RAWDATA

Final Report

Front-end

DECEMBER 2021

GROUP 10

DELEBECQUE Alexis
DUMONT-ROTY Loïc
SHOWIKI Ali Isac

RUC

PART 1: SUBPROJECT 3 REPORT

CONTENT

PART 2: REFLECTIVE SYNOPSIS

CONTENT

INFORMATIONS AND LINKS

- GROUP MEMBERS

DELEBECQUE Alexis - DUMONT-ROTY Loïc - SHOWIKI Ali Isac

- GITHUB LINK FOR THE SUBPROJECT 1

https://github.com/elmuobus/RAWDATA1

- GITHUB LINK FOR THE SUBPROJECT 2

https://github.com/AlexisDelebecque/Portfolio_subproject_2

- GITHUB LINK FOR THE SUBPROJECT 3

https://github.com/elmuobus/RAWDATA3

PART 1: SUBPROJECT 3 REPORT

A QUICK SUMMARY

The goal of the RAWDATA project is to create a tool to search, consult, rate and compare movies, TV series, video games or actors. It is inspired by the Amazon platform called IMDB which allows to restore a large amount of information on cinema, television or video games. We will come more in detail on the purpose of the project later in the report.

The project is divided into three sub-projects:

The first one consisted in creating the database of the tool where will be stored both our raw data that we had retrieved via the API of **IMDB** and **OMDB**, an open-source database, and at the same time the data of our users and our functionality. Thus the database is split into a Movie model, a User model and has functions that will perform different tasks on our database. We can, for example, mention the history recording function which will add as a string everything the user types in the search bar or the rating functions which will calculate the average rating of a title (Movie, TV series, video games) in relation to the number of votes. The database was built using **PostgreSQL**, an open source relational and object database management system (**RDBMS**).

The second sub-project consisted in managing the Backend part, that is to say the API that makes the link between our database and our application. It is the API that will take care of sending the data requested by the user and to analyze what the latter sends us according to his actions. As for the database, it is separated between the Movies domain and the Users domain. We have used the C# language with the .Net framework to realize this webservice.

Finally, the third sub-project consists of creating the frontend, i.e. the user interface. Globally, this is the part visible to our customers who can use it to access the data. Through this interface, the user can see the titles he is interested in, create an account, perform searches, bookmark titles and much more. In short, it is on this interface that we will find graphically all our functionalities that we have developed throughout the sub-projects. This tool has been developed in HTML, CSS and Javascript using the Knockout.JS framework which is an implementation of the MVVM (Model-View-Model) pattern via the use of templates. As a reminder, this template allows to easily separate the development of graphical user interface by separating the logic from the view, that is to say from what is displayed on the screen.

THE IDEATION PHASES

For this first phase of the development of our tool, we wanted to find a design that would offer the best user experience to the customers (i.e. users) who will use our application. As a reminder, User Experience (abbreviated as UX) defines the quality of the user's experience in digital or physical environments, i.e. how a user will live and experience an interface with which he interacts. This notion was very important to us for several reasons:

First, for most of the project members, we had the opportunity to work in companies that put forward this notion to create their product. Thus, we have incorporated this notion in our work process during the realization of projects and we make it a point of honor not to create a tool taking into account only for its technical aspect, but also the way our product will interact with users.

Then, in parallel to the courses allowing us to give us the keys to realize the portfolio project, we followed an Elective course dedicated to the User Experience. It therefore seemed interesting to us to take some of these teachings and concepts out of their theories and incorporate them into a project which, de facto, is more practical.

Finally, since this project is meant to be a Portfolio project, that is to say a project that we can be brought to a potential employer or simply to add on our showcase site in order to show our skills, we wanted to put ourselves in a business situation by taking into account the different steps of the development of a project.

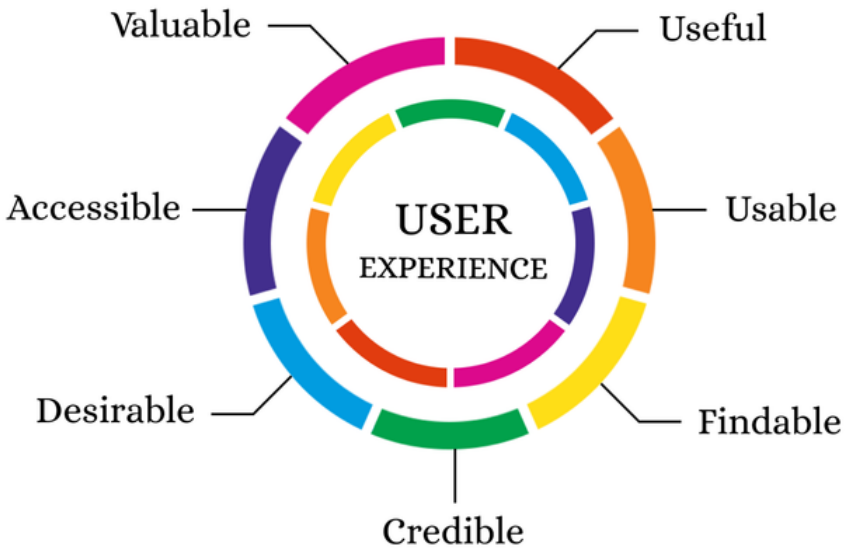
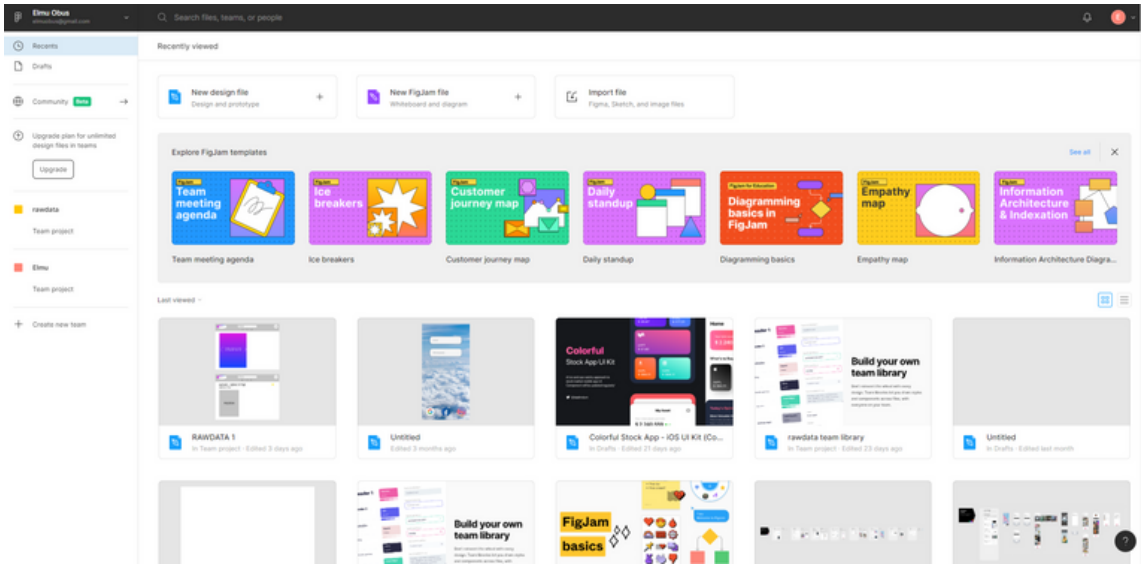


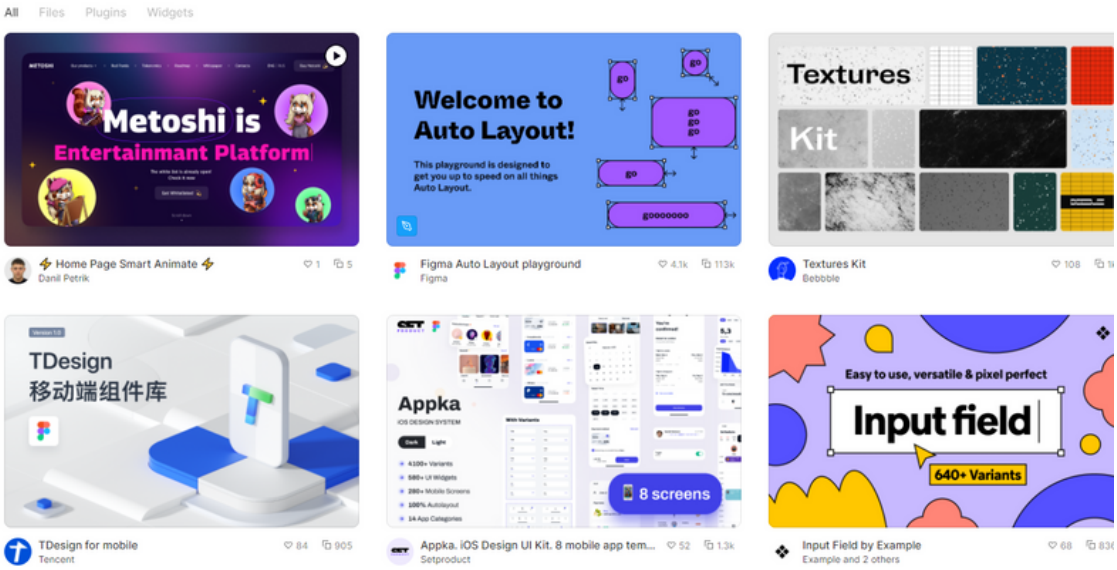
Diagram showing the different criteria necessary for a good user experience when creating any interface.

Thus, this first part was based on the creation of a model of our interface. To do this, we used the vector graphics editor and prototyping tool Figma, which is a very famous collaborative design tool in the world of application design. The advantage is that, once the mock-up is finished, we could translate each element of a model into a line of code in order to save time during development and to create better quality interfaces.

The particularity of Figma is that it offers, in addition to creating free models with a powerful tool, access to a tab called "Figma Community". This tab gathers templates, widgets and plugins made by the community using this tool which allows to have access to prototypes made by UX designers or graphic designerswho have been working in this field for several years and who will give illustrated advices on how to make such or such templates based on their experience.



Screenshot of the Figma homepage showing the different features offered by the tool to create models.



Screenshot of the Figma homepage showing the community page. You can find different elements like application templates, tutorials made by designers or graphic elements in the form of kits that can be reused for free for our prototypes and applications.

To build our tool, we decided to base it on a mockup of the codes and methods of the Netflix streaming service, which was created by a UX designer. This mockup was interesting because it offered us a whole collection of views for almost all the steps we would have to take to realize this project.

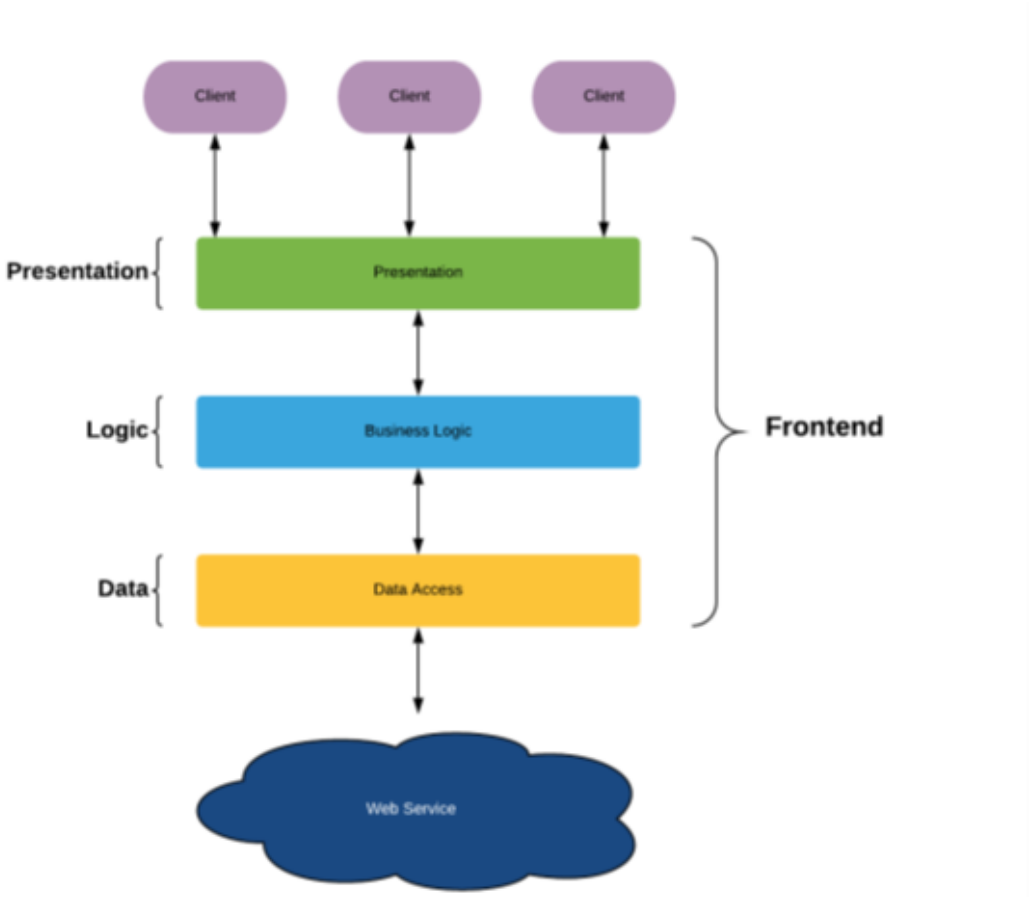
So, with the help of these templates, the different parts that we had made previously and the roadmap listing all the points to achieve that we follow since the beginning of the project, we have made a model that will serve as a step to achieve our frontend.

Screenshot of our model, we can find a tree showing the different templates needed for our application that are linked together. For more clarity, each link corresponds to the path made by the application when the user will make such or such action.

THE DEVELOPMENT PHASES

Now that we have our model ready, it was time to start developing our tool.

We chose, in accordance with the subject and what the knockout.JS framework allowed us, to develop our application via the MVVM (Model-View-Model) pattern. By separating the data, the logic and the views (what is displayed on the screen). The interest is to have a clear code, easily maintainable and well designed while allowing to work with several people without risking conflicts with Git in particular because we seek to separate at most each element of code.



Screenshot explaining the functioning of an MVVM pattern, we can see that the Web Service, i.e. our backend, communicates with our frontend divided into three parts: the Data Access which manages our data, the Business Logic which manages our logic and our Presentation which is the graphic part displayed directly to our user.

So for example, if we want to display the list of movies in our database, we would first have a service file that will make the requests and retrieve the data according to different parameters (the number of pages for the pagination, the number of titles that we want to retrieve per page...) and where we have the management of our data, then we will have a component that will retrieve this data and perform the logic such as assigning each data to an element on the view side and finally the view which is an HTML page with CSS code that will display the different elements in the style of our application.

Thus, we have developed several different components:

- The t**itle** component which is loaded on the home page of our application, it displays a grid, that is to say a group of containers arranged in rows and columns, where we will find each title with its name, its poster, its average score out of 10 and its genre. Each element is clickable and the user can access the movie of interest by clicking on the movie poster. Moreover, a pagination system allows to display only the first X results when you make a query in the database, we will detail it below.

- The **titlePage** component which allows to display the information relative to a title, it is displayed when the user clicks on a movie in the title component. It displays the name of the title, its poster, its note, its creation date, its plot and other informations.

- The **search** component which manages the searches made by the user. It retrieves the information sent by the client in string form and then makes a request to retrieve the data related to the client's input. Finally, it will fill the title component with the title data.

- The **pagination** component which allows you to display only a defined number of titles while displaying the number of pages available (by default, twenty elements are displayed per page). Thus, when we retrieve our data from our web service, we will indicate the page we want and the number of elements we want, the backend then calculates the titles in the database to be displayed according to this information.

- The **loading** component which will display an animated gif on the whole page and hide the other components. It is launched each time the user ask for a component and is displayed as long as the component is not loaded. Once our component is ready in the background, the loading component is removed to make room for the other one.

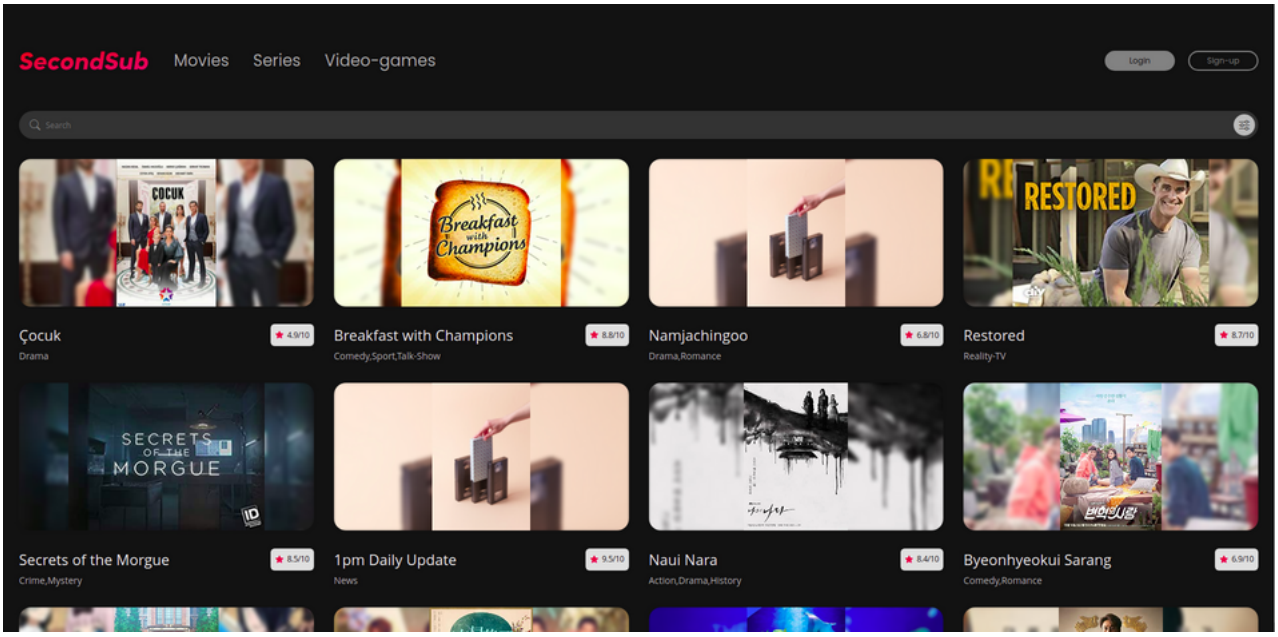
- The **bookmark** component which allows to bookmark a title. It is launched as soon as the user clicks on the bookmark a title button and will retrieve its ID. Finally, it sends to the database the title ID and the user username in a dedicated table.

- The **header** component which displays the header of the tool, we find the name of the site, important information such as the management of the user (the account component that we will detail soon) and the search bar.

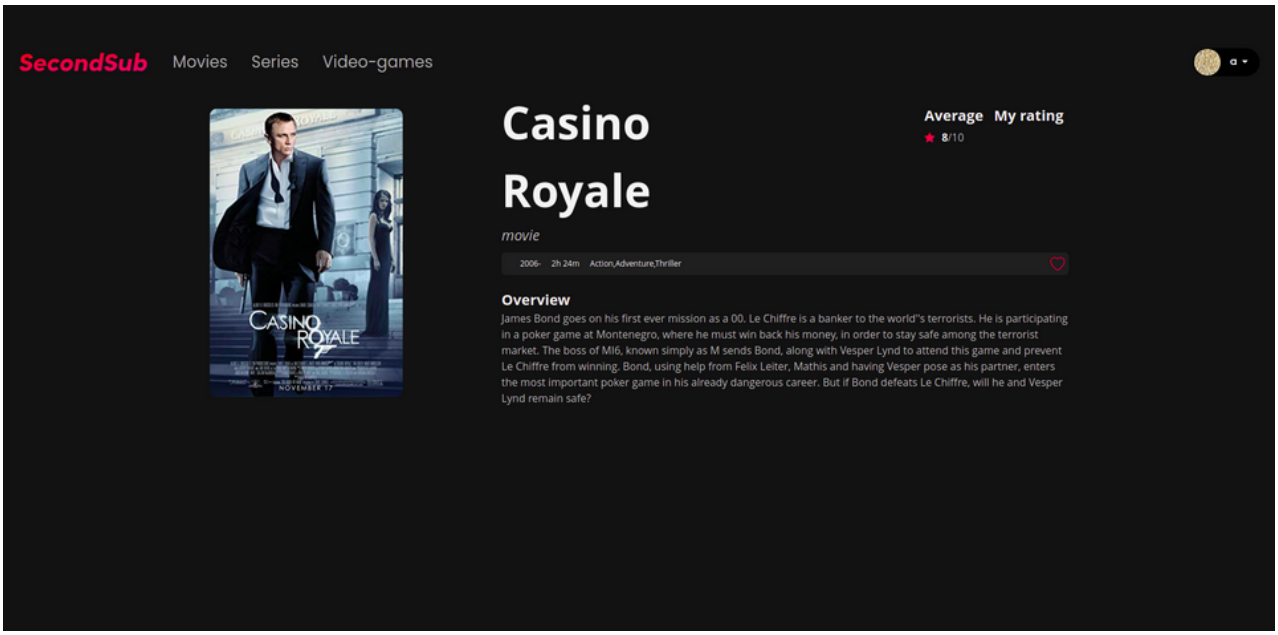
- Finally, the **account** component: this component, separated in three parts, will manage the authentication. Using Redux, an open-source state management library for web applications, we have three sub-components: the login which will check if the input sent by the user (username and password) is good and store a token to avoid reconnecting each time, the register which will allow a user to create his account and the info which will allow to display the user's information on the header. The interest of Redux is that it will allow to access this or that component depending on the state of our application, in other words, depending on whether the user is connected or not, Redux will choose which component to load and display: this allows us to just call the account component and then let the library manage it according to the action of the users.

Each of these components will be managed in our view model which will choose which component will be displayed according to the users' actions, in a way, it is the heart of our application.

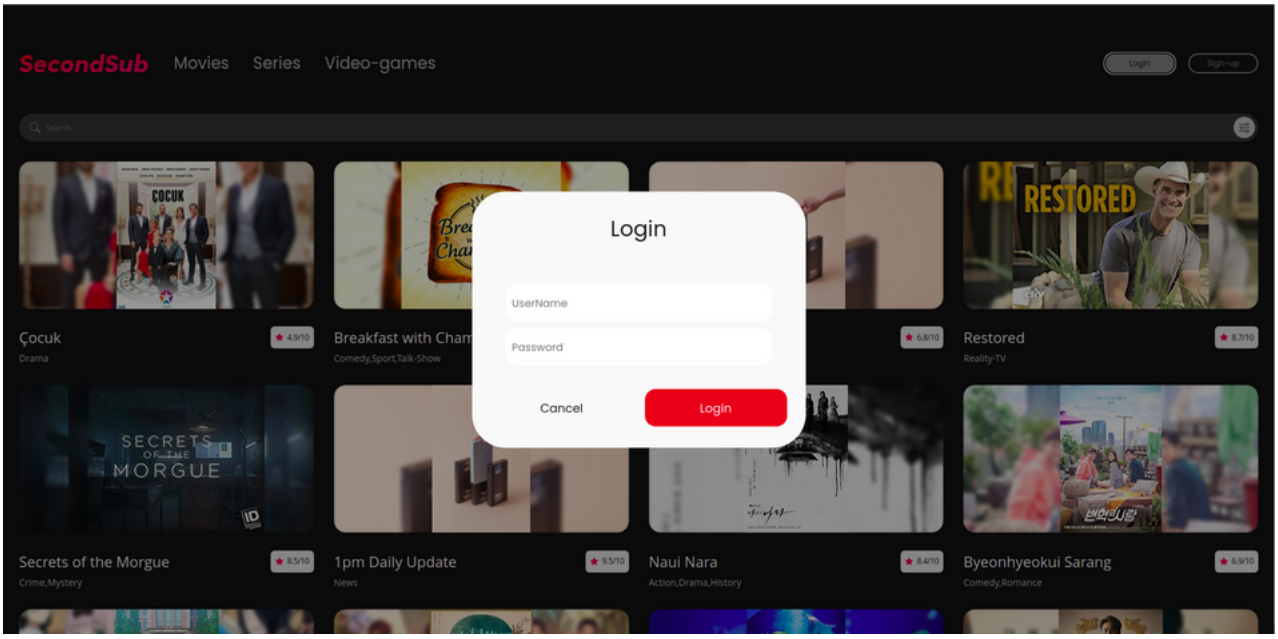
Finally, to manage the passage of our information when we change components, we continued to use Redux. Thanks to this library, we are able to store data in such a way as to avoid making as many requests as possible and to transform this data according to the pages so that it adapts to what the user wants to see.



Screenshot of the home page, we can see several titles with some information like the name of the title or its rating.



Screenshot with information from the movie "Casino Royale".



Screenshot with the login pop-up.

We also made some changes from the earlier portfolios, adding functions and an index, for some of the functions to run more efficiently.

Indexing

We decided to create an index on the word column in the wi table, as it was used in both the best-match and exact-match functions. We had earlier tried creating an index on the primarytitle, but it was used in a like statement (in some functions), wherefore an index would not be helpful as it has to search through everything to see if some part of the title is equal to the input.

```
create INDEX wiindex on movie.wi(word);
```

The running time of the function was decreased by 0,722s by creating the index, which took about 9s. This means that the time spent creating the index would be paid off after using the function about 12,5 times. As it is a function that will be used very often, we deemed this as being worth it.

Adding SQL functions to API.

We added some of the relevant SQL functions to the API, by creating services, domains and controllers that would provide paths for the functions to be used.

We split the services and controllers into 'search' and 'sitefunctions', where search is for functions used by the user and sitefunctions are for functions embedded into the website.

The services required domainobjects and for it to be added to the context. As the tables we map to are created in the SQL functions (and are not actual tables in the DB), we added the hasnokey to the entity type. We then mapped all of the function tables to the domainobjects in the API.

```
modelBuilder.Entity<SimpleSearchResult>().HasNoKey();
modelBuilder.Entity<SimpleSearchResult>().Property(x => x.TitleId).HasColumnName("id");
modelBuilder.Entity<SimpleSearchResult>().Property(x => x.PrimaryTitle).HasColumnName("title");
```

This allow is by using the fromSqlInterpolated

```
var result = _ctx.StructuredActorSearchResults.FromSqlInterpolated($"select * from structured_actors_search({str1}, {str2}, {str3}, {str4})");
```

objects from the context.

```
foreach (var searchResult in result)
{
    Console.WriteLine($"{searchResult.TitleId}, {searchResult.PrimaryTitle}");
    searchResultsStringSearches.Add(searchResult);
}
return searchResultsStringSearches;
```

And return that list, containing the search results.

CONCLUSION

This sub-project allowed us to discover or rediscover the different steps to build a robust and maintainable frontend thanks to the MVVM pattern. Thus, we were able to discover new tools like Figma and new frameworks like Knockout.JS for the MVVM and Redux for the data management between our components. We also had the opportunity to do some design, either on Figma when we were making our mockups or directly on the CSS, which took us out of our comfort zone.

So, unlike the other two sub-projects, we had a rather different approach on the front end.

First of all, because most of our members had already worked on similar projects (i.e. on dashboard or web interface development) but with different technologies: while one of our members knew React, another one knew Vue.JS. This particularity forced us to review our way of working, React and Vue.JS being rather used to make MVC (Model View Controller, a pattern where we have a model which contains the information to be displayed, a view with the graphical interface and a controller which contains the logic concerning the actions carried out by the user), we thus had to learn how to use correctly the MVVM pattern and Knockout.JS, a framework rather different from what we were used to use.

Then, unlike subprojects one and two, subproject three also required a designer's approach where we had to take more into account the user experience. As we explained earlier in the report, since we were developing the interface that would be in direct contact with users, we had to think through and justify every choice we made, because we believe that a tool can be as complete as possible, if it has a bad user experience, it will be shunned by the public.

Nevertheless, we enjoyed working on this sub-project and, more generally, on the portfolio project, we learned new and interesting technologies and methods and created a tool that we find satisfactory.

PART 2 - REFLECTIVE SYNOPSIS

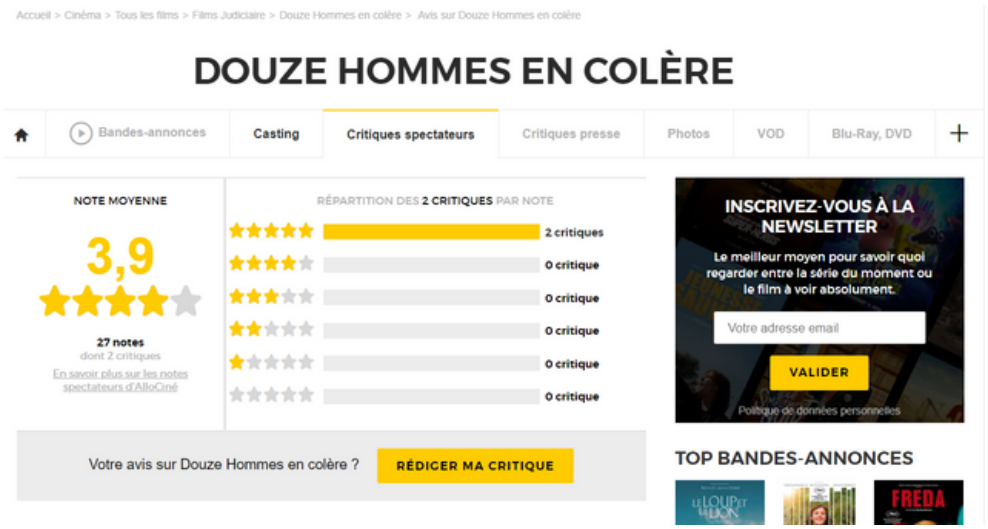
WHAT IS THE RAWDATA PORTFOLIO PROJECT?

The goal of the Portfolio RAWDATA project was to develop in four months (from September to December 2021) a tool to search, consult, rate and compare movies, TV series, video games or actors. This tool is inspired by the online database belonging to Amazon named *IMDB* which allows to restore a large amount of information on cinema, television or video games. It is in particular thanks to this tool that we were able to recover data sets allowing us to realize the project.

IMDB screenshot showing different information about the movie 12 Angry Men like the title, the poster or its rating.

Thus, in the space of four months, we had to develop this tool as a multi-user and responsive web-application, i.e. an application available on the Internet that must be able to manage several users by offering them functionalities such as the saving of their search history, a rating system or the possibility of marking titles as bookmarks for example. This project is divided into three sub-parts, each of which includes a part of the tool.

The particularity of our project is that we wanted to take inspiration from the French website called *Allociné*. This service was originally a database similar to ***IMDB*** but then decided to incorporate a social network system where users could rate titles, write reviews or rate the reviews of other users in order to catalog the most and least relevant. The social dimension of this tool where users become actors of the tool and can recommend titles to others appealed to us, so we quickly incorporated this idea into our project ideation process.



Screenshot of the website Allociné, it shows the page of the movie 12 Angry Mens (Douze Hommes en Colère in french) we can see a rating system similar to IMDB

User information box containing his profile picture, his nickname, his number of subscribers (here 3) as well as two buttons to see his other reviews and to follow him.

Screenshot of the website Allociné, it shows the comments section, we can see an user posting a review of the movie.

First, we worked on the database of the application during the first subproject. The idea was to create a database for the application while preparing key functionalities of our tool, especially those related to the user. Our database was composed of two data models, one to store title data (i.e. information about movies, TV or video games) that we collected from different sources (on the one hand datasets put online for free by *IMDB* and on the other hand thanks to *OMDB*, an open-source online database) and the other one to store user information and to run different components of our application like the movie rating, the bookmark system or the search history.

This database and its functionalities have been realized via **PostgreSQL**, a free and open-source relational database management system (**RDBMS**) that we learned to use for the RAWDATA project.

Then, we worked on part two of the project portfolio which consisted in the development of an interface between our database and what was going to be our web application. We had to develop a restful webservice that would allow us to manipulate the data in our database in a simple and secure way while proposing the most extendable and maintainable architecture possible. The idea was to think about how our web application would work in order to create the most logical endpoints possible and only retrieve the information we need.

For part two, we created our webservice with .NET 5.0, a framework developed by Microsoft and the C# language, an object-oriented programming language.

Finally, for the third part, we developed a frontend for our application. The idea was to offer to the customers, i.e. the users of our solution, a convenient tool to use that would group the different functionalities of our project. Thus, we had to develop a design that would provide a good user experience while extending our backend (i.e. our web service developed in part two) so that it would fit in with our frontend when needed. We chose the MVVM (Model-View-ViewModel) as our design model, which allows us to separate the view, i.e. what is displayed on the screen, from the logic (the functions that perform an action requested by the user) and the data access.

For the frontend, we chose to develop the application in HTML, CSS and JavaScript using the Knockout.JS framework, which implements our MVVM pattern as well as Bootstrap for the CSS which contains templates to develop web tools in a responsive and mobile-first way.

THE DIFFERENT TOPICS COVERED

During the development of this tool, we used a number of technologies or methods. Thus, we have chosen to cover, for each sub-part of the Portfolio project, one of these components in order to give its definition, where we used it, why and how it communicates with the rest of the application.

THE NOTION OF RELATIONAL MODEL

At the time of the elaboration of our database, we used what is called the relational model.

The principle is to model the existing relationships between several pieces of information (for example, the data found in our database). This modeling was originally proposed by E.F. Codd, a British computer scientist considered to be the inventor of the RDBMS model (or relational database, the type of database used for this project).

In our case, relational models represent how our data is stored in our database. Thus, a relational database is composed of several elements: First of all, we can find a Relation schema (in our case with PostgreSQL, a table) that represents the name of the relation with its attributes, that is to say the properties that will define a Relation. (The number of attributes in a Relation corresponds to the degree of the Relation. For example, if a Relation has five attributes, we say that it has a degree of five.

It is only called a relational schema when the schema has more than one relation.

In this relation, we will find rows that will be our data, each row is called tuple, the number of tuples in a relation is called cardinality. As for the degree of a relation, if a table has for example five rows of data, we will say that it has a cardinality of five.

Each attribute has what are called domain constraints. These constraints ensure that the attribute can only take on certain values that are available in the domain range. For example, if we have a constraint where an INT can only be positive, inserting a negative value will result in a failure or will be marked as NULL, a variable dedicated to representing unknown or unavailable values, which is represented by an empty space.

Finally, it is important that each relation in a database has at least one attribute that allows to define a tuple in a unique way. This tuple, called key, must be unique among all other tuples and cannot have NULL values. It allows to search data against conditions and is also used to create relations between different schemas.

To illustrate, here is a screenshot of one of our tables called TitleBasics.

titleid	titletype	primarytitle	originaltitle	isadult	startyear	endyear	runtimeminutes	genres
tt10850402	tvSeries	Çocuk	Çocuk	f	2019	2020	120	Drama
tt9055052	tvSeries	Breakfast with C	Breakfast with C	f	2017		(Null)	Comedy,Sport,Talk-Sh
tt8769260	tvSeries	Encounter	Namjachingoo	f	2018		60	Drama,Romance
tt8141256	tvSeries	Restored	Restored	f	2017		(Null)	Reality-TV
tt10066406	tvSeries	Secrets of the M	Secrets of the M	f	2018		42	Crime,Mystery
tt12511606	tvSeries	1pm Daily Upda	1pm Daily Upda	f	2020		(Null)	News
tt10850888	tvSeries	My Country: The	Naui Nara	f	2019		80	Action,Drama,History
tt7340800	tvSeries	Revolutionary L	Byeonhyeokui S	f	2017	2017	(Null)	Comedy,Romance

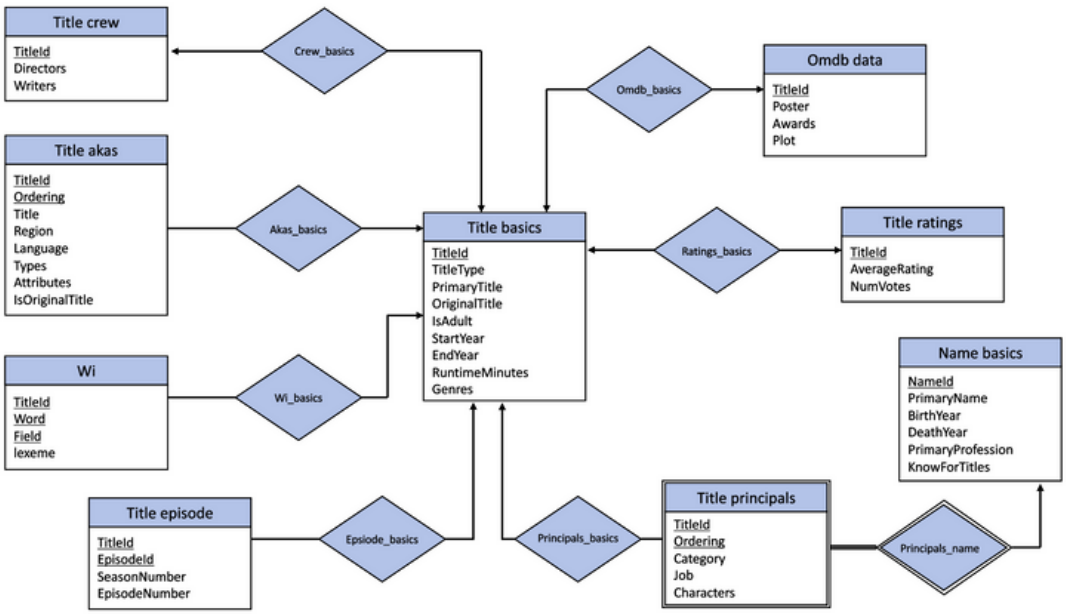
We can see nine attributes (titleid, titletype, primarytitle, originaltitle, isadult, startyear, endyear, runtimeminutes and genres), which means that our table has a cardinality of nine and is therefore the relationship scheme the TitleBasics(titleid, titletype, primarytitle, originaltitle, isadult, startyear, endyear, runtimeminutes, genres). There are also eight rows, so we have eight tuples.

Finally, our key here would be the titleid attribute because it is unique and cannot be null.

For our project, these concepts are important because they are the basis of relational and object databases (RDBMS) and therefore of PostgreSQL, so, in order to understand the functioning of this tool and therefore to elaborate our database, it was necessary to understand these concepts.

This was particularly useful when we were developing our E/R (Entity/Relationship) diagrams, which are models that allow us to graphically theorize the entities and their relationships between them in our database in order to see the organization of the latter. Thanks to these graphs, we were able to think about the functioning of our tool in order to optimize our data and their management.

Each attribute has what are called domain constraints. These constraints ensure that the attribute can only take on certain values that are available in the domain range. For example, if we have a constraint where an INT can only be positive, inserting a negative value will result in a failure or will be marked as NULL, a variable dedicated to representing unknown or unavailable values, which is represented by an empty space.



E/R diagram of our Movie model which allows to see how our tables communicate between them.

Thus, understanding these concepts is essential to achieve a tool like the RAWDATA project, because it allows to think and to pose the problem to have a functional database

ENTITY FRAMEWORK & THE API

- ASP.NET

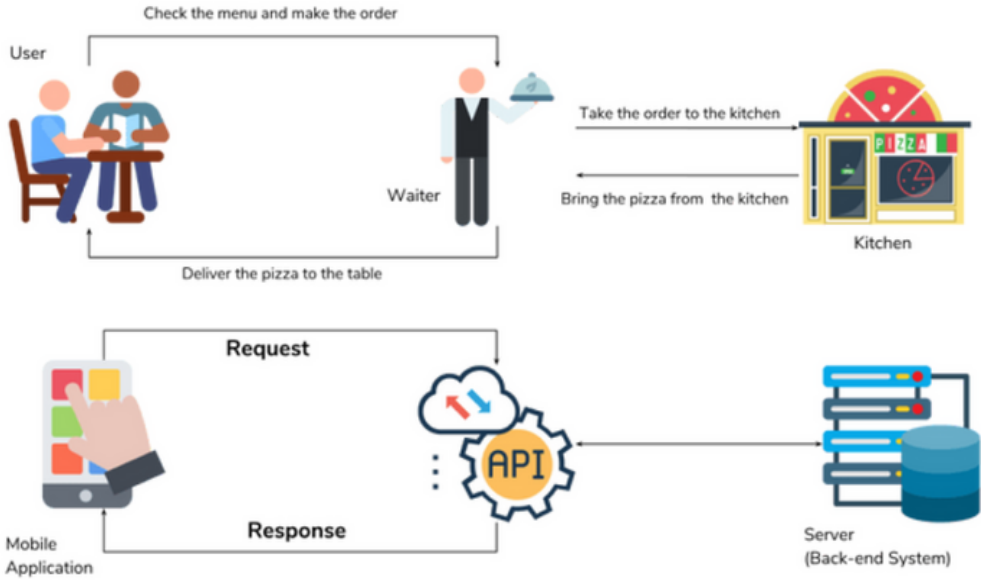
ASP is an extension to the .NET platform and provides tools for developing web apps.
It can be used in conjunction with C#, HTML, CSS & JavaScript to create dynamic websites.

- EF

Entity framework is an object relations mapping framework for .NET applications. It allows one to map relations in a database to objects in a .NET project.

- API Explanation

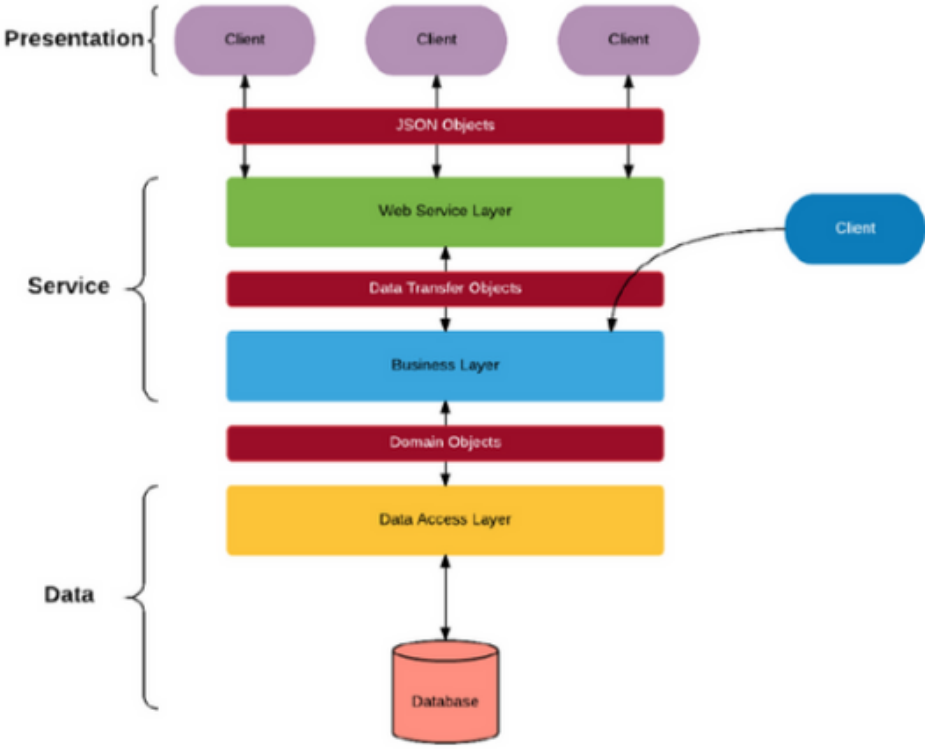
Suppose we wanted to order a pizza. A pizza with a stuffed crust, stuffed with cheese. Parmesan.
Suppose that we wanted some sauce on that very pizza. Tomato sauce of course. Maybe with some spices. Oregano, basil and so on. Paprika if you're feeling frisky. Sprinkle some pieces of chicken and cheese on top as well. No pineapple though. I hate pineapple on pizza.
Anyways, if we were to order this pizza in a restaurant, a waiter would take our order to the kitchen, and return with a pizza.



Replace order with request, pizza with response (although i'd prefer a pizza), the kitchen with the backend system and you've got an API. Just like that.

ENTITY FRAMEWORK & THE API

Our API



The Context (data access layer) in our project was used to map the relations to the domain objects we had created.
The business layers/services allow us to add functionality, by either calling SQL functions or manipulating the data in/from the database.
Finally the controllers (web service layer) provide paths for us to access the functionality in the business layer.

Connection to DB and Front End
EF Core allowed us to connect to the database, and map all the relations from the database or tables returned by functions in the database.
We implemented business layers that provided functionality and controllers (web service layer) that provided endpoints through which we could access the functionality.
This allowed the API to serve as a framework that connected the database, with the front-end, through functionality that allowed retrieval and manipulation of data from the database. Through calls in the front-end, we would be able to retrieve and display data or manipulate it through a user interface.

THE NOTION OF MVVM

For the front end, several technologies exist to create a dynamic and responsive page. Many frameworks integrate this idea in their operation, such as View JS or React JS, which mix HTML with Javascript (JS) and never touch HTML files. The frameworks will take care of loading the HTML data into the relevant files.

For this project, our goal is to be able to separate JS and HTML to have a distinctiveness of the dependencies on the different programming languages.

That's why we use knockout JS which allows us to have pure HTML and JS files.
It is the JS files that will be in charge of adding the data to the pages.

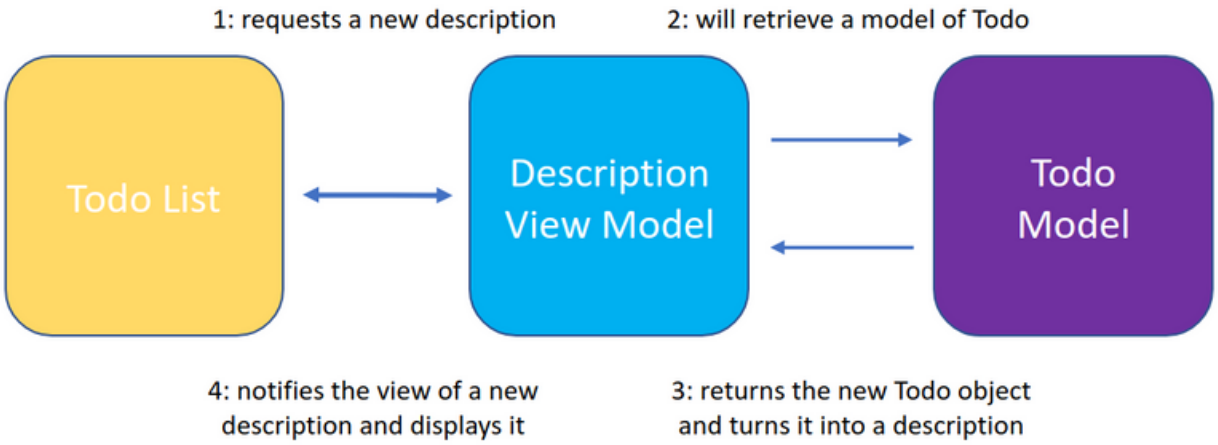
The architecture principle used is MVVM. But what does this mean?

MVVM is an acronym for model view view-model and corresponds to an application design that works like this: the view accesses the view-model, which interacts with the model.

We will now explain the 3 different parts:

- The **view** is the part that is displayed and on which the user interacts.
- The **model** is pure data.
- And the **view model** is a modified and adapted version of the model to have only useful information for the view.

To understand with a concrete example we can imagine a to-do list with the **view** which asks for a description to the **view model**, with the **view model** which retrieves the data of a Todo present in the **models**.



The **model** Todo is changed to a **view model** description that notifies the **view** of the modification for display.

To represent it in our project, we have 3 parts:

- Our HTML represents the **view**.
- The JS files that use knockout as our **view model**.
- And finally, the service JS files are the **models**.

The **view model** part is important because it is the heart of our application.

Indeed, it is in this part that we choose what to display of the **model**. This is also the section where all the logic is stored to allow specific actions to be taken.

A LITTLE DISCUSSION

During the different projects, we had to go through some steps and encountered some problems, so we wanted to detail some of them to explain how we experienced this project.

The first sub-project allowed us to discover or rediscover relational databases while becoming familiar with our group.
One of the particularities of this sub-project was to think about how to build our database in relation to the data files we had been given. Thus, we had to do a kind of reverse engineering in order to determine how we were going to build our different tables while creating the model to manage our users and the functionalities related to them.

Thus, a significant part of the sub-project was to draw E/R diagrams while modifying them as we went along so that they would better fit the idea we had of the final product.

The second particularity was that we worked with our working group on a real project for the first time. Because although we had two assignments before this project to learn how to use PostgreSQL, this was the first time we had to do a project where we didn't have to follow a predefined roadmap. Thus, we had to discover the strong and weak points of each member, their working method in order to find working methods allowing us to return a final product which is convenient for us all.

For the second sub-project, the main difficulty was to adapt our backend to our database. One of the elements that was going to become recurrent in our project was to have to come back to tasks that had already been done due to the fact that as we were developing, we sometimes had to modify our project, rethink our way of doing things or simply come across elements that were not or no longer compatible with our tool or with the technology we were using.

For example, in the first sub-project, we created an SQL function that, when used in the business layer, made an sql query by adding each element of an array (the search values from the user) individually, as we did not know in advance how many elements there were in the list

```
public List<ExactMatchSearchResult> ExactMatchSearch(int nbResult, params string[] words)
{
    Console.WriteLine("Exact Match");
    var query :string = "select * from exact_match('" + words[0] + "'";

    for (int i = 1; i < words.Length; i++)
    {
        query += ", '" + words[i] + "'";
    }
    query += ")";

    Console.WriteLine(query);

    var result = _ctx.ExactMatchSearchResults.FromSqlRaw(query).Take(nbResult); //SQL injection
```

Knowing the number of elements is required when calling fromSqlInterpolated, which is safer than fromSqlRaw for preventing SQL injections, as the input only gets interpreted as strings, and not potentially as part of the query.
To call it, we split a string by the commas in the controller, so we could create a single string in the url when using GET, to receive a result.

```
[HttpGet( template: "exactmatch")]
0 references
public IActionResult ExactMatchSearch(string searchKeys, int nbResult)
{

    string[] searchStrings = searchKeys.Split( separator: ',');
    var exactMatches :List<ExactMatchSearchResult> = _searchBusinessLayer.ExactMatchSearch(nbResult, searchStrings);
    return Ok(exactMatches);
}
```

To combat this, we decided to split the string in the SQL function instead, thus simplifying the controller and making it safer.

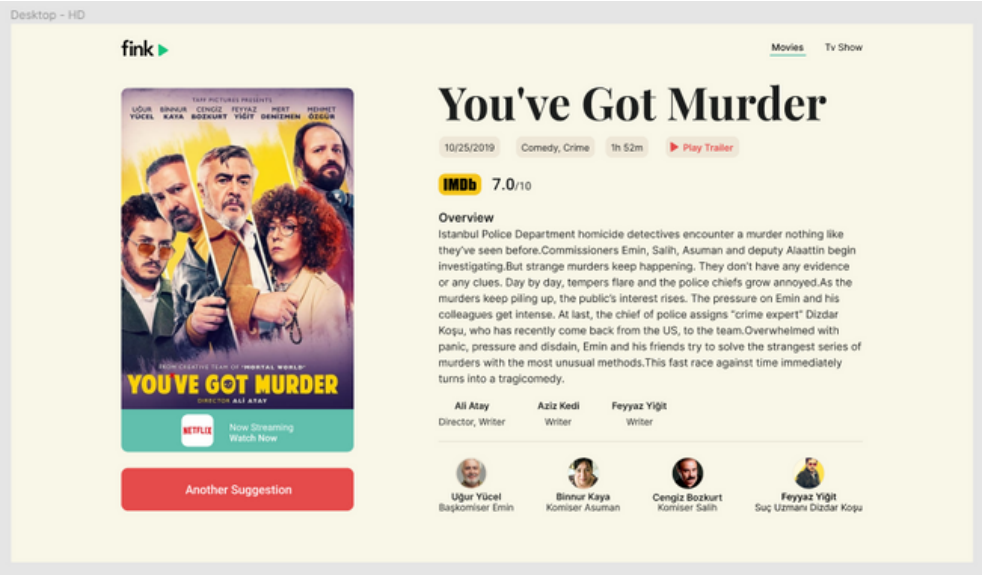
```
w = string_to_array(searchstring, ',');
```

This could be done with a single call in the SQL function (exact-match & best-match), where we pass a string in which search input is separated by commas, instead of being an array. The string_to_array call then creates an array with the search input values.

Finally, for the third part of the project, the main difficulty was the development of the graphic side and the consideration of the user experience. Indeed, one of the expectations of the project was to create a tool that would be useful and pleasant for the user, so we had to put ourselves in the place of the user when we developed a functionality.

For example, when we wanted to create the home page, we had to think about the layout of our information, how users would function and react to what they see on the screen and thus anticipate their actions.

To do this, we scoured the Community branch of the Figma website which offers interfaces and mock-ups for a plethora of projects. These mock-ups, most often created by UX designers or seasoned front-end developers, are accompanied by recommendations or explanations as to their choice during the development of these tools. We were therefore inspired by these works to create our tool.



Screenshot of Figma where we see a template for a site displaying movies, we used as inspiration for our project.

TO CONCLUDE

In conclusion, the Portfolio project allowed us to discover new methods and new concepts while working in a way halfway between what we find in a company and in a university.

First of all, we were able to discover or rediscover new languages and frameworks, especially C# which had already been used by members of the group for software development as well as ASP.NET frameworks and especially Knockout.JS which forced us to work on JavaScript in another way than what we were used to do.

This project also forced us to think more about how we want to create the tool, notably by making us create E/R diagrams and UMLs, but also by creating mock-ups of our final product on software such as Figma (a prototyping tool available online).

Finally, for EPITECH students, this project was different from what we were used to do in our school because it required a theoretical part. Indeed, we used to focus on the learning of languages and the technical side even if it meant leaving out the ideation and reflection phases. Our projects looked more like a checklist of things to do than a tool to shape with our point of view, the realization of the Portfolio has allowed us to review our approach and our way of working

Conversely, for the RUC student, it has been nice to learn from a more practical approach, especially in a project/course as far reaching and with as big of a workload as this. Learning by doing, and doing a lot.
With a cross-disciplinary bachelor, working with more specialized students has also shown me the specifics of my education, where my strengths lie and where i want to develop.