

-Samrat Bikram Shah  
-1815674

CIS Project

Game Review - Ragnarok

# Introduction:

This project expected us to reflect on various software development concepts and create a software system that collects data from a group of people and analyze it. Some of the criteria and concepts we had to explore from this project were using the Object-Oriented Programming Approach, Event-Driven Programming, Interoperability, Virtual Identity and using a third-party web-based Application Programming Interface (API). Based on this, we were asked to make an application with documentation and a video explaining these concepts.

# Topic background:

Using the given information, we also had to choose a related topic to work on and a programming language of choice. For this, I chose to make a web application that takes feedbacks in the form of Ratings and Comments from its users for an imaginary game called Ragnarok and analyses the obtained data accordingly. The scenario goes as:

Ragnarok is a recently made multiplatform game released by SBS studio. It is only released on PC so far but they are planning to release the game on multiple platforms in the future. They provided the games to a group of people to obtain feedback. It uses a website for gathering and further analysis of feedbacks so that the developers can apply changes to the game accordingly. The website is provided to the players through an in-game browser from where they can register for their Ragnarok game accounts to play the game and provide feedback in the process if they want.

# Features

The features of this application are as follows:

* It collects feedback from multiple users who will register and login providing their general information. Then analyze the results accordingly using their demographic and preferences regarding the topic and the feedbacks received such as comments, feedback types and ratings.
* There is an admin page that can view all analyzed results.

**Criterion:**

|  |  |
| --- | --- |
| **Criteria** | **Link to the report** |
| Client-Server Architecture (REST API) | [4.1](#_Client-Server_Architecture_(Front-e), [4.2](#_Client-Server_Architecture_(Front-e_1), |
| Event Driven Programming | [4.1.2](#_Event_Driven_Programming) |
| Virtual Identity | [4.3](#_Virtual_Identity) |
| Object-Oriented Approach | [4.4](#_Object_Oriented_Approach) |
| Interoperability | [4.5](#_Interoperability) |
| External Web API | [4.6](#_External_Web_API) |

Additional Work: [Sea-Level UML diagram, E-R diagram](#_Diagrams)

# Development and Testing

To meet the client-server architecture requirement, the application is divided into Front-end and Back-end.

## Client-Server Architecture (Front-end)

The Front-end GUI is made using Angular Framework and Bootstrap which has premade styles available.

### GUI

The GUI consists of dynamic webpages with routing. There is a Login Page, Register Page, Homepage that allows user to give feedback and an Admin Page which has all the feedbacks given by the user and the analyzed data. It also has a page for performing a CRUD Operation on the User.

|  |  |  |
| --- | --- | --- |
| Test No. | Objective | Result |
| 1. | Functional GUI | Pass |

Some images of the GUI are as follows:

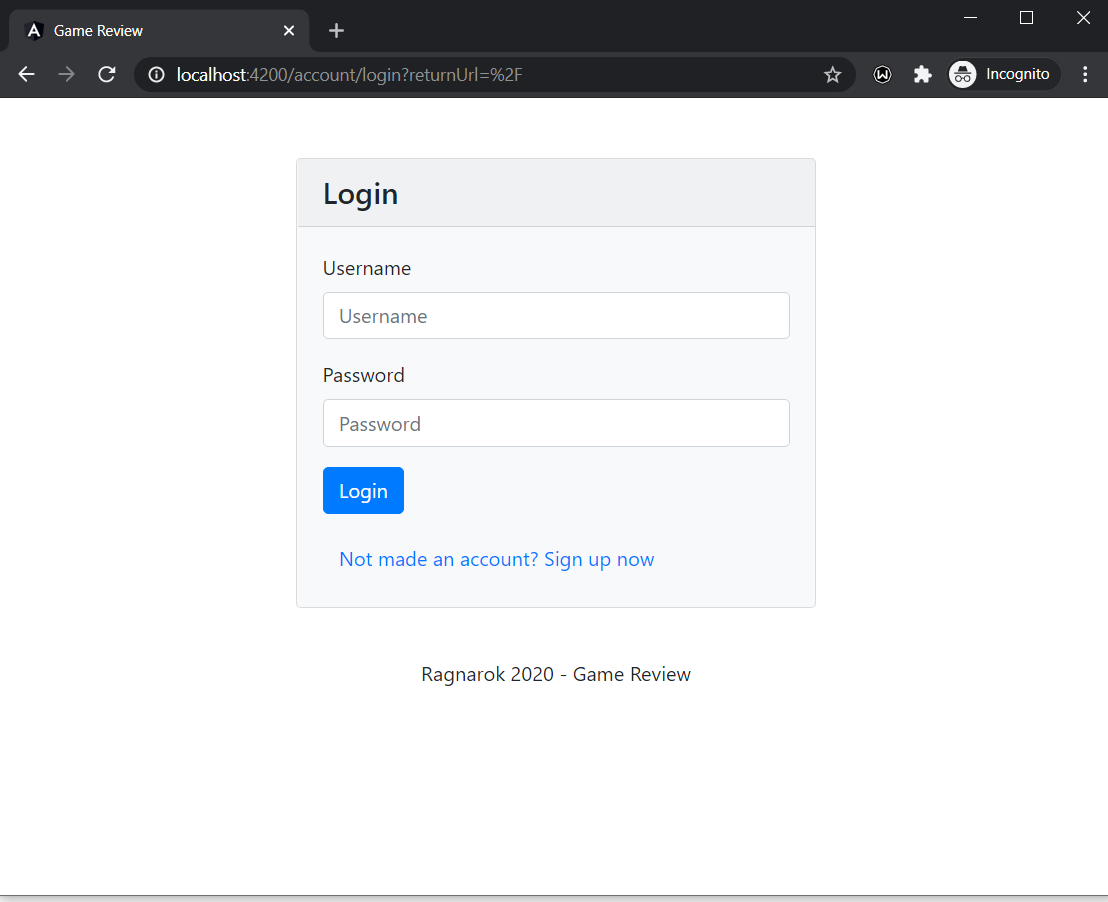


Fig: Login Page

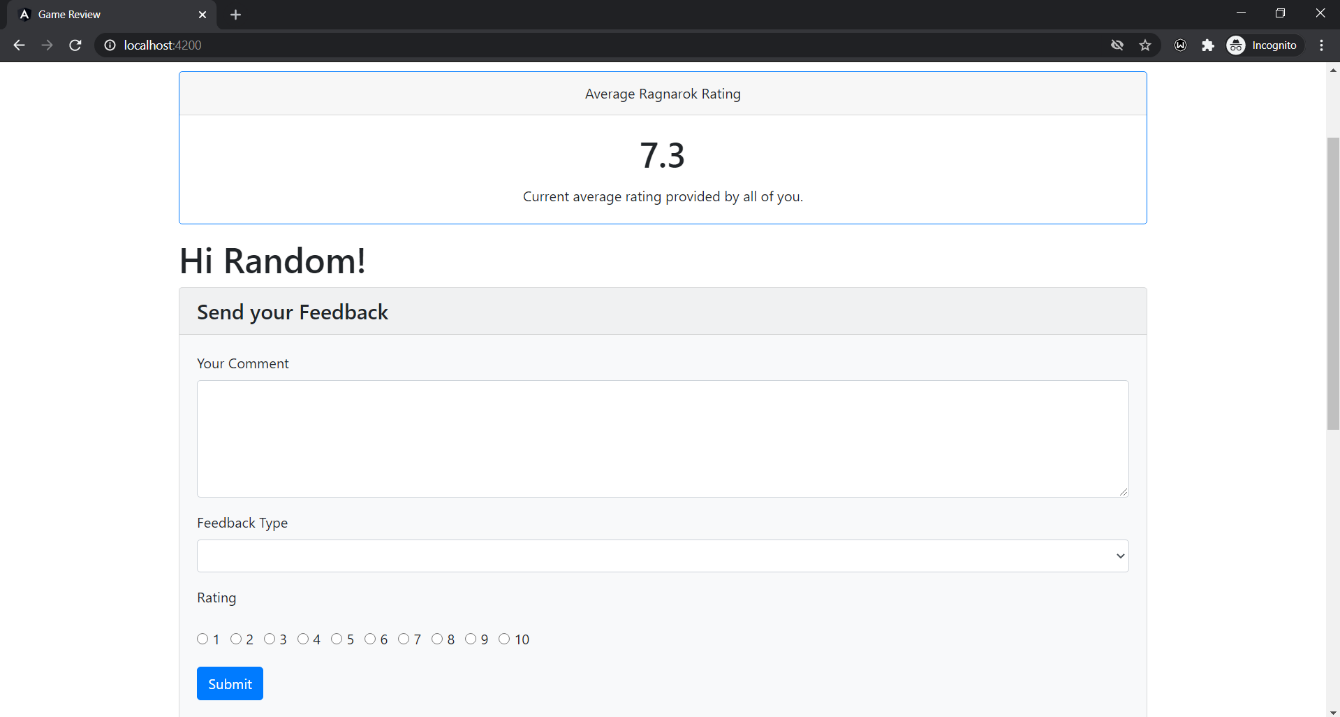


Fig: Homepage before Feedback

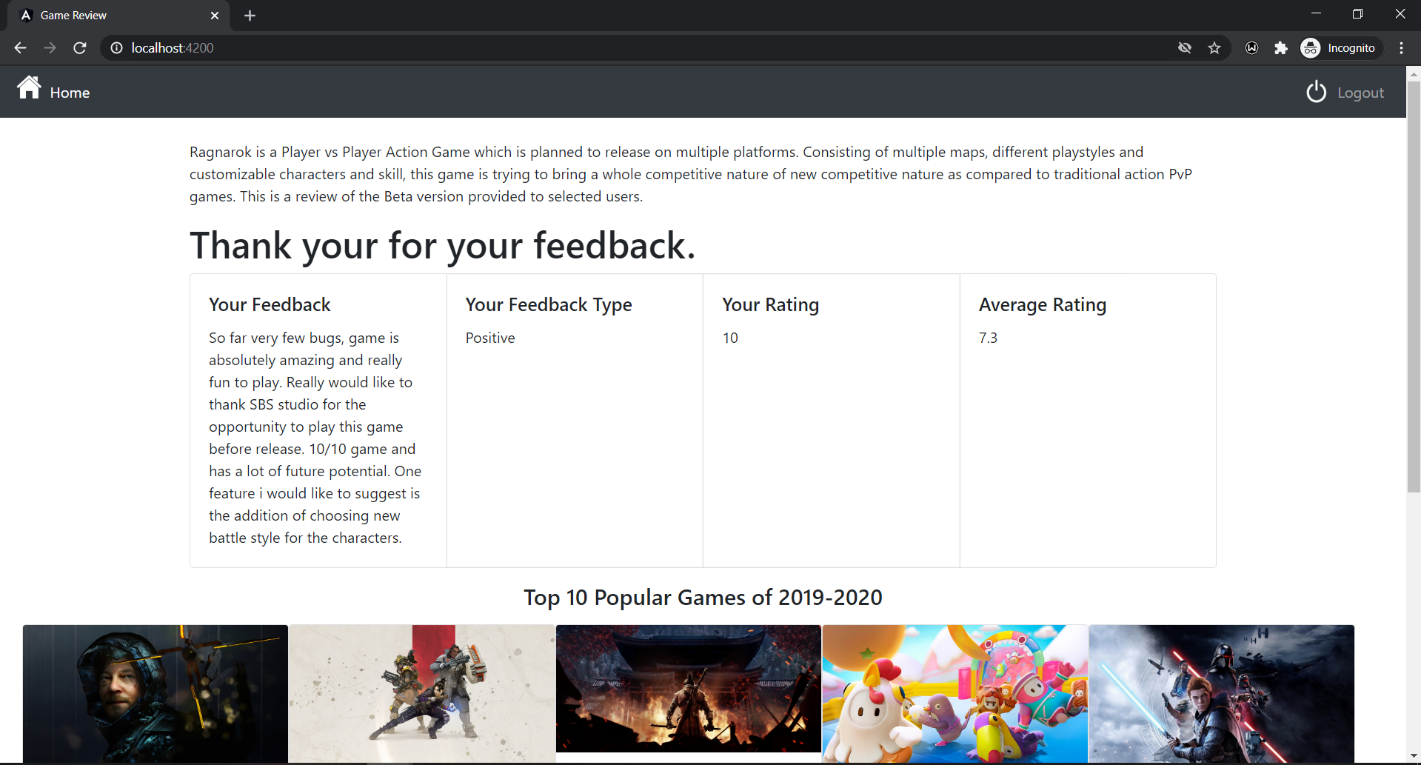


Fig: Homepage after Feedback

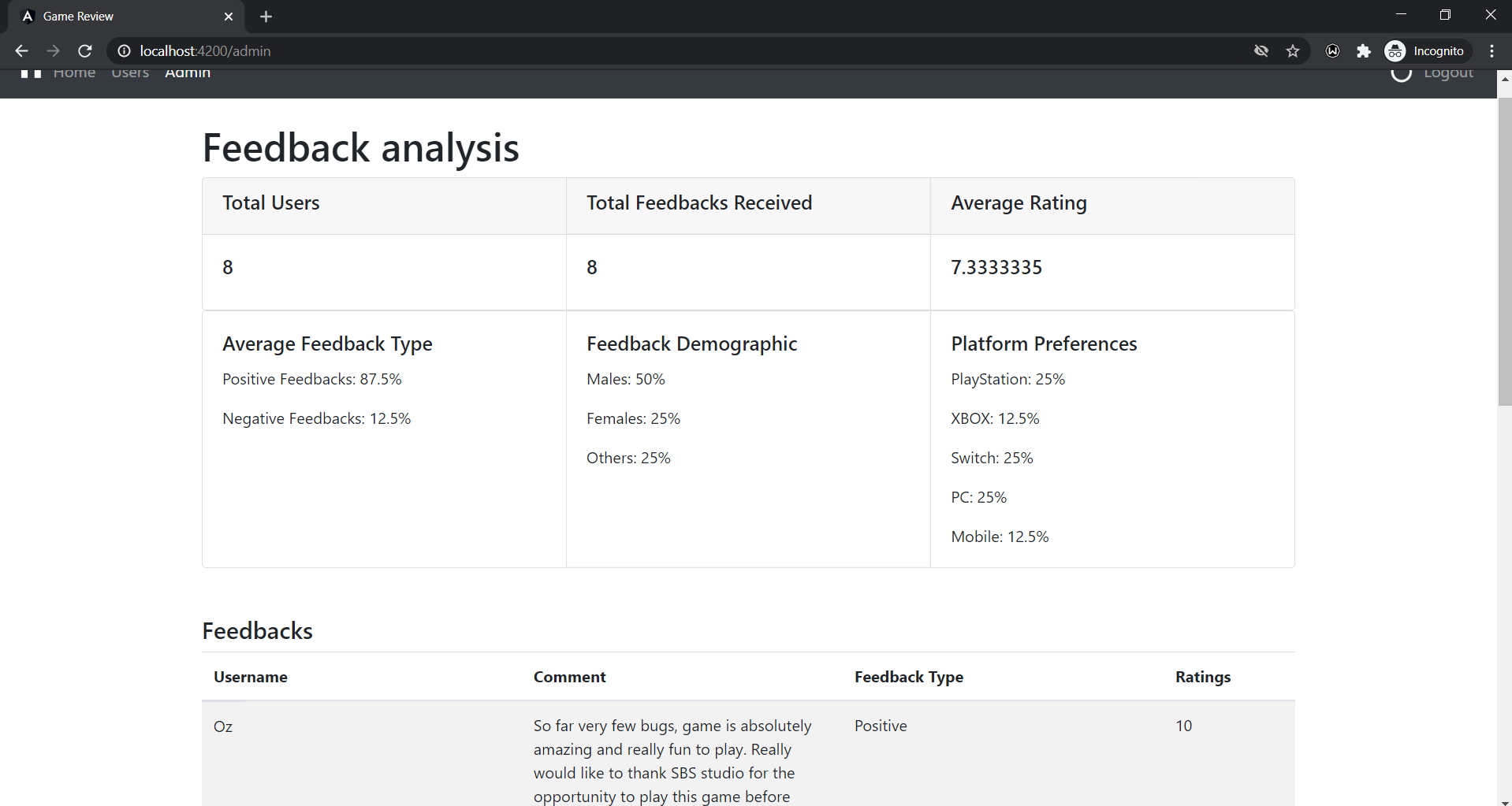


Fig: Admin Page

|  |  |  |
| --- | --- | --- |
| Test No. | Objective | Result |
| 2. | To make sure users can’t access the Homepage without Login | Pass |

The code are as follows:

@Injectable({

    providedIn: 'root'

})

export class AuthGuard implements CanActivate {

    constructor(

        private router: Router,

        private userService: UserService

    ) {}

    canActivate(route: ActivatedRouteSnapshot, state: RouterStateSnapshot) {

        const user = this.userService.userValue;

        if (user) {

            // authorised so return true

            return true;

        }

        // not logged in so redirect to login page with the return url

        this.router.navigate(['/account/login'], { queryParams: { returnUrl: state.url }});

        return false;

    }

}

Here the AuthGuard class is created which inherits CanActivate class from the Angular router library. This class is then used to check whether the user is logged in or not. If they are logged in it returns a true else returns a false and redirects to the login page.

This class is implemented in routing as:

const routes: Routes = [

    { path: '', component: HomeComponent, canActivate: [AuthGuard] },

];

|  |  |  |
| --- | --- | --- |
| Test No. | Objective | Result |
| 3. | To make sure only the admin account can access the Admin page | Pass |

The code is as follows:

@Injectable({ providedIn: 'root' })

export class AdminGuard implements CanActivate {

    constructor(

        private router: Router,

        private userService: UserService

    ) {}

    canActivate(route: ActivatedRouteSnapshot, state: RouterStateSnapshot) {

        const user = this.userService.userValue.username;

        if (user == "admin") {

            // authorised so return true

            return true;

        }

        // not logged in so redirect to login page with the return url

        this.router.navigate(['/account/login'], { queryParams: { returnUrl: state.url }});

        return false;

    }

}

Here a similar class like the Auth Guard is created but instead, it checks whether the logged-in account is Admin or not before providing access. Then this class is reused in multiple instances of the routing class.

const routes: Routes = [

    { path: 'users', loadChildren: usersModule, canActivate: [AdminGuard] },

    { path: 'account', loadChildren: accountModule },

    { path: 'admin', component: AdminComponent, canActivate: [AdminGuard] },

];

### Event Driven Programming

One of the requirements of this project was to incorporate an event-driven architecture. It is added in this project on various pages as buttons which calls the onSubmit() method.

|  |  |  |
| --- | --- | --- |
| Test No. | Objective | Result |
| 4. | The event handlers are working. Buttons are functional. | Pass |

An example of the event code:

For GUI:

<form [formGroup]="form" (ngSubmit)="onSubmit()">

…

<div class="form-group">

    <button type="submit" class="btn btn-primary my-1">Submit</button>

</div>

</form>

It uses html where the button and form elements are inside a form tag that calls the onSubmit() from the typescript file after the user clicks the submit button.

The onSubmit() method for the homepage is as follows:

onSubmit(){

        this.submitted = true;

        // stop here if form is invalid

        if (this.form.invalid) {

            console.log("invalid");

            return;

        }

       this.feedbackService.add(this.f.comment.value, this.f.feedback.value, this.f.rate.value, this.user.id)

            .pipe(first())

            .subscribe(data => {

                this.alertService.success('Feedback added successfully', { keepAfterRouteChange: true });

                this.isFeedback = true;

            },

            error => {

                    this.alertService.error(error);

                }

                );

    }

This basically performs an event after the button click which calls the service class who is handling the communication with the backend and adds a user feedback to the database. It returns an error if the user provides invalid data.

## Client-Server Architecture (Back-end)

The backend is made in C# using ASP.NET CORE. A REST API is used for the server-side code which can also be considered as an additional feature in a sense.

### Database Connectivity and Access

|  |  |  |
| --- | --- | --- |
| Test No. | Objective | Result |
| 5. | To make sure the Database is connected. | Pass |

Here, the connection string is set as a local sqlite database.

{

"ConnectionStrings": {

"WebApiDatabase": "Data Source=LocalDatabase.db"

},

}

It uses an in-built DataContext class to create an instance that connects and interacts with the database.

The code is as follows:

namespace WebApi.Helpers

{

public class DataContext : DbContext

{

protected readonly IConfiguration Configuration;

public DataContext(IConfiguration configuration)

{

Configuration = configuration;

}

protected override void OnConfiguring(DbContextOptionsBuilder options)

{

// connect to sqlite database

options.UseSqlite(Configuration.GetConnectionString("WebApiDatabase"));

}

///<Summary>

/// Connects the \*\*Users\*\* Entity class to the Database.

///</Summary>

public DbSet<User> Users { get; set; }

///<Summary>

/// Connects the \*\*Feedbacks\*\* Entity class to the Database.

///</Summary>

public DbSet<Feedback> Feedbacks { get; set; }

}

}

This class is used to connect and interact with the database with the set Connection string.

### Rest API

It is an API that uses HTTP requests to access and use data. It performs Create, Read, Update and Delete operation using POST, GET, PUT and DELETE requests. It supports various formats to communicate and return data but JSON is the most popular use. The reason to use this is Rest API supports client-server architecture. It is fast, reusable in multiple Front-end instances and Stateless in nature.

|  |  |  |
| --- | --- | --- |
| Test No. | Objective | Result |
| 6. | To ensure the API endpoints are working accordingly | Pass |

Some examples of the API endpoints used to access the database are as follows:

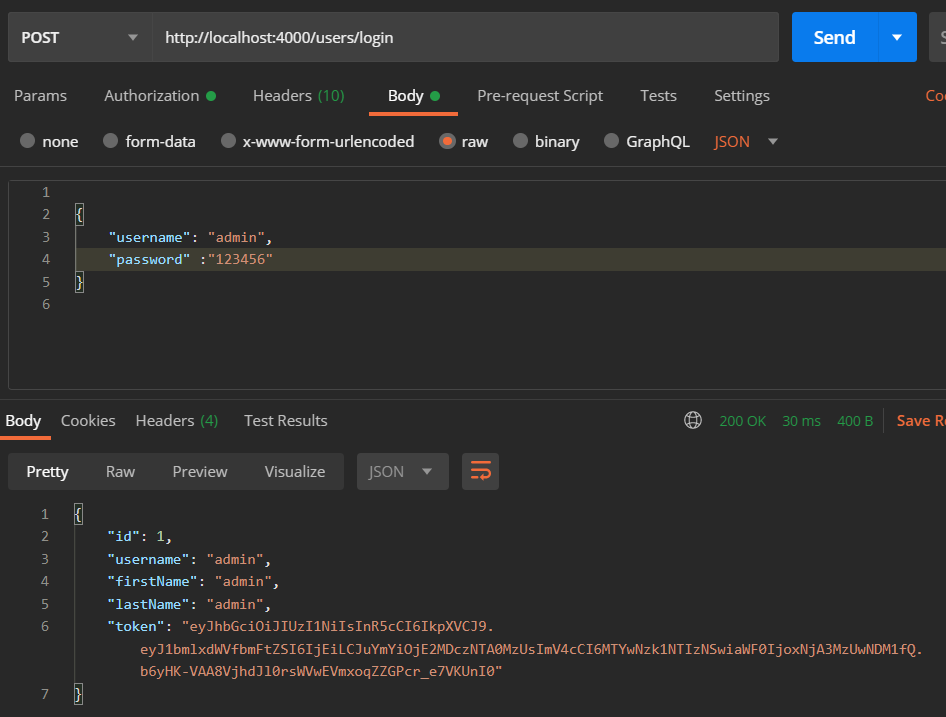


Fig: Login endpoint using POST

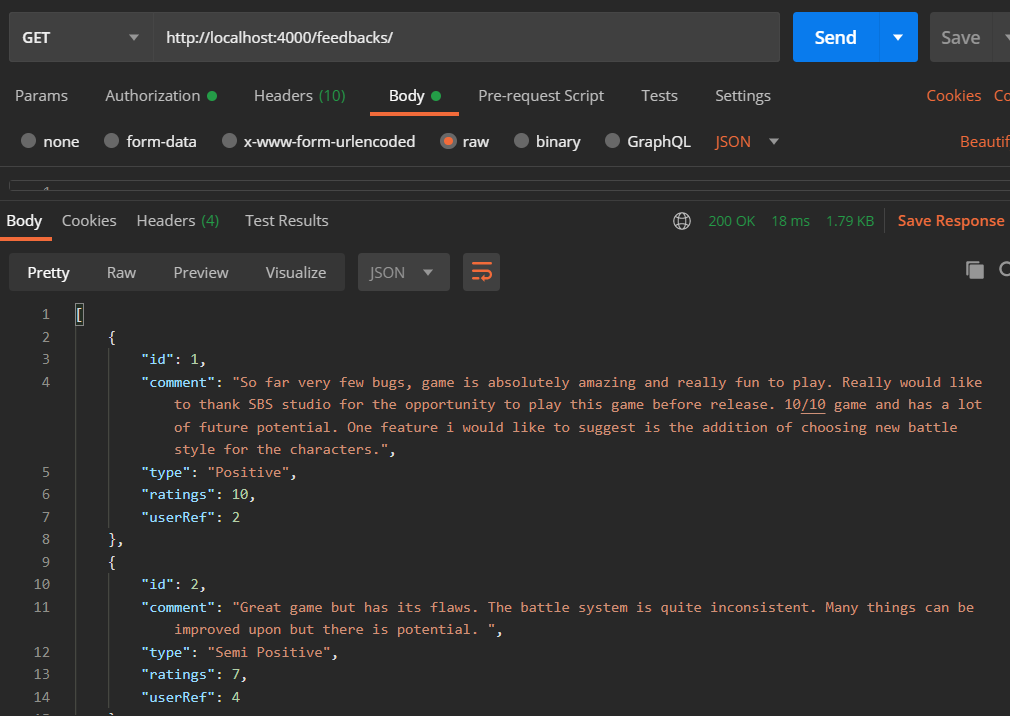
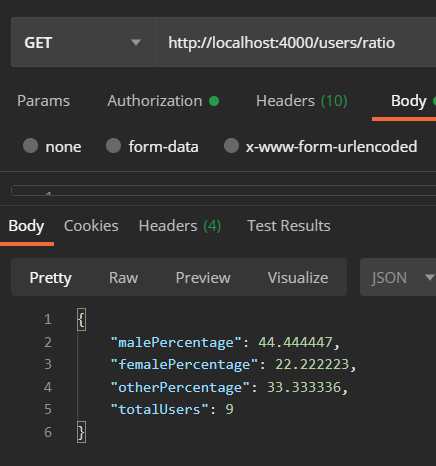


Fig: GET request to get all the Feedbacks Fig: GET request for analyzed data

There are several endpoints in the API which is configured to a service class via a controller that is responsible for handling all the backend logic such as validation, calculating the analyzed data and communicating with the database.

## Virtual Identity

Virtual Identity is one’s identity in the virtual world i.e. computers or the web. To achieve this, the user must register their details in the database and login accordingly. This is maintained using username and password. For the server-side, it uses the id and password sent from the client-side interface to login after matching with the database. It uses Angular’s Local Storage on the client-side to retain the user information sent back from the server.

**Server-side code:**

[AllowAnonymous]

[HttpPost("login")]

public IActionResult Login([FromBody]LoginModel login)

{

var user = \_userService.Login(login.Username, login.Password);

if (user == null)

return BadRequest(new { message = "Username or password is incorrect" });

var tokenHandler = new JwtSecurityTokenHandler();

var key = Encoding.ASCII.GetBytes(\_appSettings.Secret);

var tokenDescriptor = new SecurityTokenDescriptor

{

Subject = new ClaimsIdentity(new Claim[]

{

new Claim(ClaimTypes.Name, user.Id.ToString())

}),

Expires = DateTime.UtcNow.AddDays(7),

SigningCredentials = new SigningCredentials(new SymmetricSecurityKey(key), SecurityAlgorithms.HmacSha256Signature)

};

var token = tokenHandler.CreateToken(tokenDescriptor);

var tokenString = tokenHandler.WriteToken(token);

return Ok(new

{

Id = user.Id,

Username = user.Username,

FirstName = user.FirstName,

LastName = user.LastName,

Token = tokenString

});

}

This method of controller class handles the login portion that calls the service class’s Login method. It then creates an authentication token and returns the user information along with the token to the frontend.

The method in service class is as follows:

public User Login(string username, string password)

{

if (string.IsNullOrEmpty(username) || string.IsNullOrEmpty(password))

return null;

var user = \_context.Users.SingleOrDefault(x => x.Username == username);

// check if username exists

if (user == null)

return null;

// check if password is correct

if (!VerifyPasswordHash(password, user.PasswordHash, user.PasswordSalt))

return null;

// authentication successful

return user;

}

This method checks and compares the password with the database and returns a user if the user is found else returns an error message.

**Client-Side Code:**

|  |  |  |
| --- | --- | --- |
| Test No. | Objective | Result |
| 7. | To ensure the user’s virtual identity is maintained on client-side. | Pass |

login(username, password){

        return this.http.post<User>(`${environment.apiUrl}/users/login`, { username, password })

            .pipe(map(user => {

                // store user details and jwt token in local storage to keep user logged in between page refreshes

                localStorage.setItem('user', JSON.stringify(user));

                this.userSubject.next(user);

                return user;

            }));

    }

logout(){

        // remove user from local storage and set current user to null

        localStorage.removeItem('user');

        this.userSubject.next(null);

        this.router.navigate(['/account/login']);

    }

These methods are present in the service class of the client-side which acts as a bridge to communicate with the backend. It sends a post request to the database for login and retrieves user information if the user is registered in the database. It then stores the user details in the Local Storage so that it can be accessed anywhere in the website until the user logs out.

## Object Oriented Approach

This application is completely written in an object-oriented approach using its concepts. The written code is reusable in multiple instances and can be expanded as needed. Multiple classes and objects have been created. Concepts like Encapsulation, Abstraction, Inheritance have been used throughout the application.

### Encapsulation

Encapsulation is the process of hiding access to the data inside of the class from other classes. For this, the variables are kept private and can only be accessed through a public method if needed. Here, encapsulation is done in every important place. One of such examples is the UserService class which is the most reused class.

Code:

public class FeedbackService : IFeedbackService

{

private DataContext \_context;

public FeedbackService(DataContext context)

{

\_context = context;

}

Here the declared variables are private because their use is only limited within the class itself.

### Abstraction

Abstraction is the process of showing only the essential attributes or methods and hiding unnecessary details. It helps in reducing programming complexity and security. This project has used Abstraction in major server-side class i.e., the Feedback and User Services which is responsible for all the backend operations. To achieve this, the methods which are not used by other classes are all kept private and an interface is implemented to add an extra layer of abstraction. All the other classes then use the interface to communicate with the service class.

Code:

public interface IFeedbackService

{

IEnumerable<Feedback> GetAll();

Feedback GetByRef(int id);

Feedback Create (Feedback feedback);

RatingModel GetAverageRatings();

}

As you can see only the relevant method names are present in the interface and all the variables, implementation details and unused classes are hidden behind the interface.

### Inheritance

Inheritance is the process of using all the features of an existing class and building new methods and attributes over it. This enables the class to use methods of both the original and the new class. It saves a lot of time and promotes the reusability of code. Inheritance has been used in multiple instances of code in this application.

One of such examples is:

DataContext Class

public class DataContext : DbContext

{

protected readonly IConfiguration Configuration;

public DataContext(IConfiguration configuration)

{

Configuration = configuration;

}

protected override void OnConfiguring(DbContextOptionsBuilder options)

{

// connect to sql server database

options.UseSqlServer(Configuration.GetConnectionString("WebApiDatabase"));

}

///<Summary>

/// Connects the \*\*Users\*\* Entity class to the Database.

///</Summary>

public DbSet<User> Users { get; set; }

///<Summary>

/// Connects the \*\*Feedbacks\*\* Entity class to the Database.

///</Summary>

public DbSet<Feedback> Feedbacks { get; set; }

}

SqlDataContext Class which inherits from DataContext class.

public class SqliteDataContext : DataContext

{

public SqliteDataContext(IConfiguration configuration) : base(configuration) { }

protected override void OnConfiguring(DbContextOptionsBuilder options)

{

// connect to sqlite database

options.UseSqlite(Configuration.GetConnectionString("WebApiDatabase"));

}

}

Here, the DataContext class has been inherited by the SqlDataContext which overrides the methods using the SqlLite Connection String rather than the existing SqlServer one. By doing this, the code to connect the Entities to the Database didn’t have to be rewritten twice since this class already inherits it from the DataContext class. This allows us to use different connection strings, one to connect to a SqlServer database for production meanwhile, the other to connect to a MySqlLite database which can be used separately for development and testing.

## Interoperability

Interoperability is the ability of systems to exchange information. In this application, the front end communicates with the backend using API endpoints and exchanges information. Such requests are sent by the service classes from the front end.

The code example is:

@Injectable({ providedIn: 'root' })

export class FeedbackService {

    constructor(private http: HttpClient) {}

    add(comment, type, ratings, userRef) {

        ratings = parseInt(ratings);

        return this.http.post<Feedback>(`${environment.apiUrl}/feedbacks/create`, { comment, type, ratings, userRef})

            .pipe(catchError(e => throwError(this.errorHandler(e))

            ));

    }

    errorHandler(error){

        console.log(error);

    }

    getRatings(){

        return this.http.get<Ratings>(`${environment.apiUrl}/feedbacks/avg`);

    }

    getAllFeedbacks() {

        return this.http.get<Feedback[]>(`${environment.apiUrl}/feedbacks`);

    }

    getFeebackById(id: string) {

        return this.http.get<Feedback>(`${environment.apiUrl}/feedbacks/${id}`);

    }

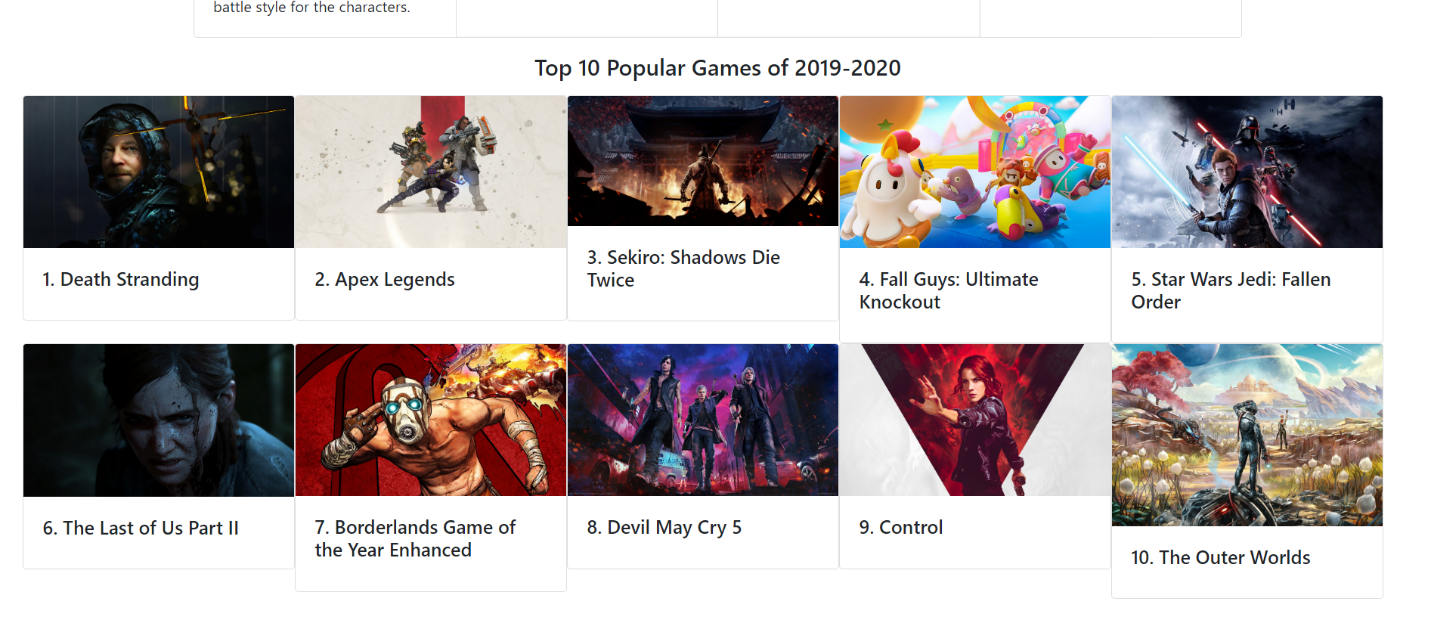
}

Here, the add method communicates with the backend using the POST method and adds feedback to the database. Likewise, the get methods fetch the data from the database by using get requests to the endpoints and use them in the frontend as required.

## External Web API

One of the requirements of this project was to use a web API provided by an external source. There are lots of such APIs that can be used for free, one of such API is a game database API which consists of all the ratings, popularity and feedbacks of recent games. This complements the theme of my project which is collecting reviews and ratings for a game. This API has been used to categorize the Top 10 games of 2019 and 2020 and display it on the feedback page. The API endpoint to access this is used from their official website, <https://rawg.io/apidocs> which can also be found on the rapid API website.

The webpage shows this as:



The code implementation of this API are as follows:

@Injectable({

    providedIn: 'root'

})

export class ExternalService {

    constructor(private http: HttpClient){}

    getGames(){

        return this.http.get<External>(`https://api.rawg.io/api/games?key=3175fbf4e77c406da00c6b12cce1701b&dates=2019-01-01,2020-10-30&ordering=-added`);

    }

}

This class is used to call a get request to the API endpoint which retrieves the top 10 games of 2019 and 2020. Then it is implemented on the website as follows:

HTML:

<div>

        <h4 class="text-center pt-3 pb-2" >Top 10 Popular Games of 2019-2020</h4>

        <div class="card-group style=width: 18rem;">

            <div \*ngFor = "let result of games.results; let i = index;">

                <div class="card" style="width: 18rem;" \*ngIf="i<10" >

                    <img class="card-img-top" src="{{result.background\_image}}" alt="Card image cap">

                    <div class="card-body">

                        <h5 class="card-title">{{i+1}}. {{result.name}}</h5>

                    </div>

                </div>

            </div>

        </div>

    </div>

TypeScript:

 this.externalService.getGames()

            .pipe(first())

            .subscribe(games=>{this.games = games;

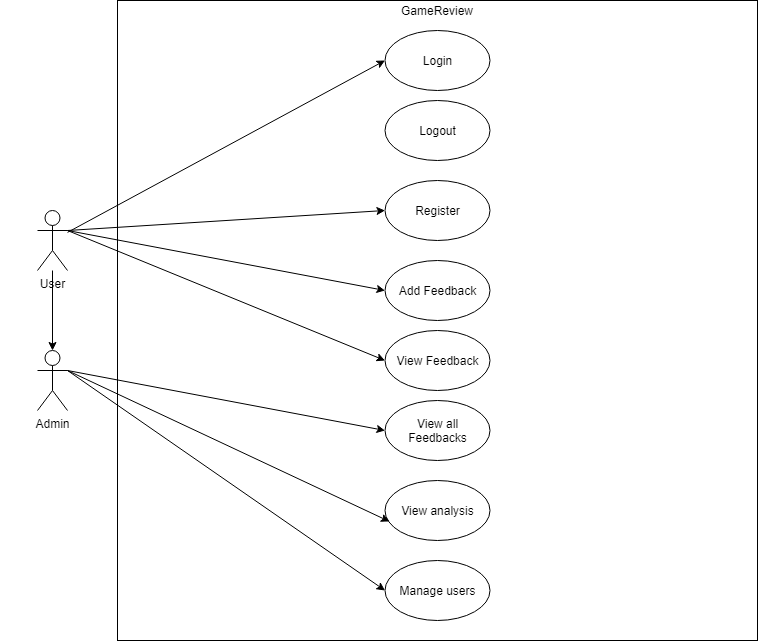
            console.log(games);

            });

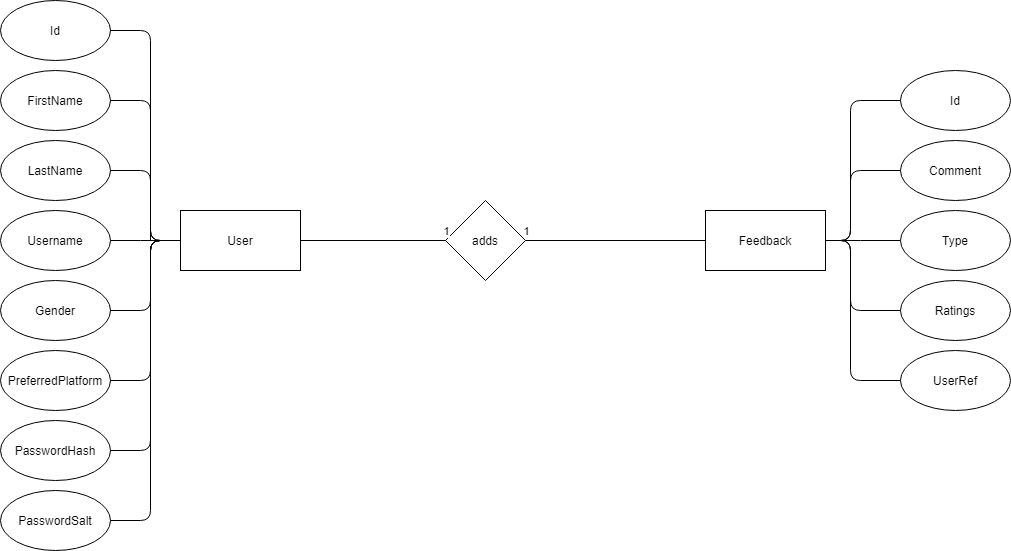
Basically, the GET request is sent to the API link which retrieves the names and images of Top 10 games and they are displayed in the webpage using Bootstrap cards.

## Diagrams

A sea-level UML diagram is made which represents the high-level functionality of the system. There are two actors User and Admin. Admin can do all the things user can and have their own usecases as well.

****

An E-R diagram is also made representing the database structure of the application. Here, there are two tables with User having 1 to 1 relationship with the Feedback table i.e. one user can add one feedback.



# Conclusion

In conclusion, the application topic, the whole development process and the documentation, everything was thought and delivered using all the requirements. All the required features are working as intended.