Project Report SEP 4

Table of Contents

[1. Abstract 3](#_Toc522637843)

[2. Introduction 3](#_Toc522637844)

[3. Analysis 4](#_Toc522637845)

[4. Design 4](#_Toc522637846)

[4.1. Choice of technologies 4](#_Toc522637847)

[4.1.1. Back End 5](#_Toc522637848)

[4.1.1.1. Programming language: C# 5](#_Toc522637849)

[4.1.1.2. .NET Core framework 2.1. 5](#_Toc522637850)

[4.1.1.3. Entity Framework 6](#_Toc522637851)

[4.1.2. Front End 6](#_Toc522637852)

[4.1.2.1. Angular 6 7](#_Toc522637853)

[4.1.2.2. PrimeNG 7](#_Toc522637854)

[4.1.2.3. Other packages and libraries 7](#_Toc522637855)

[4.2. Code structure 8](#_Toc522637856)

[4.2.1. Back end code structure 8](#_Toc522637857)

[4.2.1.1. Controllers Folder 8](#_Toc522637858)

[4.2.1.2. Models folder 9](#_Toc522637859)

[4.2.1.3. Startup.cs 9](#_Toc522637860)

[4.2.2. Front end code structure 10](#_Toc522637861)

[4.2.2.1. app folder 10](#_Toc522637862)

[4.2.2.2. \_guard and \_helper folder 10](#_Toc522637863)

[4.2.2.3. \_models folder \* 11](#_Toc522637864)

[4.2.2.4. \_services folder 11](#_Toc522637865)

[4.2.2.5. Component folders \* 11](#_Toc522637866)

[4.3. Database 11](#_Toc522637867)

[4.4. Client – Server connection 12](#_Toc522637868)

[4.5. GUI 13](#_Toc522637869)

[5. Implementation 16](#_Toc522637870)

[5.1. Front end Registration 16](#_Toc522637871)

[5.2. Back end registration 20](#_Toc522637872)

# Abstract

Cinema Web Page is a web page, that enables user to look through movies, which will be played and after registration in our system, the user is then able to book a seat for a specific movie and book parking if he wishes to. User will have overview of his bookings with aim for managing these reservations. Our web page also supports role of administrator. Administrators are able to create a movie and has an overview of all users which are currently in the database and also is capable of alternating of the users and deleting them also he is capable of overviewing all parking reservations and delete user reservation of a parking place.

# Introduction

In the current age it is hard for small cinemas to compete with huge multiplex cinemas. Modern cinemas must have web page to compete because majority of people are searching on the internet for free time activities. Our Cinema Web Page is the solution for this issue. It allows our registered users to book a movie and parking, so they can just come to the cinema and enjoy they free time without any worry for lack of parking places and sold out movie projections.

# Analysis

# Requirements

# Functional requirements

Functional requirements should reflect how our web page will work from the user perspective. These requirements are divided to four categories which represents their priority for development.

**Must have:**

* Database of users, movies and parking
* Registration of users
* Login for users
* Users can book a movie with seat

**Should have:**

* Two roles Administrator and user
* Calculation of price for tickets
* Administrator can create a movie
* Administrator can change or delete a movie
* Administrator can change or delete users
* Users can book a parking place

**Could have:**

* User can see their current and past bookings
* Administrator can see history of all data
* Administrator can see free and occupied parking spaces
* Administrator can remove user from a parking place
* JWT authentication

**Nice to have:**

* PDF generation
* Gateway for payments

# Non-functional requirements

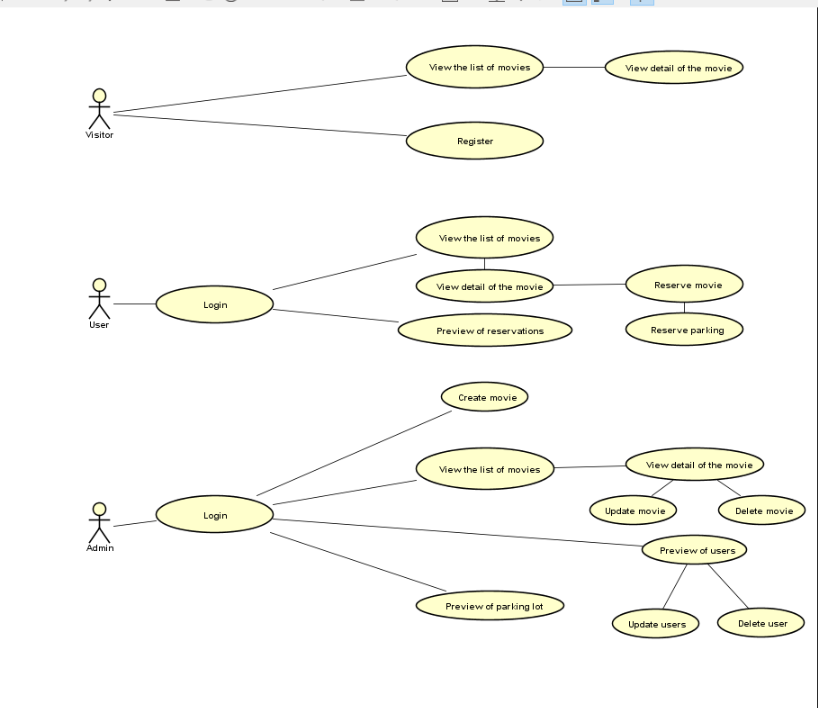
These requirements should reflect technological requirements for our web page.

* Back end written in C#
* .NET core framework
* Entity framework
* Angular 6 on front end
* Compatible with Google Chrome version 68
* RESTful web services

# Use case

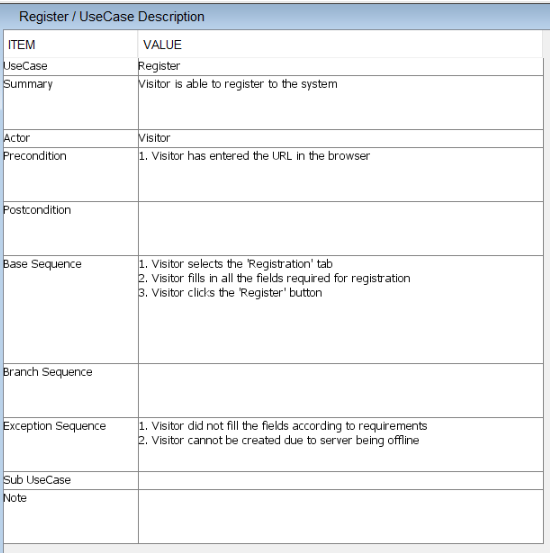
Before the project start we made an internal analysis and created use cases. These use cases are created in regard of our goal which we presented in the Introduction. These use cases reflect suspected behaviours of our three types of actors.

* Visitor: is a non-registered user which has limited possibilities. Visitor is only available to observe movies which are played and register to become a user.
* User: is a registered person in our database. This user can log in and observe played movies, but he can also reserve seats in specific movie and reserve parking for himself. User is also available to see all his reservations of parking and movies.
* Administrator: is special type of User. Administrators are not created through registration but are manually inputted into the database. Administrator upon logging in can see all the users and is able to manipulate with them (change user or remove him), he can also create movie as well as modify it or remove it. Lastly, he can see all parking places and remove user from reserved parking place and make it available.

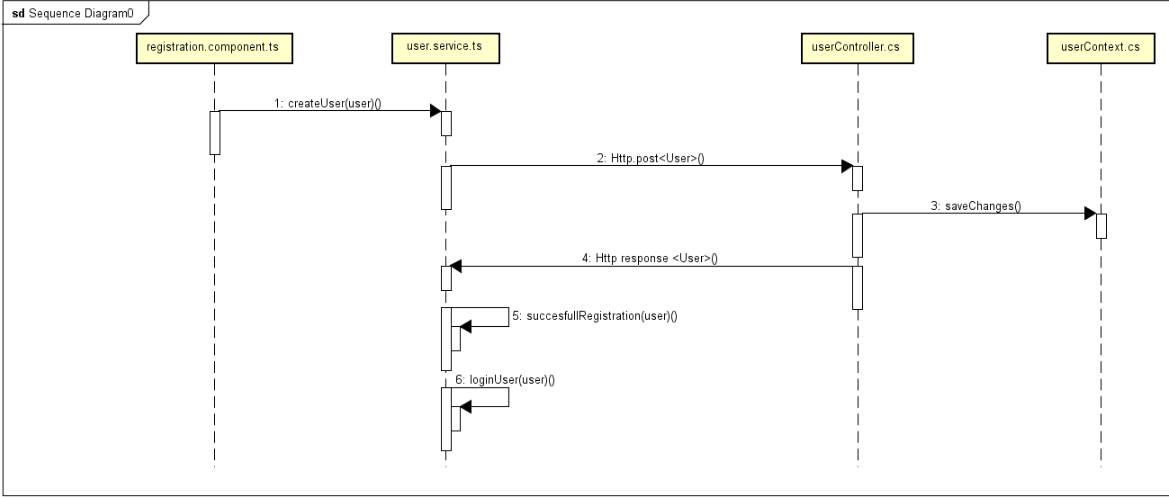


As an example, we will use register user use case.

# Registration use case detail



# Sequence diagram for Registration



# Activity diagram for Registration

# Design

# Choice of technologies

In Cinema Web Page we were trying to use the best available technologies for web development. Bellow we will list all technologies we used with reasons why we choose them over other possibilities. This list will be divided into two subcategories which are: Front end and Back end.

# Back End

In regard of back end programming language for a web page we could choose between function programming languages and object-oriented programming languages this choice will then affect other frameworks and databases we use:

* Functional programming languages: Are very powerful for web page application. They are lightweight and really fast but sometimes they have issues with data consistency and requires multiple frameworks to operate accordingly. In this field we were mainly considering using Node.js but for a such a short period of time it would be really challenging to set up this back end correctly.
* Object-oriented programming languages: These languages are on the other hand heavier and in a lot of ways slower then properly set Node.js back end, but they support a lot of good frameworks, have great documentation and mainly they are easy to set up and are more conventional approach towards web page development.

After consideration we decide to use Object-oriented programming language. Here we could choose from Java or C#. In terms of efficiency they are similar, therefore decision was made upon “friendliness” of these system and what frameworks they support. With this in mind we choose to go with C# and .NET core framework.

# Programming language: C#

As said above we choose this programming language with regards of powerful frameworks we can use with C#, good documentation and better user friendliness of this language.

# .NET Core framework 2.1.

.NET Core framework 2.1. is a great framework for working with RESTful web services. It contains several methods to handle request/response calls.

# Entity Framework

Because of the time limitation we had to choose database which is easy to set up and easy to maintain. Entity framework is perfect for this job because it allows us to create just models of database and Entity framework will create the tables and relations for us. Also, when we need to change some table we can just change the model and it will do the rest of the work for us. Another big reason is built in methods for queries which will make our work database much easier.

Also, Entity Framework support multiple types of database from proper persistent SQL databases to in memory database which is great for development. In our project we used the In Memory Database because we anticipated that database will be changed quite frequently during development and therefore it’s fine for us when database will be recreated every time we start the server.

If you would wish to run database in proper SQL server in *Startup.cs* there are comments to help you switch between these two.

# Front End

After we decided on Back end we had to decide how we are going to build our front end. There are even more options then on the front end. Three main ways we could go is create front end with ASP.NET, use one of many JavaScript frameworks or combine these two approaches.

* with ASP.NET: This way would probably the most straight forward because we had already ASP.NET back end API. But there is a serious down site for this approach. First of all, it’s not widely used for this purpose and it is slow compared to JS frameworks.
* JavaScript Frameworks: They are really fast and with some knowledge of functional programming very user friendly and with lots of packages which can be easily imported.
* combination of two: created with ASP.NET templates for JavaScript frameworks (Angular, React). This combination is perfect for fast creation of web pages, but it is really challenging to make changes of the template.

We decided to work with JavaScript frameworks here we choose Angular 6 (latest at the time) over React. We choose Angular over React even though it’s DOM is great for rendering pages and show the changes without necessity to reload page but to do that we would need to establish Redux for passing along data between components and it would take too much time.

At first, we tried to use ASP.NET with template for Angular but there is only template for Angular 5. We tried to update this template version to Angular 6, but it didn’t work for us. Therefore, we moved to stand alone Angular 6 front end.

# Angular 6

Angular is the most used JavaScript framework for front end. It has a lot of functionalities integrated into it where he also gets an edge over React with Redux. It also contains TypeScript which enables us to write JavaScript code with classes and interfaces, this is important for us because we must learn how to write in Angular 6 on the way.

# PrimeNG

PrimeNG is a library for user interface, we are using it to have unified UI and it also saves us a lot of time because we can use components such as calendar, inputs, tables and others.

# Other packages and libraries

All packages and libraries can be found in *Package.json* located in front end folder.

# Code structure

We tried to make the code structure as straightforward as possible.

# Back end code structure

Our project on the back end is WEB.API project template from ASP.NET. Here except config files and nugget packages you can find folders Controllers and Models as well as file Startup.cs. These are the most important for our project.

# Controllers Folder

Inside this folder there are WEB API controllers, each controller represents part of logic and can be accessed by calling “/api/[controller]”. The API endpoints name is name of the controller without the “controller” part. Therefore, endpoint for MovieController will be “/api/movie”.

Each controller contains methods for HTTP requests relevant to its purpose. Within these controllers we also processing data and storing them to the database which is not the best practice. Reason why we did it this way is the short period for this project, so it was convenient for us to have everything at one place.

Special place between controllers belongs to MovieController. Because we are using in memory database we need to create database every time we are running the program. Once you start the server it will open browser with URL to API call to movie (.../api/movie). This action will call the constructor of this controller where we call private function fillDatabase(), this function then set the context of the database.

# Models folder

Models are folder where we are storing our models. In this folder there are three types of files.

* Database models: These models are used for generating the Entity Framework database tables. These models are Movie, MovieSeat, User and ParkingPlace. Within these models there are special fields with annotation [NotMapped] which tells Entity Framework not to put them into the database.
* Database context file: This file is called CinemaContext.cs and it is responsible for creating tables from database models with the Entity framework.
* Data transfer models: The rest of the models are specific models for handling HTTP requests from front end. They were created because HTTP methods can send only one object.

# Startup.cs

In this file we are adding services which are used in our project.

* Setting database: by calling AddDbContext on CinemaContext.cs. Also, there is a commented service for switching to SQLServer.
* Enabling CORS: for every call from localhost:4200, for every method, header and credentials. This is required for running this application on localhost.
* Serializer settings: for our database, to help Entity Framework handle table reference in the database.
* JWT token authentication: In early stage of development we wanted to use JWT authentication. If you would like to use it uncomment this code, it was tested on back end and it’s working,

# Front end code structure

Front end is bit more complicated regarding structure. First, we look into configuration elements.

* Angular.json: is a configuration file for a whole project.
* Package.json: it contains all dependencies required for running an application, also there are scripts which can be used from the terminal. We modified script start (npm start) to use a proxy file.
* Proxyconfig.json: here we are setting up simple proxy to helps us avoid problems with CORS.

Then most important part is in app folder where is the actual application.

# app folder

App folder contains the top level of our application as well as several folders which we will look into later.

* App.module.ts: here we instantiate all components, importing third party modules and setting up providers so they are accessible through the whole application.
* App-routing.module.ts: here we are setting up routes to each component so we can then navigate to them through router.
* App.component.\*: is the top component we have there first level of navigation and getting user information from localStorage if it has any.

# \_guard and \_helper folder

In general, these folders contained TypeScript files which were created to handle JWT authentication. But because we didn’t had time to finish implementation of this in the front end, they are never called. We are keeping these files inside to show the way how they meant to be implemented.

Only exception is handler.ts in \_helper folder. We are using this for handling errors from HTTP response.

# \_models folder \*

In this folder we are keeping our models. These models are TypeScript equivalent of back end models.

# \_services folder

Here we are storing our services. We have three types of services in our application:

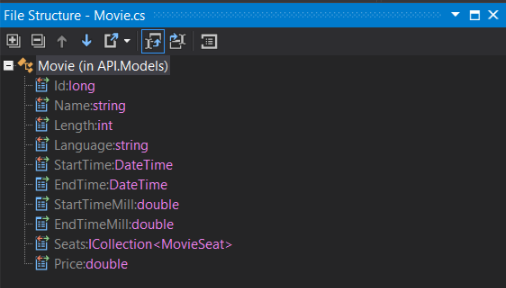
* Services handling API calls: these services are equivalent of back end controllers. Sending requests to the server and getting the responses.
* Toast service: for calling toast in the component upon success or fail.
* Data service: this service store data we are getting from back end. These data are immutable therefore it can be only set or get.

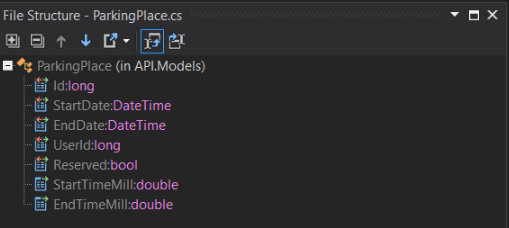
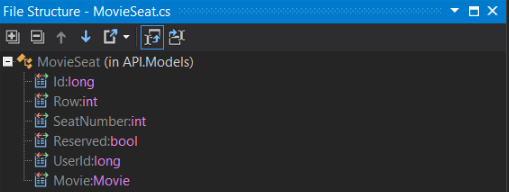
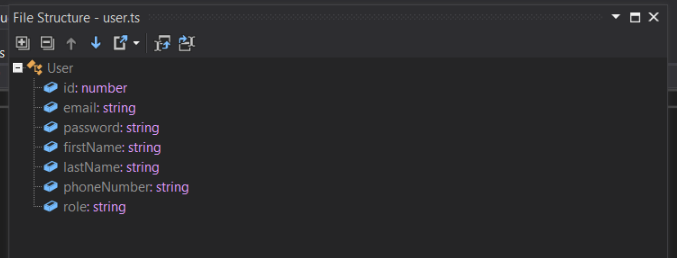
# Component folders \*

The rest of the folders are component folders. They include html, CSS and TypeScript files for each component.

# Database

As said above we are using Entity Framework to generate database tables for us from the code. We have four tables Movies, MovieSeats, Users and Parking. Movies, MovieSeat and User are connected between each other as will be seen from the class diagrams bellow. For purpose of getting correct data we had to implement serialized options, because without it we had loop reference.



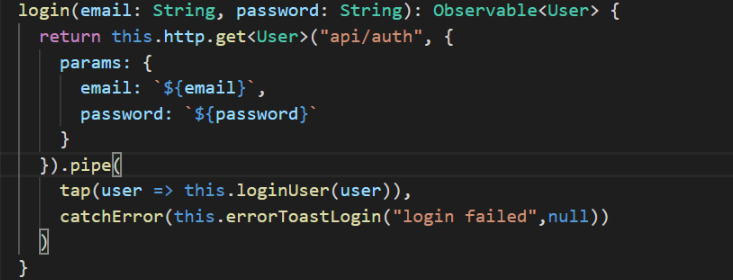


# Client – Server connection

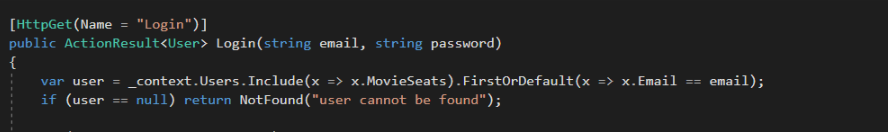
For connection between server and client we are using RESTful web services. This is handled by .NET core 2.1. on the back end and http client module with help of observable module on the front end.

The server API endpoints are made by the controller so every controller on our server has its own endpoint. The HTTP methods are handled by .NET core 2.1.

Client creating HTTP method request by calling http client module which will then fill all necessity of the HTTP method. At login function we alternate this approach because we are sending two strings in the GET method, so we must pass them by parameters instead of sending them as an object.



And then at the server we are getting these two strings from the parameters.

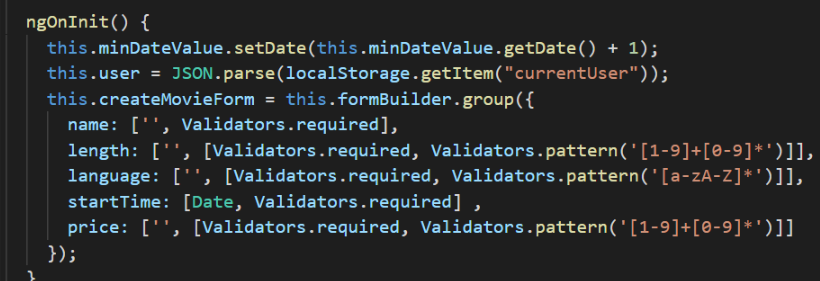
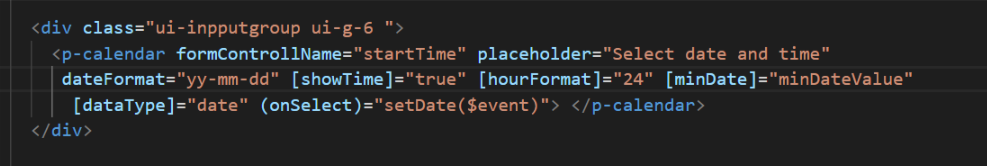
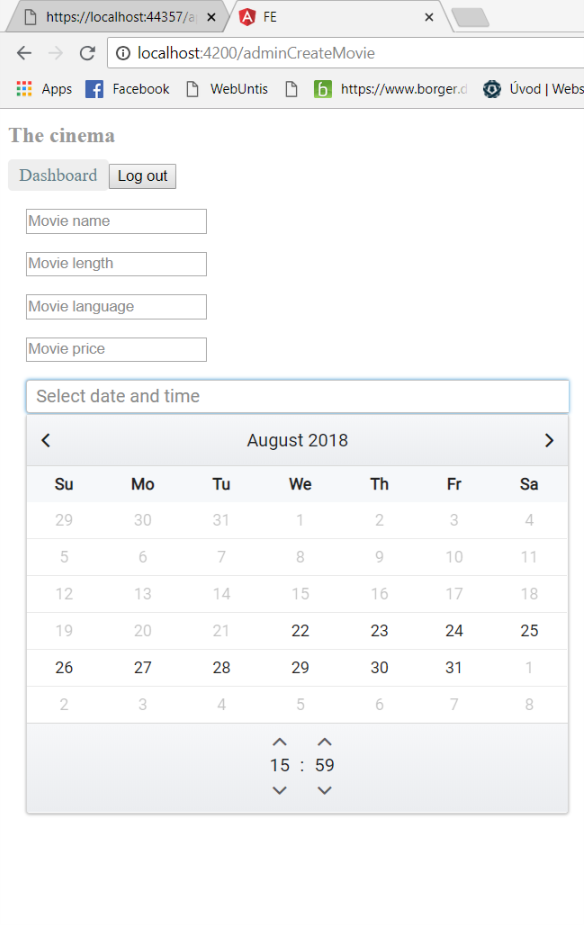


As a protocol we are using ISS express protocol provided by Visual Studio.

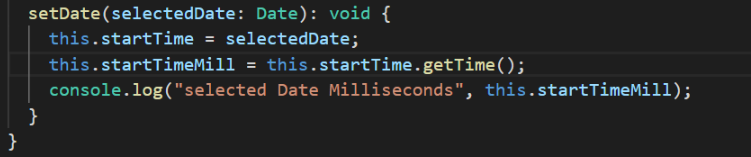
# GUI

For a GUI we used HTML with CSS. Because it is a web page we didn’t have much choice over this approach. Our GUI is simple because creating proper UI takes a lot of time and we were more focused on the functionality.

In the UI we were using PrimeNG which is as stated above a library with its own components. As an example, we used p-calendar component for calendar component with time picker as well and then we put it into the validation form which is provided by angular.



But because we had problem of getting the selected date to the form directly we had to call onSelect attribute and pass the event (date in this case) to setDate method which assign it to the ng model and from there we set it in onSubmit method witch is called after clicking on the button Save show on the web page.

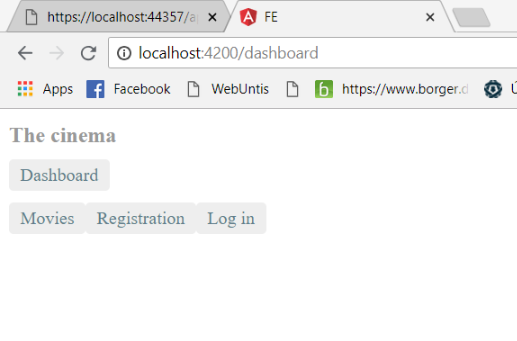


# Implementation

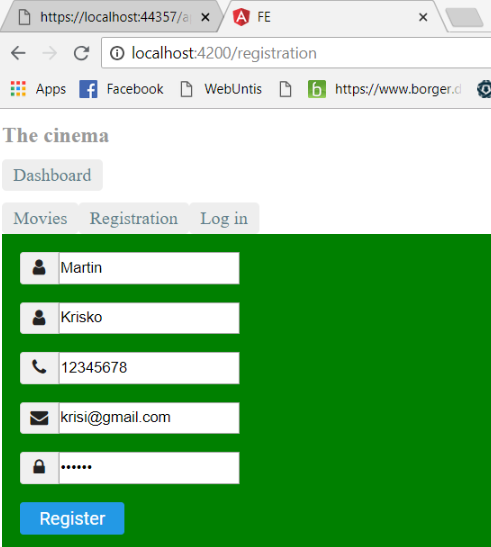
For implementation we choose to show registration of a new user which is then followed by a login. We choose to present this function because it’s touching several services, it’s crucial for our web page to work and include HTTP request and handles the response.

# Front end Registration

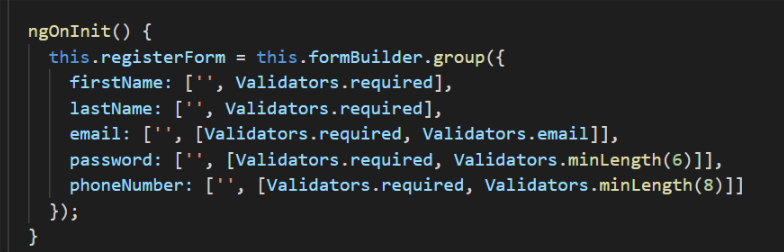
If visitor (defined in the analysis section) visits our web page he will be see a dashboard with options to see movies, log in or register. In this case visitor wants to register himself.



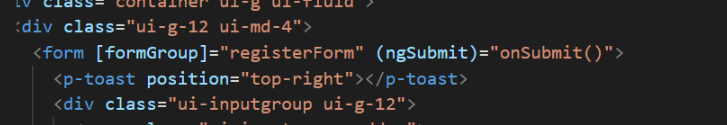
When visitor selects tab for Registration, form will be showed which he had to fill.



This form has validations for an empty field on every input. Further the email input field has validation on email format, phone number must be longer than eight digits and password must be longer than six characters.

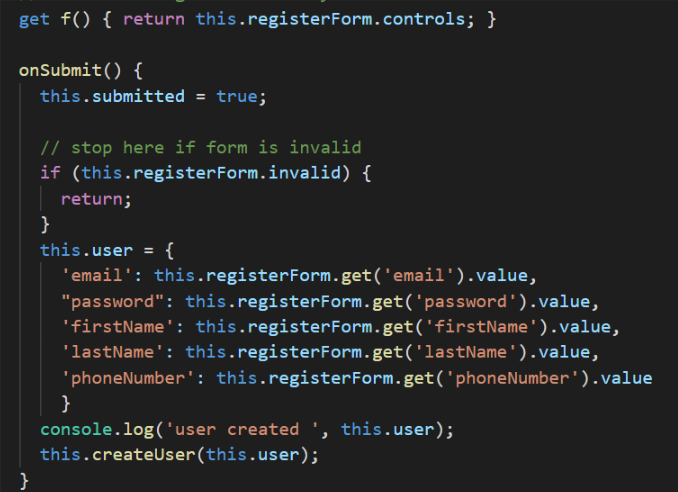


Then upon clicking the Register button, form property ngSubmit will be triggered and call function onSubmit().

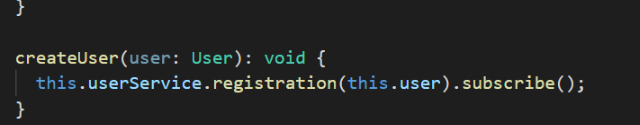


Before we move to the onSubmit method. You can see that we are using our helper function f(). Reason why we are using it is to be able to see if registrationForm is valid on the frontend.

Back to the onSubmit. Here we are checking if register form is valid and if this is true then we assigning to ng model user values from the registration form.



After the ng model user is populated we are calling createUser method with the parameter of user.

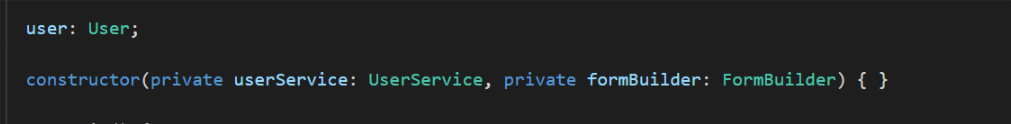


This will call registration method inside the userService with the parameter of newly populated user. Also, the function subscribe is important here because it works as an asynchronous call because userService then calls the server API.

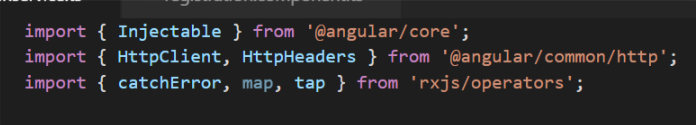
The userService service is important into this component with:

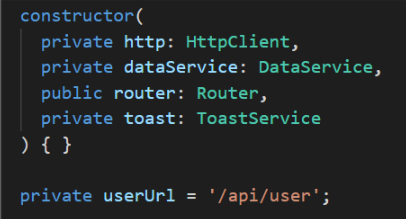


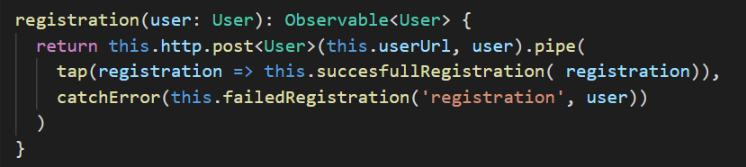
And then it must be added into the constructor through dependency injection:



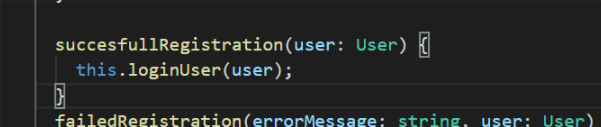
After we call the userService it will call the registration method which then calls the server with api/user. For the call we are using HTTP client module provided by Angular. In a pictures bellow will be importing of the client, adding it to the constructor and the method for registration.



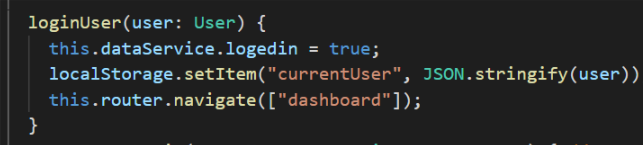




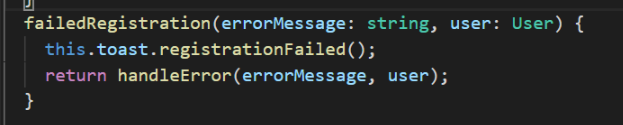
The pipe method in the end of the HTTP POST call allows us to manipulate with the response from the server. If the request was successful, the tap method is triggered here we are calling successfullRegistration method with the object from the back end (will be shown in a back end implementation of the registration).



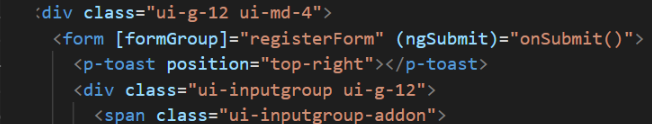
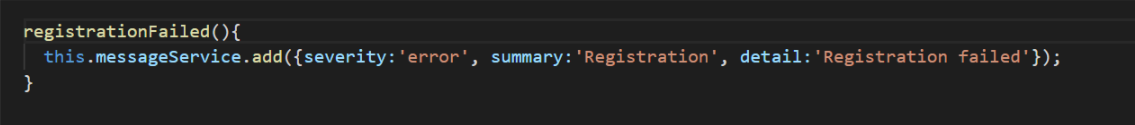
This method then calls loginUser with the object from back end. Here the user is logged in, saved to the localStorage and then navigate back to the dashboard.



If the registration fails it will trigger catchError and this call function failedRegistration with the name of the method where it fails and the response object.

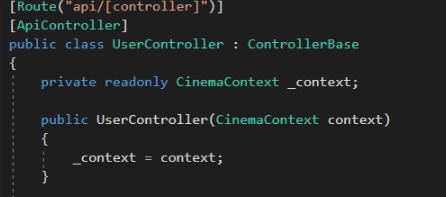


In failedRegistration we then call toastService which will create toast on the component p-toast in our HTML and then send the message and object to our helper service handle.ts where method handleError is called with these parameters.

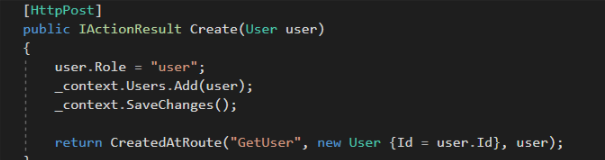


# Back end registration

After the clients calls the server on the /api/user it will get the database context by using dependency injection.



Now it will recognize that the request is for HTTP POST, so .NET core framework will then call method with this annotation, in our case function Create.



Here we will get the user from the client and because the models for user in client and user in server are identical we will get the correct object. Then we assign him a role User (Admins are created by directly inputting them into the database). Then it will call the context table of Users and add the user, after this the database will save the changes.

Then it will return method CreatedAtRoute which is .NET core method for handling post methods. It will return the newly created user with assigned Id from the database with response code 201.

# How to run it

In this part we will give you some basic guidelines how to run the program and where to change configuration if you have problem with running it.

# Back end

Back end is written in C# with .NET frameworks therefore we highly recommend using Visual Studio 2017 for running the server. Before you open the back end make sure that you have installed .NET Core framework 2.1. it can be downloaded . Our back end is wrapped inside solution, so you can just open the solution. Nugget packages should be downloaded once the solution is opened. If they are misbehaving please look into API.csproj to find your missing dependencies.

# Result

The system was created in regards of the requirements we set in the beginning of this project. During the development we start from the non-functional requirements and set the technologies accordingly and then we continue functional requirements. Here we started with must have category and after fulfilling all requirements from this category we moved down in the hierarchy list.

From must have category we fulfilled all the requirements. Therefore, we have set up one database with four tables, we had to add table of movie seats, so our movie table works as supposed to. Visitors can register themselves, which will store them into the database and therefore they are able to login afterwards. Once user is logged in he can then book multiple seats for a movie.

Once all must have requirements were fulfilled we moved to should have category. We implemented Administrator role into our web page. Give them their own dashboard and specialized set of functions for deleting and changing the data with exception of changing movie, also Administrator can create movies on the web page. For a user we are showing total price of the tickets after he select seats in the movie and give him possibility to book parking place for a movie.

From the could have category we implemented possibility for Administrator to see free and occupied places in the parking lot as well as removing user from the parking place.

# Discussion

Our web page fulfils most of the requirements we set up in the beginning but there are some exceptions. This project is not perfect and there is a lot of things which can be improved. Due to lack of time we had to abandon most of requirements from could have and nice to have category as well as simplified several operations.

The most visible is shortage of UI elements inside our web page. If we had more time we could beautify the UI quite significantly because the logic is already in place, but we just didn’t have time to play around with CSS and HTML elements. This part of a web development is important especially for a release version but during our development we decided to prioritize functionality over UI design.

From the should have category we weren’t able to implement changing of the movie for Administrators. The back-end functionality is in place as well as logic in the services to hand this requirement. But because of time shortage we had no time to implement separate component for changing a movie, also we figure out how to do the detail pages properly in later stage of development and we decided to go with user detail for an Administrator and movie detail for a User as a priority.

JWT authentication supposed to be one of our key features in the application, reason why we placed it into the could have category was that it is really time consuming to implement, hard to debug and once implemented it would require login to see changes on frontend and for getting data from server. We worked on it for two days and correctly implement it on the back end, on front end we created all necessary files to support JWT authentication and they were tested, but if we wanted to send the authentication request, pre-flight request from the http method was not successful. Even after we put several hours into the debugging we were not able to make it work. To implement this, we would need more time or help from some senior developer.

Regarding the possibility for User to see all his bookings we implemented the logic on the server side and even on the front end. But because we found bugs in the higher priority requirements we had to abandon fully integration of this feature to the web page. To make this requirement work we would need just one day more because all what is left is propagation this data to the HTML and few functions in the front end.

Another unfulfilled requirement is that Administrator can see history of all data. This wasn’t implemented because it would require new history log database, complicated database queries, creating of new controller and whole logic and UI on the front end. We just haven’t had enough time to make it work. In order to implement this, we would need few days of work dedicated to this.

The nice to have category was not implemented at all. They represent nice add-ons to our web application which would push it closer to real app that could be widely used in production. We knew that it would require small miracle to finish this requirements due the time shortage. For implementing this we would need much more time.

Regarding the code structure of a back end. We know that it is quite messy at this time. It is not a good practice to have all the logic with database queries inside of the WEB API controllers. We should have different project which would contain the logic under one solution. But again, time pressure forced us to have it in one place because it was convenient for development. Separation of logic, controllers and database queries would not take so much time, but it would require more time than we had.

Another issue with our project are toasts which would appear as a fail or success action in the front end. Sometimes they show fail error even though the request was successful. Reason for this misbehaviour is our bad implementation of response from the server. HTTP methods requires specific type of response and if you not provide one it will go to catchError phase even though it triggered tap phase in the API calls on the front end. To fix this misbehaviour we would need to adjust server response sending and implement http errorInterceptor on the front end. Error interceptor is created but because it was meant to be implemented with JWT authentication we didn’t implement it in our working code.

In case of scaling this application lots of work need to be done mostly on the back-end side of the project. As of now it supports only one movie projection at the time and it automatically assign parking place if requested to. Reason why went only for one movie projection at the time was to decrease the amount of time to make the application working. With some more time and slightly changes in our back end and front end infrastructure we would be able to implement several movie projections as well as multiple parking places and selection of specific parking place.

# Conclusion

The web application fulfils most of our requirements but there is still lots of work to do. As of now the web application gives all our actors basic functionality and it has built infrastructure to handle more functionalities.

We were limited in our progress by necessity of learning the Angular framework and how to properly set up the WEB API with entity framework. During this process we improved our skills in both fields and therefore it causes some inconsistency in the code, because we haven’t had time to apply new development procedures to already working and debugged functionalities.

With some time and a little help from senior developer we could improve UI and functionalities to state when this web application could be deployed and used in real life situations.