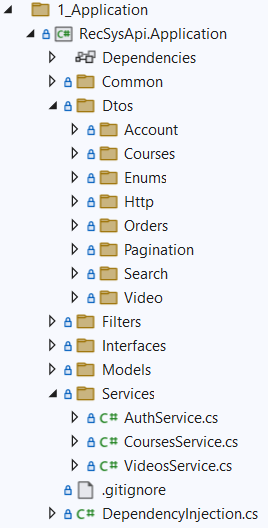
Continuing with the **Application** Layer, we can see that it contains common helpers and constants, **Data Transfer Objects (DTOs)**, Filters for search, Interfaces – which contain interfaces for services contained either in Application or in Infrastructure. This way, most services in the Application layer depend on abstractions, that are made concrete and injected into the API by the Infrastructure layer **DependencyInjection** class.

Figure 16 - Application Layer Overview

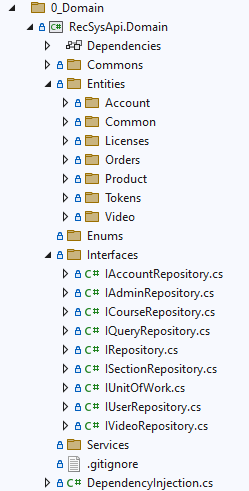
Because each call to a remote interface is expensive, response for each call must bring as much data as possible. Therefore, if you need a lot of requirements to carry data for a specific task, the data given can be combined in a DTO so that a single request can carry all the required data. Also, the DTO can protect sensitive information retrieved from the database, like if we want to retrieve user details, but not the password hash.

Figure 17 - Domain Layer Overview

The **Domain** Layer, the Core of this application, shared also with the OrdersAPI, contains all the classes requiered to interact with the database. Most important part are the entities, which define how the database is structured, and also they enforce the connections between data, the constraints regarding string sizes, nullability, values and others.

Sometimes it is necessary to define logic between our business logic and the data access logic. This logic can be defined using the repository model. Essentially, the repository model is an intermediary between the two classes. Repository models are mainly used when we need to modify data before moving on to the next step. Sometimes it is necessary or preferable to create all the repository logic in one place using common logic. We can only create one repository class, which will be responsible for managing the entire scenario. This could be the logic between the repository and data access, or the logic between the business logic and the repository. In any case, we only write one repository. The main advantage of the Generic Repository Pattern is code reusability.

I have defined repositories for various parts where I felt that I could add aditional logic besides the main Repository, which I extended in each of the repositories, in order to maximize reusability and to respect DRY.

### OrdersAPI

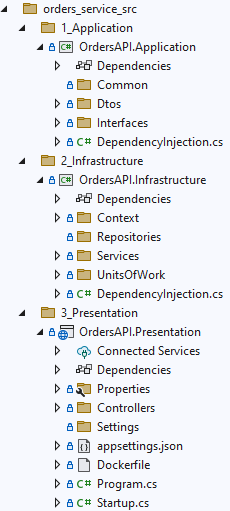
**OrdersAPI** is structured following the design patterns and architecture of the **RecSys API**, depending on the domain part of the RecSys so that the SQL database remains consistent across both of the applications. Its layers have the same logic behind them and the same structure for classes. It is not as complex as RecSys, because I wanted it to have only the role of processing orders and creating orders for courses. In a real life application, this microservice could handle a lot more functionalities, like having an invoice generation function, or an email service that notifies the users when an order has been processed successfully.

Figure 18 - OrdersAPI Overview

### RecSys Client

The application will be built using the architectural design style “single page application” **(SPA)** with the help of the **Angular 13** framework together with **RXJS** for state management. Front-end style and implementation of controls will be based on **Angular Material**.

The application is split within multiple modules, each handling its own part of the application. Accessing each part is called routing, ensuring that components are created and destroyed without having to refresh the web page.

Routing in the application is handled by each module, lazy loading being used in order to improve the loading speed of the application. Lazy loading is very important because of the following reasons:

* **Response Time**: This is the time it takes for the web application to load and for the UI to respond to the user. Lazy Load optimizes response time by decoupling code and loading the desired package.
* **Resource Consumption**: If a web page takes more than 3 seconds to load 70% of users will give up. Web apps must not take long to load. So to reduce the number of resources loaded, lazy load loads the required bundle of code at a time.

Every route that should not be accessed by a user who is not allowed to do so is protected by a special Angular service called Guard, which prevents the route from being activated and loaded properly. Each outgoing HTTP request will be wrapped with an interceptor that has the role of adding the authentication header containing a JWT as the identity provider.

## Hosting and containers orchestration

Hosting the application is an important part of releasing the application. It has to be planned beforehand and especially in the case of microservices, templates have to be defined in order to take the application from a bunch of code to a completely functioning application that runs on predefined environments. Docker helps us do just that through the use of Dockerfiles and Docker compose command. Simply put, Dockerfile is the template needed to be run in order to build the application on every environment that supports Docker, by simply emulating the needed Operating System functions. It will be described below in more detail.

### Docker

A Dockerfile is a text document (without a file extension) that contains the instructions to set up an environment for a Docker container, a Docker image can be built using a Dockerfile. The Dockerfile represents a number of steps to be followed in order to prepare the image. The Docker image is like an application that is ready to be run on every machine, starting from the same startup state. It is the starting point for containers. In order to be able to prepare the application to be run, the docker file has to start from an image hosted on Docker Hub or other hosted images from the internet. Those starting images can be anything from a minimal version of an operating system with minimal packages to full grown operating system with all capabilities installed. The former is almost always prefered because it consumes minimal resources and should contain only the dependencies.

In order to increase image build performange, hashes of all steps are generated, so in case the Dockerfile is changed, only starting from the first differnt hash the steps are rebuilt, otherwise they are reused from internal Docker cache.

A Dockerfile built for the RecSys API can be observed in **Figure 19**:

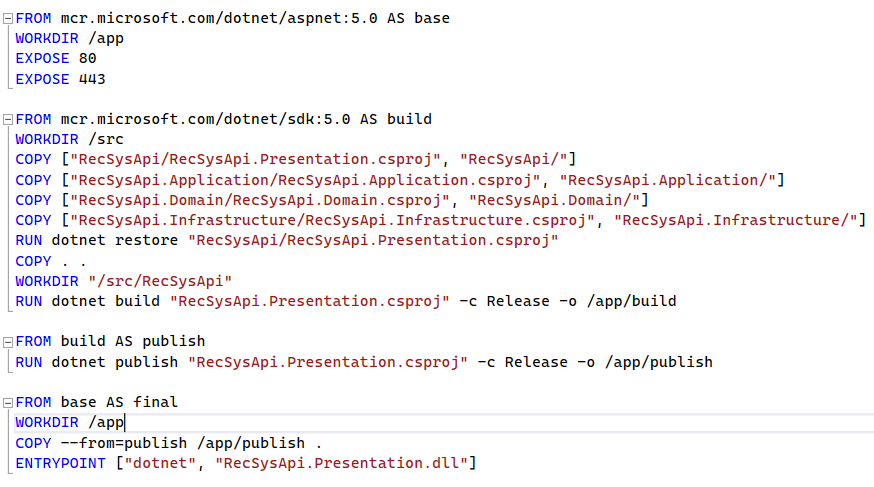


Figure - RecSys API Dockerfile

What this **Dockerfile** does, in the following order, is to fetch a base **ASP.NET** image from the internet, setup a workdir inside the image called /app, inform external agents that the ports exposed by the application are 80 and 443. The next from creates another layer, which contains the dotnet **SDK** needed for building the application. It sets up a folder called /src and it copies all the layers from the current host directory. Then it restores the main presentation project using a dotnet command. It sets up a workdir inside RecSysApi and starts a build with the Release flag. It then creates a new layer called publish and starts a dotnet publish.

In order to improve memory usage, the base layer is reused and the build and publish layers are automatically discarded, because they are no longer relevant for the application. Using the ASP.NET image, the final application is defined as the entrypoint the command:

***dotnet RecSysApi.Presentation.dll***

When starting up the container, that command is executed and the application gets up and running inside that container. Having this built-up image ensures us that whenever we need to get a new instance, we can easily just start up a new container from the same image. Adding a load balancer and scaling the containers to multiple machines ensures us that the application can be available at the busiest of times, handling easily any peak hours. Then after the peak hours have passed, the containers can be stopped, no longer using up machine resources.

The same structure is used on the Dockerfile used to create an image for the **Orders API**.

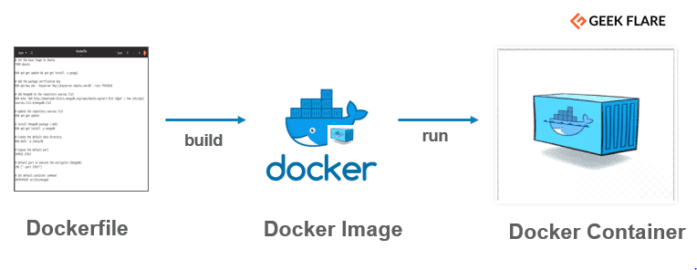


Figure - Dockerfile to container

### Docker compose

**Compose** is a orchestration tool that makes it easy to build multi-container distributed applications with Docker. It's easy to define a multi-container application and its bundled services in a single file, then launch it in a single command that will get an entire application up and running.

It sets up the network, the volumes, it instructs images what dockerfile to use and containers what settings to have. An concrete example of docker compose usage in my application is presented in **Figure 20.**



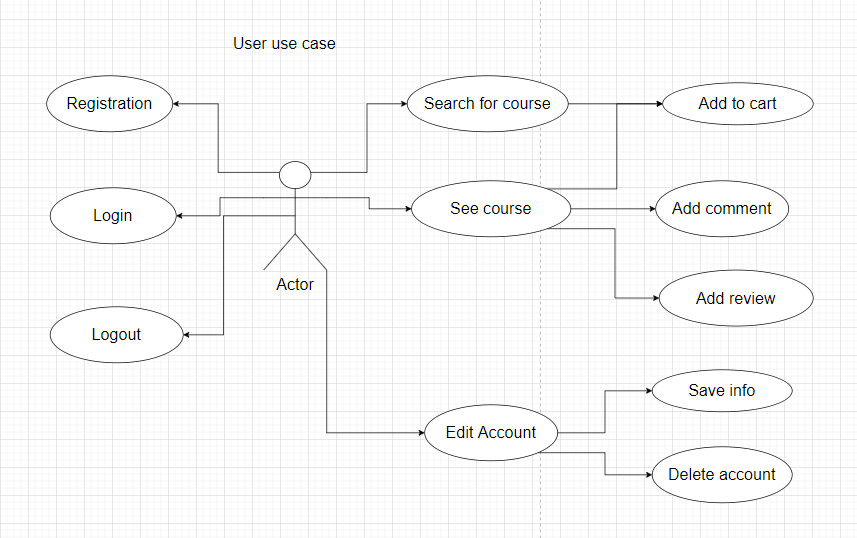
Figure - Docker Compose file for the RecSys System

# System Functionalities

### Use cases

Determining how a customer will interact with a product or system is essential for requirements gathering and high-level communication with stakeholders. A use case diagram is a visual representation of the product users expected interaction with the product, and what the product can do.

Use case for different flows:



* 1. The system displays the main page and the user can access the search bar in order to search for the courses
  2. The user starts typing and predictions appear to help him.
  3. After writing key words for what he is searching, he can press the search button or enter.
  4. A list of courses is shown and also more filter options to choose from to help him find what he wants faster and also to make the courses more suitable for him
  5. He can scroll down to search for courses
  6. Can choose one to see it and can also come back to the search page.

Alternatives:

* 4. If there is no course matching the searched words:
  + 4.1. There is shown a message that informs the user that no courses were found.
  + 4.2. The user can press again on the search to modify it
  + 4.3. The user redoes the process from step 2

Esential use cases regarding searches and purchases are presented in **Figure 21**.

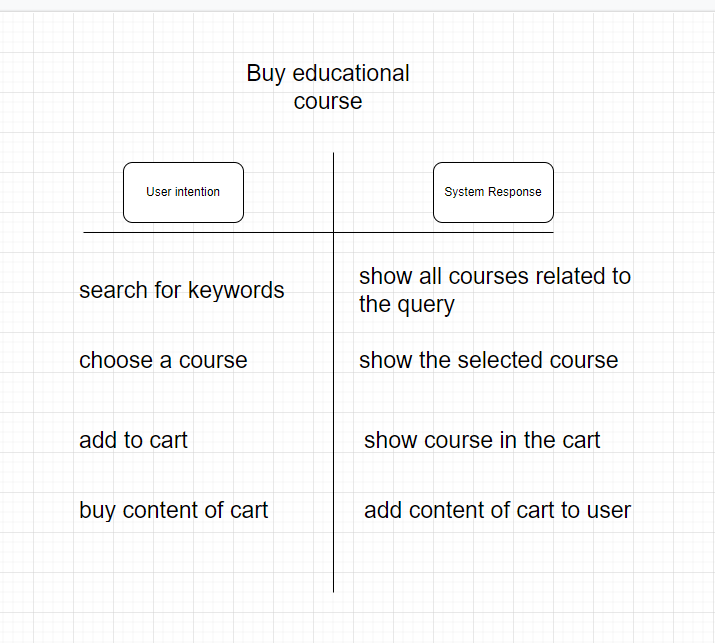


Figure - Essential Use Cases

### System functionalities

Concrete example of the implementation of the search use case, inside the application can be seen in **Figure 22** and **Figure 23**

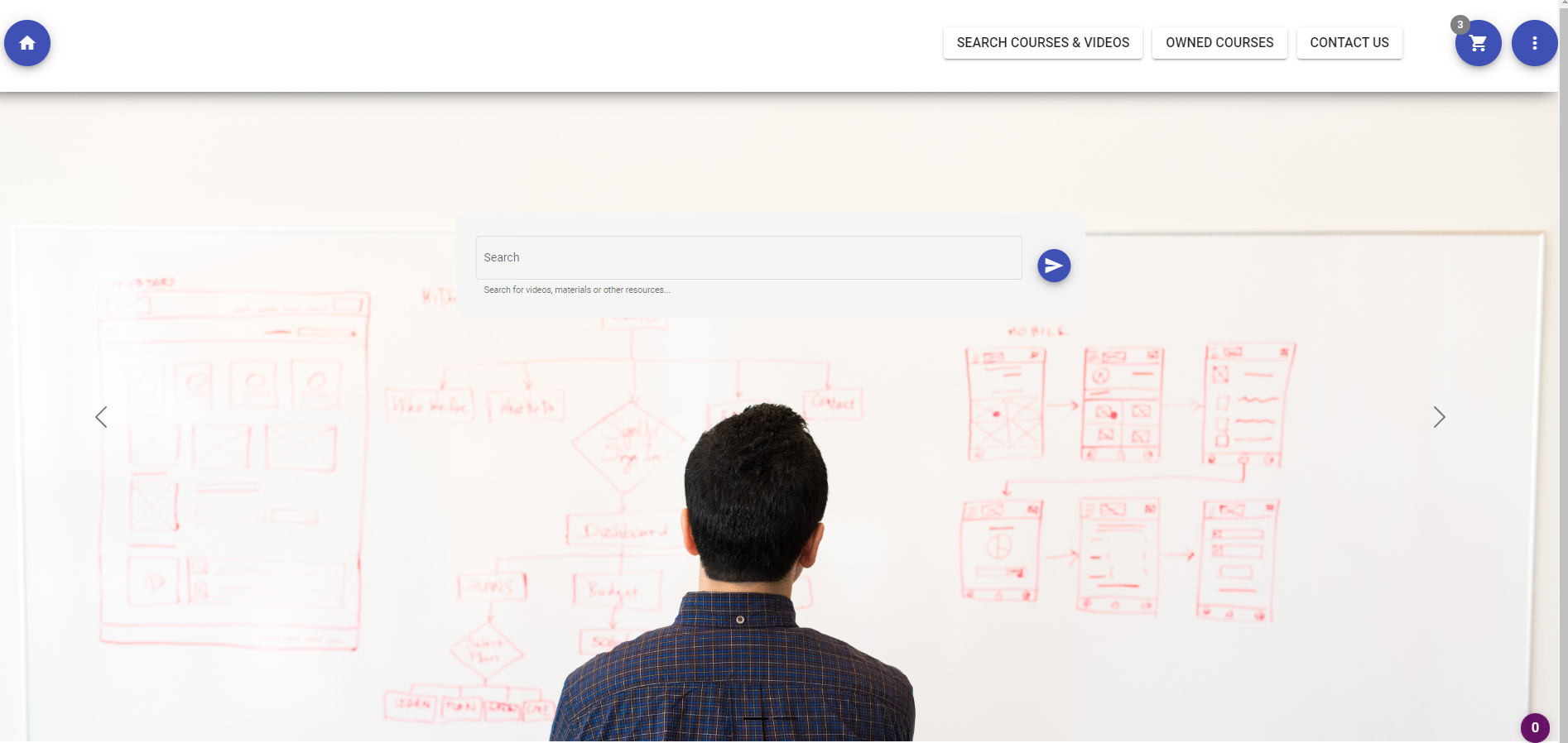


Figure - Home page

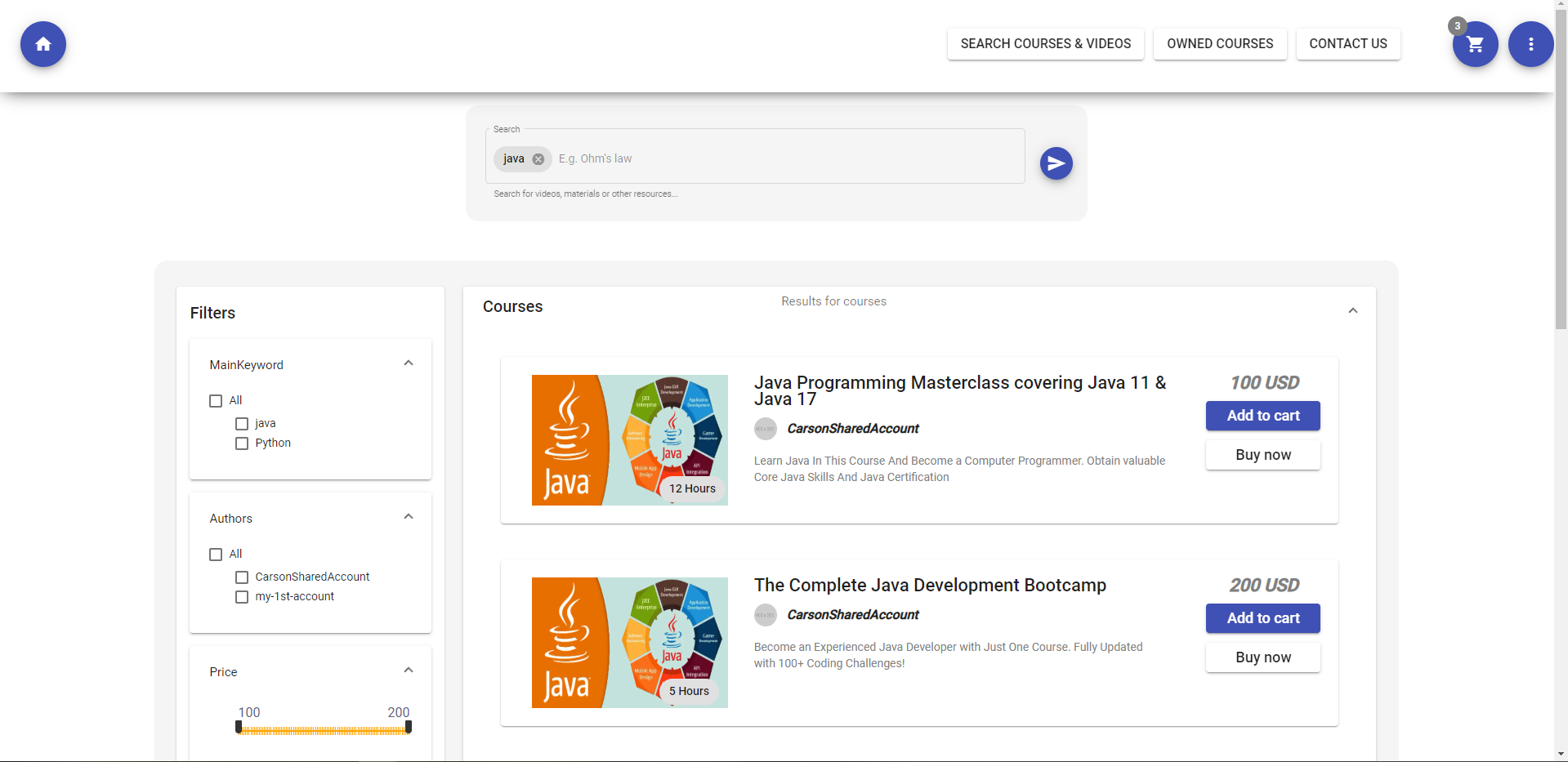


Figure - Search results page

When a user sees a course that interests him, he can access it in order to see details, like sections overview (containing all the videos presented in the course) and a short overview of the course contents. From this page the user can choose to either add the product to cart or to buy it now. A picture containing the overview of a course called C# Basics for Beginners can be seen in **Figure 24**.

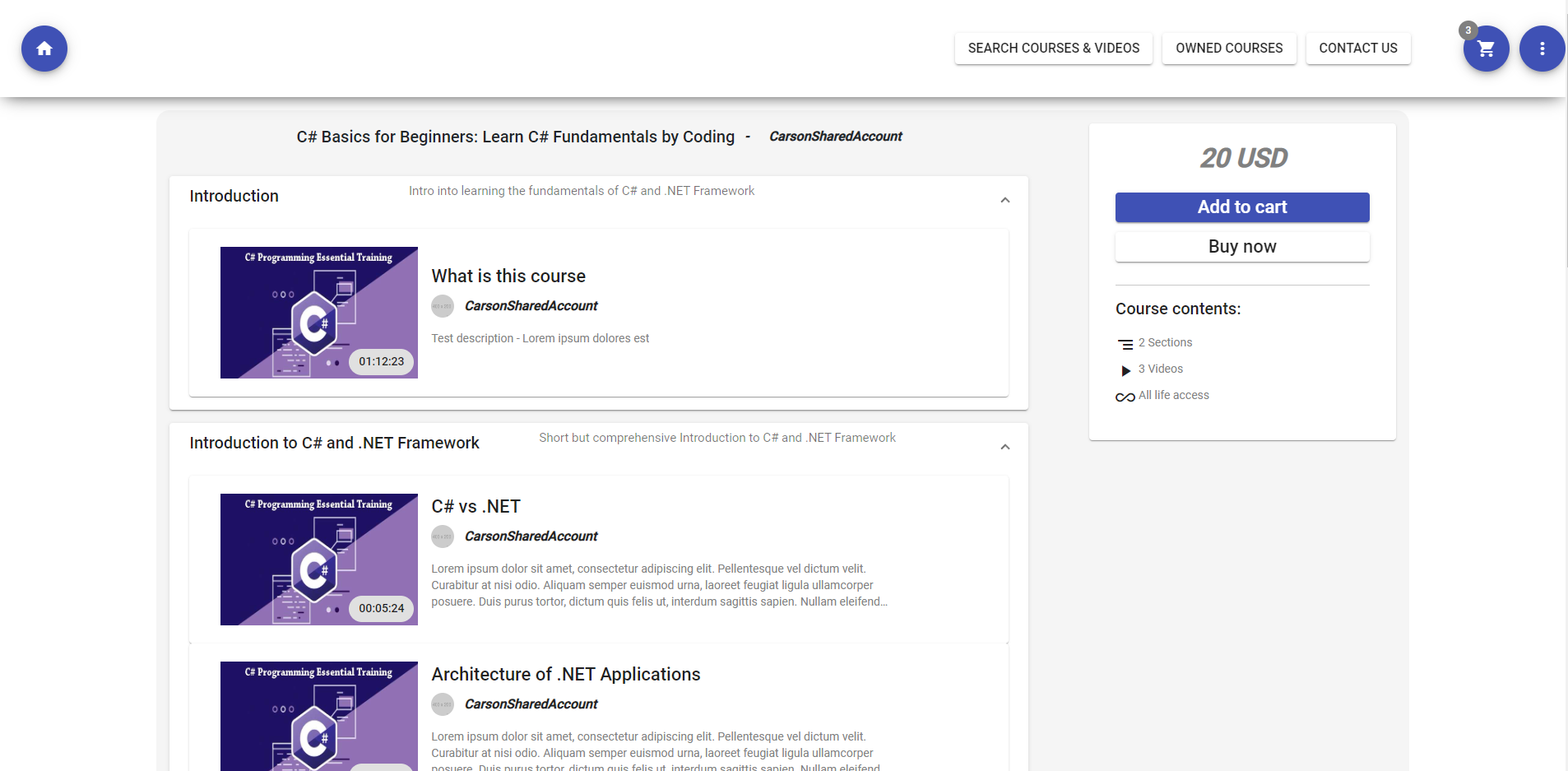


Figure - Course overview

After adding the courses to the cart, the user can review the contents of the cart and adjust it according to his wishes. The user can remove courses that he doesn’t like and when everything is ready to be purchased, the user can press the checkout button. In a production environment, this should redirect to a payment processing engine. In my application, it is simulated that the payment has been processed successfully and course licenses are generated for each course.

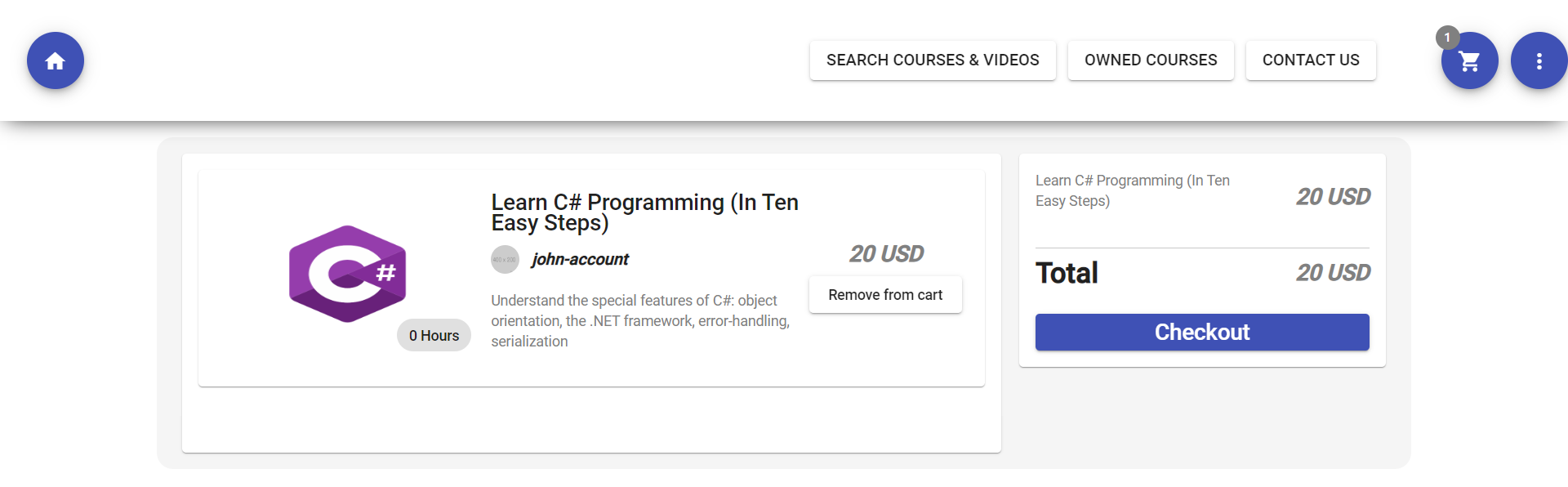


Figure 26 - Cart page

When courses are purchased, the user can see his courses by clicking on the owned courses button. There he will see a list of all the courses that he has access to.

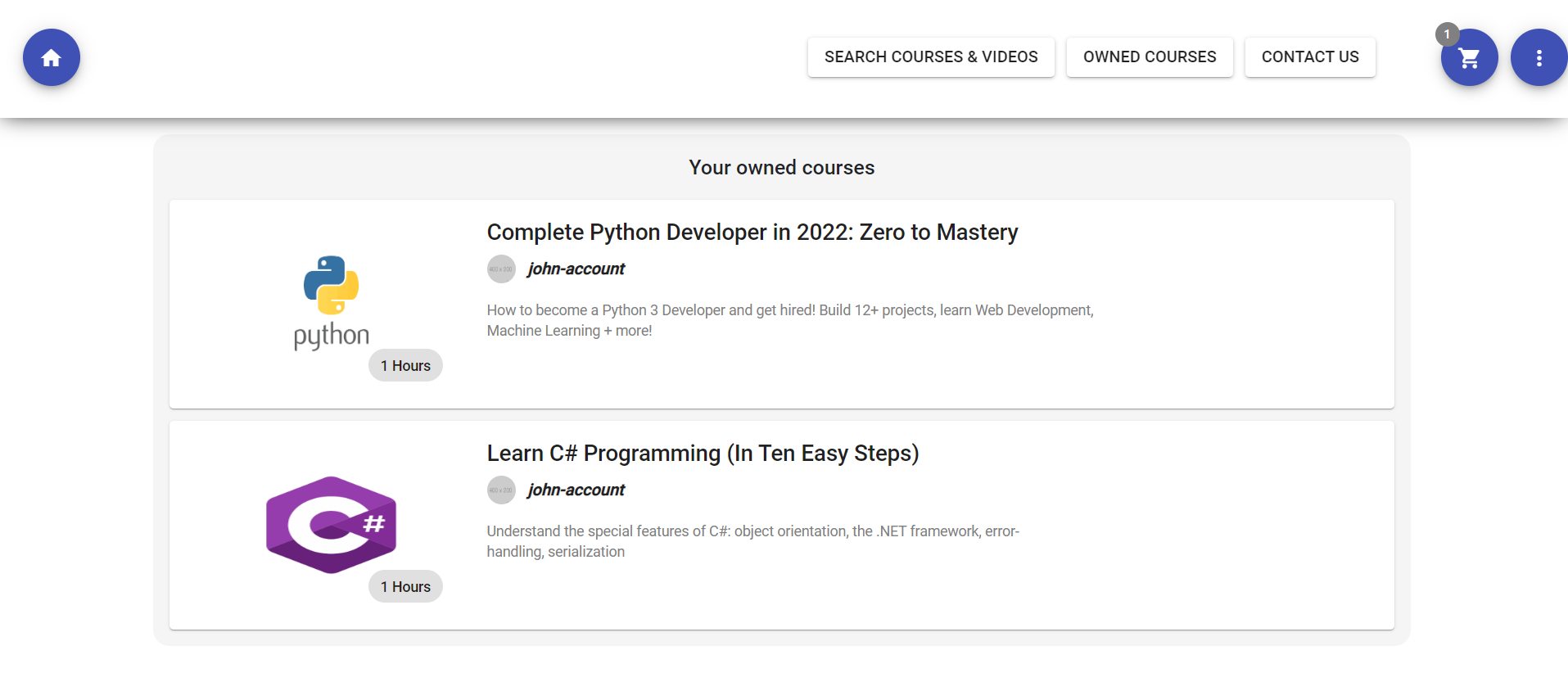


Figure 27 - Owned courses overview

After clicking on an owned course, the user can see the course overview and details and select videos from sections. The watch video page can be seen in **Figure 27**.

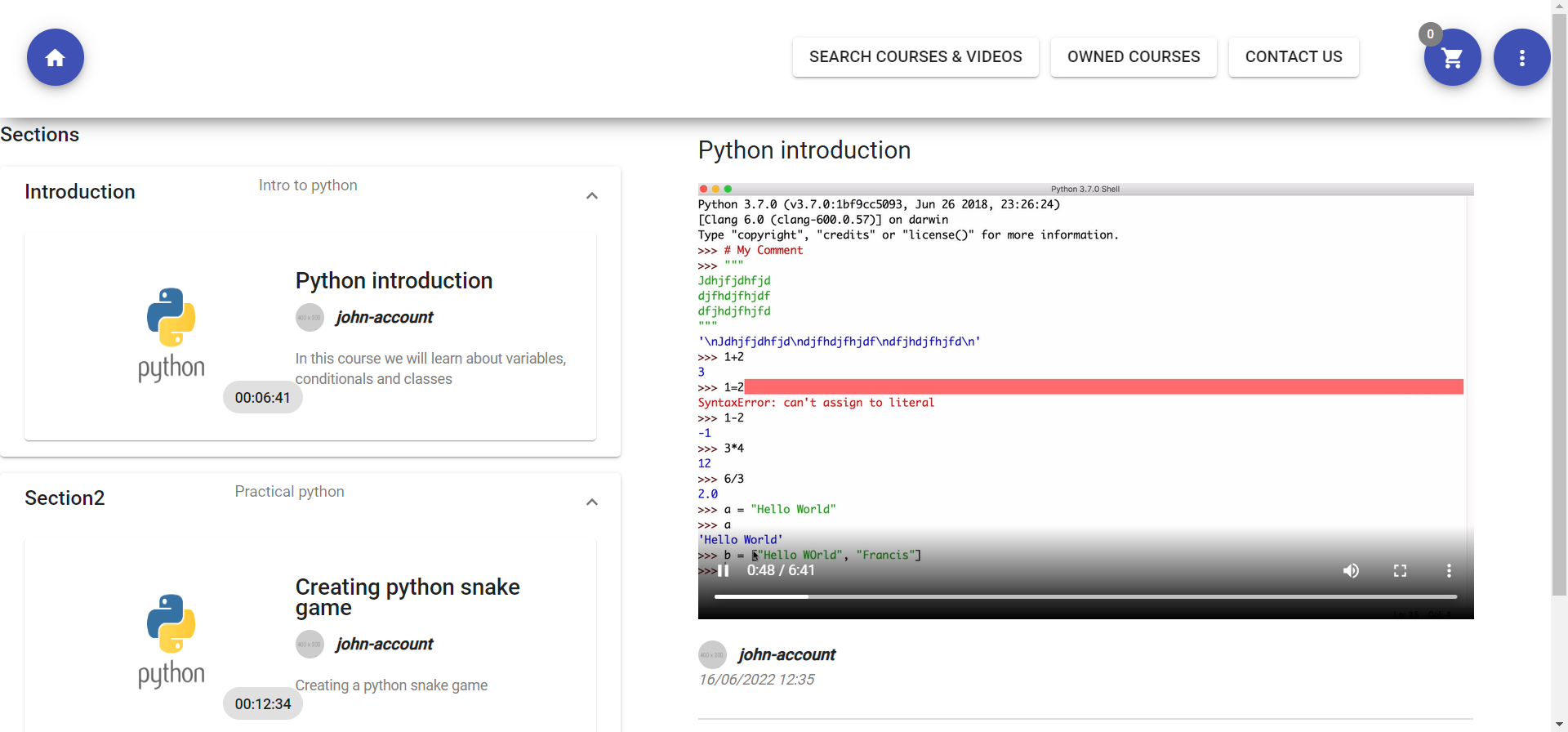


Figure 28 - Watch video page

In the end, this flow shows how the user can go from not having any course to viewing videos after purchasing them. It is very important because it must be maintained as simple as possible, meaning that there should not be any loops in the flow or any disruption that might get the user from accessing the desired content. The way the UI is designed is very important for a pleasant user experience.

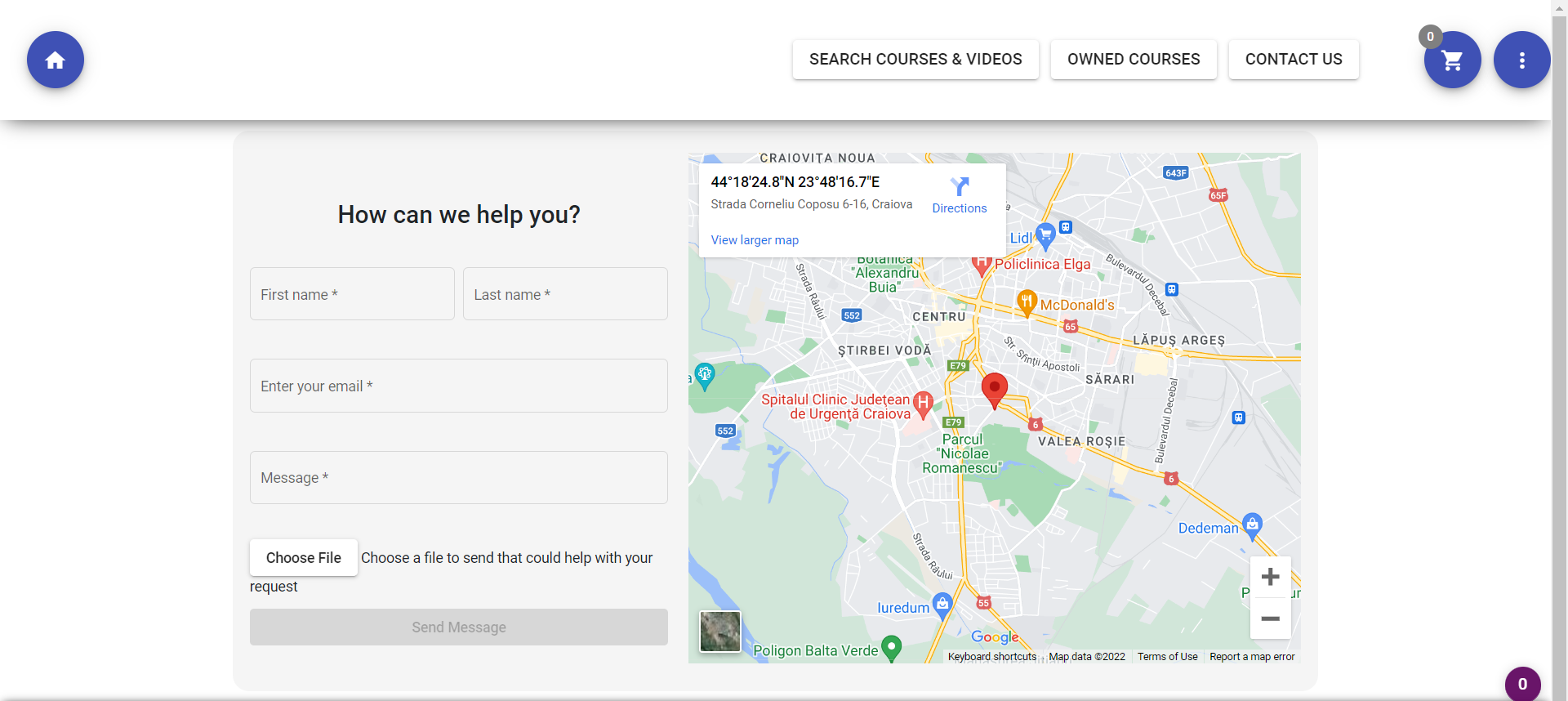


Figure 29 - Contact us page

In case a user has any problem, a support request can be easily sent to the application via the Contact Us page, presented in **Figure 28**.

An important part of this application is adding content, a feature that only admins and publishers can use. It can be seen in **Figure 29**, **Figure 30** and **Figure 31.**

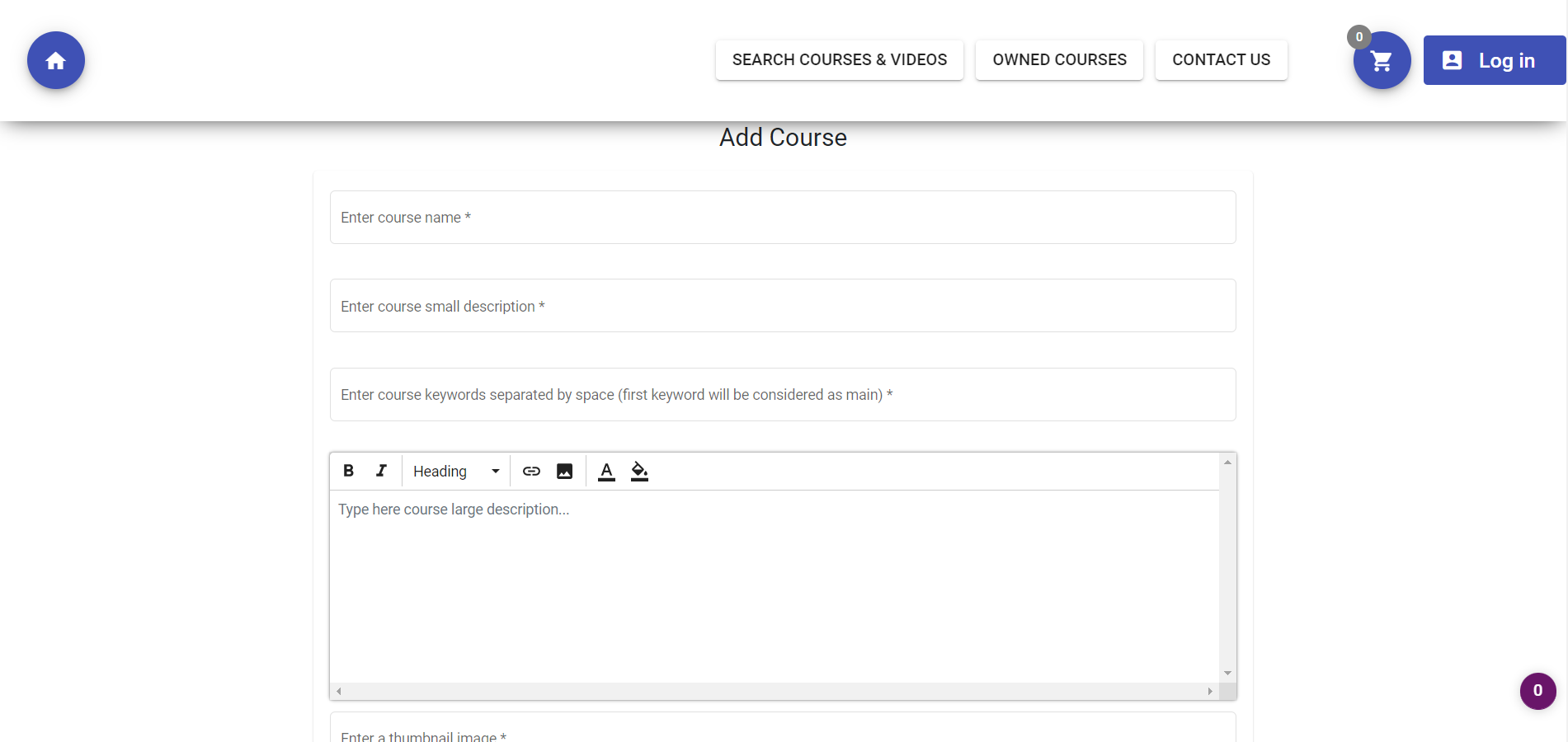


Figure - Course details

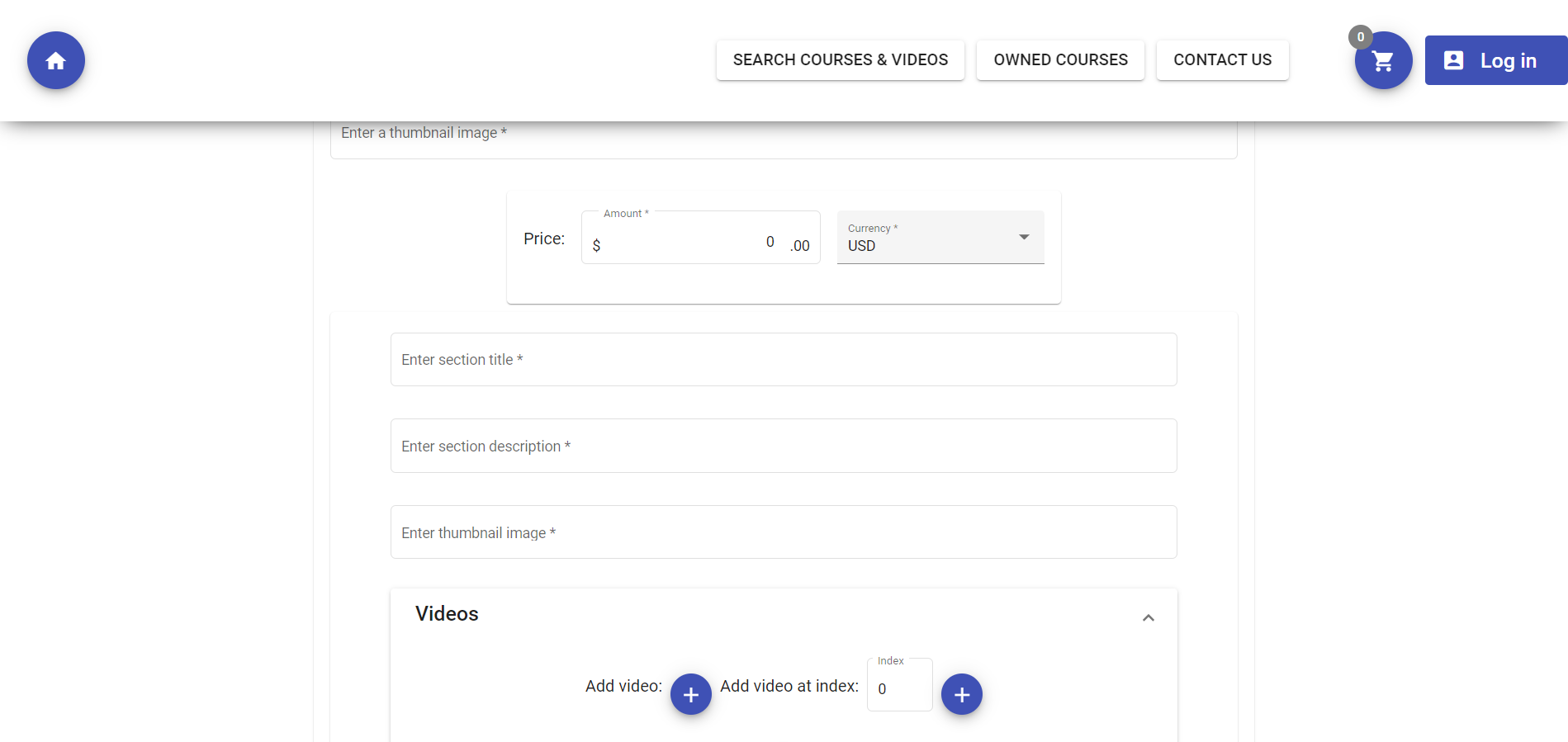


Figure - Section details

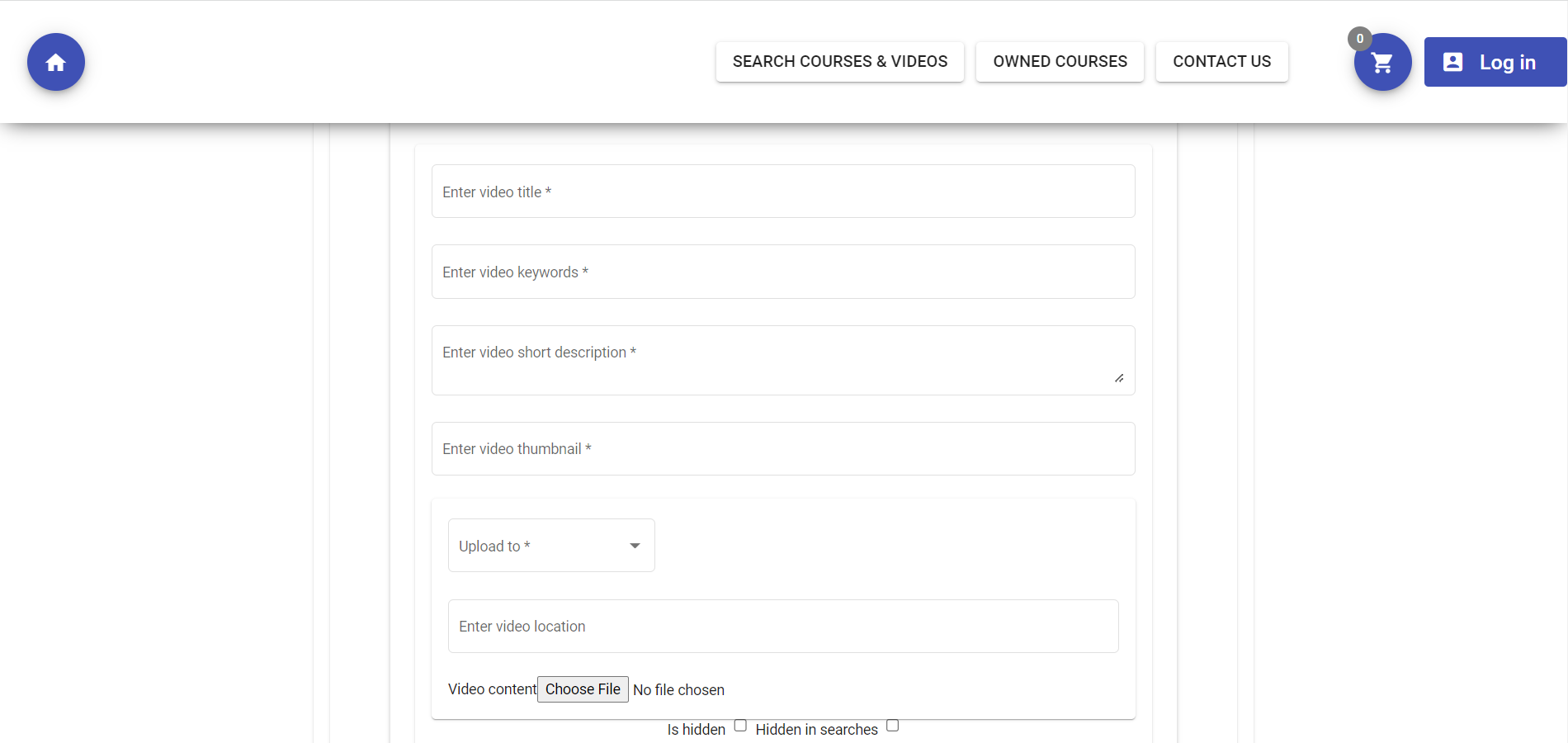


Figure - Video details and content upload

**Conclusions**

The architecture of the system has been designed with an emphasis on extensibility, maintainability and modularity. This is in line with the trend of modern applications developed to run in the cloud, the main benefits being the uninterrupted availability of services, the ease of accessing these services, can be accessed from any modern browser and any operating system and high speed access to this is due to the globalization services provided by the platforms where it is published.

This solution was developed using the latest and most appreciated technologies for each of the applications, in line with the trends in the production software environment from the moment it was developed.

The future aims are to: integrate the system with a payment system, publish the project in production, switch from the file system, used to save images and videos, to a solution in the cloud, for example using AWS storage services (S3 bucket ) or Azure blob storage, translating the web application into multiple languages ​​to make it more user-friendly in different countries where native language is not English

This project has been a great learning oportunity, as I’ve strived to study as much as possible in order to achieve the goal of making everything work with the latest technologies. The way technology advances in this field is amazing, and keeping up with everything is always a challange. In the end, I am glad that I have managed to complete this project and that I’ve learned a lot of things in the process.

# A. Bibliography

* Clean Architecture – by Robert C. Martin, Pearson Education, september 2017
* Microsoft, Overview of ASP.NET Core, 2018, <https://docs.microsoft.com/en-us/aspnet/core/mvc/overview?view=aspnetcore-2.1>
* Angular Architecture overview, <https://angular.io/guide/architecture>
* Udemy business and revenue model, <https://futureworktechnologies.com/how-udemy-works-business-revenue-model>
* Microsoft, Introduction to the C# Language and the .NET Framework, 2018
* Adam Freeman, Allen Jones, Programming .Net Security, 1st Edition, 2003
* Erich Gamma, Design Patterns – Elements of Reusable Object-Oriented Software, 1995
* Nicole Legault, What is E-Learning, <https://community.articulate.com/series/getting-started/articles/what-is-e-learning>
* Programming C# 8.0: Build Windows, Web, and Desktop Applications - Ian Griffiths, editor O’Reilly 2020

# B. Web references

[1] – Entity Framework Core

<https://docs.microsoft.com/en-us/ef/core/>

[2] – Docker Engine Components Flow

<https://docs.docker.com/engine/images/engine-components-flow.png>

[3] – IBM Cloud Learn Hub - What is Docker?

<https://www.ibm.com/in-en/cloud/learn/docker>

[4] – MongoDB

https://www.mongodb.com/why-use-mongodb

[5] – GridFS

<https://www.mongodb.com/docs/manual/core/gridfs/>

[6] – MongoDB Compass

https://www.mongodb.com/products/compass

[7] – Git

<https://git-scm.com/about>

[8] – What is microservices architecture?

<https://cloud.google.com/learn/what-is-microservices-architecture>

[9] - Design a DDD-oriented microservice

<https://docs.microsoft.com/en-us/dotnet/architecture/microservices/microservice-ddd-cqrs-patterns/ddd-oriented-microservice>

[10] – Clean Architecture – by Robert C. Martin

<https://www.amazon.com/Clean-Architecture-Craftsmans-Software-Structure/dp/0134494164>

[11] – Oauth 2.0

<https://oauth.net/2/>

[12] – EF Core Full Text Search

<https://www.bricelam.net/2020/08/08/mssql-freetext-and-efcore.html>

[13] – Full-Text Search, Microsoft

<https://docs.microsoft.com/en-us/sql/relational-databases/search/full-text-search?view=sql-server-ver15>

[14] – Automapper with neted typess

<https://dotnettutorials.net/lesson/automapper-with-nested-types/>

[15] – Angular file upload, Angular University

<https://blog.angular-university.io/angular-file-upload/>

[16] – Monolithic vs Microservices Architecture

<https://articles.microservices.com/monolithic-vs-microservices-architecture-5c4848858f59>

# C. Source code

The source code for this application is found on this GitHub repository:

<https://github.com/RoVRe232/E-LearningRecSys/tree/develop>