INFT3970 MAJOR PROJECT

CAM
TAG

TEAM 6

SHERIDAN GOMES - C3196901

MATHEW HERBERT - C3260254

DYLAN LEVIN - C3252150

DAVID LOW - C3260947

HARRY PALLETT - C3256688

JONATHAN WILLIAMS - C3237808

TABLE OF CONTENTS

INTRODUCTION

CamTag is a mobile game which is designed to promote an active gameplay experience throughout an entire day, or for as little as 10 minutes. The game took inspiration from the classic childhood game, tag, providing the core gameplay experience and concepts. Also, the game was influenced by the movie “Tag” released in 2018, providing the idea of a long-term game which can be played over many hours and in a large playing area. It aims to enhance the classic game by using today’s modern smartphones to provide an engaging user interface and exciting features. The core gameplay revolves around the player earning points by successfully taking photos of other players in the game; called tags. However, to earn points, other players in the game must review the photos taken and confirm the photo is a clear and eligible photo of the other player. Resulting in a peer review voting system for other players to earn points.

PROJECT OBJECTIVE

Within 13 weeks and with a budget of $50, to design, build, and deliver a fully functional game using modern web technologies, one which harnesses the modern hardware capabilities of mobile smart devices to provide an engaging augmented reality gameplay experience. The purpose of this game is to create competition between friends which involves taking photos.

CORE GAME & FEATURES

Tag Gamemode

Tag is the core game mode of CamTag which involves at least three players in a large scale game of tag. Players earn points by taking photos of other players within their game using their mobile device and other players successfully voting on photos submitted. After the time limit has elapsed, the player with the most amount of points is the winner of the game.

No Account Required To Play

CamTag users are not required to create an account with login information such as username and password to create or join a game of CamTag. Each game of CamTag is its own entity, where each player inside the game is treated as a brand new player, so no persistent player information is recorded, allowing players to avoid the hassle of creating an account and remember login credentials to play. The only information required to play is a contact such as an email or phone number, and a nickname.

Custom Settings

A user can create a game of CamTag with options to adjust the game settings to their liking to best suit the style of game they wish to play. The user can change the game mode, time limit, amount of ammo and many more options. Players can then join this game by using the created game code provided to the host player. *(please see Business Rules Section for a detailed explanation of all settings the player can adjust.)*

Joining A Game

Users can join a game of CamTag given they have been provided with the game code supplied by the host player (the player who created the game). Users can join a game before the game has begun, but, also can join a game in progress if the host player created the game with this option enabled.

In-Game Photo Capture

Players can capture photos of other players using the in-game camera interface, bypassing the need of taking photos via the devices default camera application and uploading from the camera roll. The technology which allowed the capability of a live video stream and photo capture within a web browser is WebRTC. *(please see Technologies Section for further explanation of WebRTC and its use within the application)*

Real-time Application Updates

While players are interacting with the web application, they can receive live updates from the game server without having to refresh their web page. This feature is used prominently throughout the application as it provides the capability of live notification updates to inform players of essential things which have occurred in their game. Also, it provides a platform to prompt the user interface to redirect to different views of the application, such as redirecting to the main gameplay screen once the game has begun, displaying the voting modal when a new photo has been uploaded. The technology which provided this capability is SignalR. *(please see Technologies Section for further explanation of SignalR and its use within the application)*

Out-Of-App Notifications

One of the limitations while using web applications is the inability to provide updates to clients while the client is not directly using the application. Our solution to this problem was using a text message and email notification system to provide players with important updates about their game. If the player is not using the web application they will receive a text message or email notification in situations such as the game has begun, a photo has been uploaded and requires voting to name a few. The technology which enabled the use of text messaging is Twilio *(please see Technologies Section for further explanation of Twilio and see Business Rules Section for further explanation of when the player will receive a text message or email notification.)*

Peer Reviewed Tagging

When a player takes a photo of another player and uploads the photo, all players in the game except the “tagging” player and the player being “tagged” must vote on the submitted photo. The other players review the photo to confirm the submitted photo is clear and eligible, containing the “tagged” player. If the voting player finds the photo to be valid, they will vote “successful”; however, if the photo is not a clear picture and the “tagged” player cannot be identified, they will vote unsuccessful. After all other players have voted, if the majority voted “successful”, the “tagging” player will receive a point, otherwise, will be notified of an unsuccessful attempt.

Last Known Player Locations

When players take a photo in a game of CamTag their latitude and longitude is submitted when the photo is uploaded to the server. Players can view a map which displays the last known location for each player, displaying the selfie image of each player in different locations on the map. The last known location will be the latitude and longitude of the last photo taken by each player, so if the player has not taken a photo, they will not be displayed on the map. This is a similar feature found in first person shooting games, when players shoot they are displayed on the minimap.

Limited Ammunition & Ammunition Replenishment

Players have limited ammo while playing a game of CamTag to reduce the possibility of spamming photos throughout the game. Each player has three photos by default, but, this can be adjusted up to 9 in the custom game settings. Players ammo will also replenish throughout the game when a player takes a photo. Each time a player takes a photo, their ammo count reduces by one, and after a certain amount of time specified in the game settings, their ammo will replenish by one, refilling the ammo used to take a single photo.

Player Verification

When a player attempts to create or join a game of CamTag, they must first verify themselves using an available phone number or email address. Players are then verified upon entering a code which is sent via text message or email. Player verification is used to ensure players are using valid contact details and can receive game updates while not using the application.

Randomly Generated Game Codes

When a player creates a game of CamTag, they become a game host. The game host will be provided with a randomly generated game code which can then be given to other players as a means of joining the game. This system allows players to quickly join a game without a complicated lobby browser and also limits the possibility of random players joining the game.

BATTLE ROYALE FEATURES

The Battle Royale Gamemode

Battle Royale is a secondary game mode of CamTag which is very similar to the core gameplay. However, the aim of the game is to be the last man standing within a constantly constricting playing area. When a player successfully tags another player the “tagged” player is eliminated from the game. The last player remaining in the game is the winner. The playing area in this game mode is also constantly restricting and becoming smaller, this is to force players to move throughout the game and will result in an increase in tags as the game progresses as players are in close proximity of each other.

Constricting Playing Area

Throughout a Battle Royale game, the playing area will continuously shrink, forcing players to be closer to one another. The playing area will be reduced around a centre point which is chosen by the host player when setting up the game. The playing area will lessen according to the time remaining in the game. For example, if the game has 40% of the original time remaining, the radius will be 40% of the original size. *(please see Business Rules section for an in-depth overview of how the playing area is calculated.)*

Disabling Players Out Of The Zone

When a Battle Royale player is outside of the playing zone, and they attempt to take a photo, they will be disabled and can no longer take photos. The reason for this approach is because of the technical limitations of a web application, and the player's location cannot be tracked when the application is closed. When a player takes a photo outside of the zone, they will be disabled for a certain period. This period is calculated as 5% of the original game time limit. Once the disabled period has elapsed, the player will be re-enabled and can begin taking photos once again. *(please see Business Rules section for an in-depth overview of how the disabled time is calculated and scheduling of player re-enablement.)*

Eliminated Players

Eliminating players is a core component of the Battle Royale game mode. When a player successfully tags another player in a Battle Royale game, the tagged player is eliminated from the game. For a player to be eliminated, the majority of the other players must vote “SUCCESSFUL” on the submitted photo. Once the player is eliminated, they will have no in-game functionality other than voting on submitted photos. *(please see Business Rules section for an overview of why eliminated players can vote on photos.)*

BUSINESS RULES

Uploading A Photo

1. The player can only submit a photo of a player who is in the same game and is taggable. The tagged player is not deleted, has not left the game and is verified.
2. The tagging player can only submit a photo of a player when there is not an incomplete photo record with the same TakenByID and PhotoOfID. This is used to avoid issues of players constantly following and tagging the same player over and over.
3. There is a time limit of 15 minutes for players to vote on a photo. After 15 mins the uploaded photo will be checked to see if voting has been completed. If voting has not completed, the photo will be automatically set to successful, and the tagging player will earn a point.
4. Each time the player takes a photo, even if not submitting the photo for voting, the player will lose ammo. Ammo will be scheduled to be replenished after a certain amount of time as specified in the game settings. This business logic can be found inside ***PlayerController.cs*** inside the method ***UseAmmo()*** which calls the ***ScheduledTasks*** class to schedule the player's ammo to replenish after the specified time.

Points 1-3 business logic can be found inside ***PhotoController.cs*** inside the method ***Upload()*** which calls the stored produce ***usp_SavePhoto*** to upload the photo to the database and calls the ***ScheduledTasks*** class to schedule the photo to be checked after the 15 minute voting period.

Beginning The Game

1. Only the host Player can begin the game.
2. Before the game can begin, there must be at least three players in the game.
3. All players in the game need to be verified before the game can begin.
4. The game will be scheduled to begin “PLAYING” after the start delay has elapsed. The scheduling of code to execute can be found inside ***GameController.cs*** inside the method ***BeginGame()*** which calls the ***ScheduledTasks*** class to schedule the game to begin after the start delay.

Points 1-3 business rules can be found inside the stored procedure ***usp_BeginGame***

Completing The Game

1. The game will complete when there are less than three players in the game. This business logic can be found inside the stored procedure ***usp_LeaveGame.***
2. The game will complete when the time limit expires. The code is scheduled to run at the game end time which will update the game record in the database to completed and then inform all connected clients that the game is completed. This business logic can be found inside ***GameController.cs*** inside the method ***BeginGame()*** which calls the ***ScheduledTasks*** class to schedule the game to complete after the end time.

Last Known Locations

1. When a player views the map, the last known locations of all other players are displayed. The last known locations are determined by finding the last photo taken by each player in the database as the photo stores the latitude and longitude at which the photo was captured. This business logic can be found inside the stored procedure ***usp_GetLastKnownLocations.***

Joining a Game / Creating a Game

1. A player's nickname can only be numbers and letters, no special characters. This business logic can be found inside ***Player.cs*** inside the ***Setter*** for the Nickname property.
2. A player must verify their contact details before successfully joining a game. The player verifies their contact details by entering a 5 digit verification code which is sent to their email address or phone number. A verification code is a number which is between 10000-99999. The business logic can be found inside the ***PlayerController.cs*** inside the method ***VerifyPlayer()*** which validates the verification code format and calls the stored procedure ***usp_ValidateVerificationCode*** to verify the code is correct.
3. A player can only join a game when the GameState is "IN LOBBY", otherwise, if the host player enabled "IsJoinableAtAnytime" inside the game settings when the game was created the player can join the game at any time.
4. The maximum number of players in a game is 16.
5. The nickname entered by the player must be unique to the game they are joining, so two players inside the same game cannot have the same nickname.
6. The contact, email address or phone number, must be unique system-wide. So two players currently in a game, different or the same, which is not completed cannot have the same contact details. Once the game is completed the contact details can be reused in another game.

Points 3-6 business rules can be found inside the stored procedure ***usp_JoinGame.***

Leaving a Game

1. If the player leaves the game when the game state is IN LOBBY or STARTING the player record will be deleted to allow other players to use the same contact details in other games.
2. If after the player leaves and the number of players is below three the game will complete.
3. When a player leaves the game any incomplete photos will be removed. So any photos submitted by the leaving player which voting has not been completed will be deleted.
4. When a player leaves the game any votes they need to complete will be removed and then all photos affected will be verified to see if the photo is now completed voting after the player has left the game. If photos have now been completed after the player left, notifications and live updates will be sent out as required.
5. If the host player leaves the game while the game state is "IN LOBBY" the game will be set to complete and all other players removed from the lobby.

All business logic can be found in ***PlayerController.cs*** inside the method ***LeaveGame()*** which calls stored procedures ***usp_LeaveGame*** and ***usp_EndLobby***.

Removing Unverified Players

1. Only the host player can remove unverified players from the game.
2. The host player cannot remove a verified player from the game.

All business logic can be found inside ***PlayerController.cs*** inside the method ***RemoveUnverifiedPlayer()*** which calls stored procedures ***usp_RemoveUnverifiedPlayer.***

Voting

1. If the result of the vote is a tie, the photo is deemed to be unsuccessful. This business logic can be found inside the stored procedure ***usp_VoteOnPhoto.***

Battle Royale

1. If the player attempts to take a photo outside of the playing zone, they will be disabled for a certain amount of time. The algorithm to check if the player is within the playing zone is by first calculating the current radius of the game and then comparing the player's distance from the centre point. If the player's distance from the centre point is greater than the current radius, they are outside of the playing zone. This algorithm can be found inside ***Game.cs*** inside the method ***IsInZone().***

   The algorithm to calculate the current radius is done by using the percentage of time remaining in the game. So if the game has 40% of the original time remaining, the radius will be 40% of the original size. There is a minimum value for the radius, so if the radius calculated is below 20 meters, the radius value to be returned will be 20 meters. This algorithm can be found inside ***Game.cs*** inside the method ***CalculateRadius().***

   The time to be disabled is 5% of the game's time limit. However, there is a minimum value of 2 minutes and a maximum value of 30 minutes. If the calculated disabled time is less than the minimum or greater than the maximum, the time to be disabled to set to the minimum or maximum respectively. The algorithm for calculating how long the player will be disabled for is found inside ***Game.cs*** inside the method ***CalculateDisabledTime().***

   The use of all the above algorithms when a player attempts to take a photo can be found inside ***PlayerController.cs*** inside the method ***BR_UseAmmoLogic()***.
2. When a player submits a photo, all players including eliminated players can vote on the photo. The reason why eliminated players can also vote is to solve the problem of a 1v1 scenario, a player must vote on the final photo to determine who is the last player standing. This business logic can be found inside the stored procedure ***usp_BR_VoteOnPhoto.***
3. When a photo has completed voting, and the voting is successful the tagged player will be eliminated from the game. This business logic can be found inside the stored procedure ***usp_BR_UpdateVotingCountOnPhoto.***
4. The game will end when there is one player left standing or the time limit elapses.

Live In-Game Updates

The below in-game updates apply to players who have the web application open and are successfully connected to the SignalR hub. In-game updates refer to the backend server updating connected clients via SignalR to receive updated information without the client refreshing their page.

1. When a new player joins a game the following live in-game updates will occur:
    a. If the GameState is “IN LOBBY” or “STARTING”, the lobby list will be updated.
    b. If The GameState is “PLAYING” the scoreboard and notifications list will be updated.
2. When a game is now “STARTING” all players lobby will update to display the starting time (the time the game will begin and players can start tagging each other).
3. When a game is now “PLAYING” all players will be redirected to the main camera view.
4. When a player uploads a photo, all other players except the tagging and tagged players will be presented with a voting screen.
5. When a player leaves a game the following line in-game updates will occur:
    a. If the GameState is “IN LOBBY” or “STARTING”, the lobby list will be updated
    b. If The GameState is “PLAYING” the scoreboard and notifications list will be updated.
6. When a game is completed, all players will be redirected to the end game scoreboard.
7. When the host player leaves the game when the GameState is “IN LOBBY” all other players will be redirected to the main menu.
8. When a players ammo has been replenished the following live updates will occur:
    a. If the player’s ammo has been replenished, but not from empty, the player’s ammo counter found on the main camera view will be updated.
    b. If the player’s ammo count has been replenished from empty, the player's ammo counter found on the main camera view will be updated as well as their notifications list.
9. When a player reads their notifications, their unread notification counter found on the main camera view will be updated.
10. When a Battle Royale player takes a photo outside of the zone they will become disabled, and their main camera view will be updated, and their notification count will also be updated.
11. When a Battle Royale player is re-enabled after the specified amount of time their main camera view will be updated, and their notification count will also be updated.
12. When voting on a photo has completed, the following live updates will occur:
    a. The player’s notification count will be updated.
    b. The player’s scoreboard statistics will be updated.
    c. If the game is a BR game and the tag was successful, the tagged player will be eliminated and redirected to the end game scoreboard.

In-Game Notifications

The below are all the scenarios in which a player will receive notifications in-game while currently connected and using the web application:

1. A player joins the game while the game state is “PLAYING” and the joined player has verified themselves.
2. A player leaves the game while the game state is “PLAYING”.
3. Ammo has replenished from empty; now the player can begin tagging again.
4. The BR player is now disabled.
5. The BR player is now re-enabled.
6. Voting on a photo has now completed.

Out of Game Notifications

The below are all the scenarios in which a player will receive notifications about the game via text message or email while they currently have the application closed:

1. A player joins the game while the game state is “STARTING” or “PLAYING” and the joined player has verified themselves.
2. The game state is now “PLAYING”.
3. The game state is now “STARTING”.
4. A new photo has been submitted and is ready for voting.
5. A player leaves the game while the game state is “STARTING” or “PLAYING”.
6. The game has completed.
7. Ammo has replenished from empty, so now the player can begin tagging again.
8. The BR player is now disabled.
9. The BR player is now re-enabled.
10. Voting on a photo has completed when the player is the tagging or tagged player.

TECHNICAL INFORMATION

Requirements To Play

General:

- A valid email address or mobile phone number are required.
- Device must have a camera.

Browser Compatibility

- Android: Google Chrome, Firefox
- iOS: Safari

Mobile Connectivity

- 3G minimally compatible.
- 4G LTE.
- Wi-Fi.

Camera Compatibility

- 5 megapixels or higher.

App Permissions

CamTag may request permission to access the following:

- **Camera**: For taking pictures.
- **Location**: To access precise locations, for map tagging (GPS and network-based).

TECHNOLOGIES USED

Angular

The Cam Tag frontend/interface layer is implemented using Angular, which can provide a very ‘app-like’ experience through a web page, making it feel like a native app. This approach is very similar to just running two separate apps. .NET Core acts as the web API, and Angular runs as an independent app/site that interfaces via REST protocols.

We have full flexibility over how the front-end is set up and run, and we can use any Javascript-related tools and plugins. This means that the frontend is entirely separate, and is processed locally, freeing up the game server for logic processing.

.NET Core

Cam Tag was built in the backend using .NET Core web API. The use of a .NET Core web API was selected as it was the best fit for our intended application structure, having the front-end and backend completely separate and the backend serving the front-end via REST APIs. .NET Core also allowed functionality such as SignalR and Twilio to be implemented much more easily as there were much more documentation and tutorials available for .NET Core. Another major deciding factor for using .NET Core was our team’s experience with C# throughout our degree.

SignalR

CamTag’s in-game notifications and live application updates rely on the SignalR real-time web functionality, with the server invoking methods on client devices to display content and run functions. The library enables high-frequency updates from the server, with connection management handled automatically.

In summary, the ***IHubContext*** is used by the controllers using dependency injection to obtain the current hub context and client connection information. ***IHubContext*** is then used to create an instance of ***HubInterface*** which contains methods to invoke client-side methods which handle live application updates. SignalR has multiple modes of transport that can be interchanged depending on the quality of connectivity. SignalR uses four different technologies to support its real-time data updates. Ideally, the SignalR connection will use a WebSocket connection if possible. However, it will fallback to using Event Source then forever frame and finally AJAX long polling.

Twilio

The CamTag SMS notifications are built using Twilio. Twilio handles the complexities of managing a mobile carrier and global regulation. For our use case, the messaging API will be used as one of the ‘out of game’ notifications medium, as there is a constraint that the user will not receive the in-game notifications while not actively on the CamTag webpage. The use of Twilio involves setting up an account on their website and purchasing credit, from there you are given a unique Twilio phone number, account SID and authentication token which is stored within the backend for use. The use of Twilio can be found inside the ***TextMessageSender.cs*** class found inside the ***Helpers*** folder. Sending a text message is very simple, only requiring five lines of code. You merely create a Twilio client using your private SID and auth token and specify the “To” , “From” and “Message”.

OpenLayers

Our Game is supported by the OpenLayers API which allows maps to display through our web application, which can work with geospatial data. The main concepts being explored in our use case are:

- **Map:** Load a rich and interactive map in which data is visualised.
- **Pushpins:** Represent point based data using graphical images on the map.
  Our Pins are based on the users last known location.
- **User Location:** Display a user's location on the map using browser locations finding functionality.

Initially, we looked at using the Google Maps API. However, we had issues with the licencing and didn’t want to use an option that may end up costing money down the track.

We then switched to the Bing Maps API. Again we had issues, this time with not being able to draw circles on the map for custom radii, and also CSS issues displaying it in a modal. In the end, OpenLayers was also to provide both a free option for displaying a map, and also provide a relatively easy way of adding our custom pins and radii without any formatting issues.

WebRTC

CamTag is built using WebRTC, an open web technology that lets you access the camera's video stream within the browser. The ***getUserMedia*** API provides access to the devices media streams such as the camera. WebRTC is used throughout CamTag as it is the foundation of how the application captures images to provide the core functionality of the game.

We use WebRTC in two places specifically: the selfie page and the main screen. In each case, the code will first try to get a video feed that is constrained to the view that we want. This will be either the 'user' or selfie camera, and the 'environment' or outward-facing camera. If it is unable to find the correct camera, it will, as a backup option, find any camera that is available and attempt to use that. If it is still unable to find any video feed, it will check the error code given by the WebRTC API, and display an appropriate response to the user. Once the API returns a valid video stream, this is assigned to a video element in HTML, and starts streaming the current data stream.

When a player goes to take a photo, an HTML canvas element is created to capture a single frame from the video at that point in time. This frame is cropped to form a square, and scaled down to a 512x512 image for easy handling. The canvas can then output the video frame as a DataURL string, which is exported in a JPG format. This DataURL string can be passed to the backend, displayed in an img HTML element, or used in whatever other way we need.

CSS Grid

The main user interface for CamTag is built using CSS Grid. Specifically it uses a number of grid template areas for a range of aspect ratios. Each grid template area is given a number of rows and columns. Each aspect ratio had the equivalent ratio of rows and columns resulting in square areas. When between two aspect ratios this square area squishes either vertically or horizontally. In order to reduce the occurrence of such squishing three portrait, three landscape and one square aspect ratios were implemented. This squishing could have been further reduced by adding additional aspect ratios.

Elements were then placed within a grid area by assigning a short label. Each element was given a two character long label with empty areas represented by two dots. The main screen of CamTag for example uses SVG images underneath a button element. These images will maintain their aspect ratio when the screen changes. This causes the buttons to be slightly larger than the image but retains the required functionality and makes the interface look cleaner. Each of these buttons is only 2-3 lines of HTML and can be placed in a single div. Should this have been done with the widely used method of nesting divs, both the HTML and underlying CSS would be much harder to read. Instead CSS Grid has allowed the majority of pages to have a maximum of 3-4 layers of nesting, instead of the likely 6-7 required with traditional methods.

SOFTWARE USED

SOFTWARE APPLICATION	PURPOSE	LANGUAGES AND FRAMEWORK
Visual studio community 2017	To develop the backend of the game	C#, .NET Core
Visual studio code	To develop the frontend of the game	Typescript
Microsoft SQL server express 2017	To develop the database for the game	T-SQL
Github	Source control, managing iterations of the backend and frontend	NA
Postman	For testing the API requests for the backend.	NA
Jenkins	Provides us with the ability to have an automatic deployment of the production build.	Powershell

BACKEND ARCHITECTURE

The backend is a hybrid between N-Teir and MVC. We are using N-Tier which provides a data access layer and MVC provides models which map to the database entities and have embedded business logic for data validation and specific model methods. Controllers are used to handle requests to control the request workflow and business logic regarding the workflow for each request. The views, however are not implemented in the backend as the front-end is acting as the view. The folder structure of the backend is as follows:

- **Response<T>** –The response class is the foundation of how the backend communicates with the frontend. This class contains a generic property, called ***Data***, which is used to store information such as objects or primitive types which will be returned to the frontend. This class also includes an ***ErrorCode*** and ***ErrorMessage***, where the error code is used to indicate to the frontend that an error occurred during the request, and the error message is the respective message to display to the user. All API requests made to the backend use this class as a means of returning data to the frontend.
- **Controllers** – The controller classes are the API endpoints which the frontend makes requests to. These classes control the program flow of the request, creating models from the data provided as input to the request and call the data access layer to perform CRUD operations.
- **Models** – Models are classes which map to the entities found inside the database. Models can perform specific business logic such as data validation for each property and perform entity specific business logic. For example, the ***Player.cs*** class can generate a verification code and the ***Game.cs*** class can confirm if a Battle Royale player is within the playing zone. Models are passed to and from the Data Access Layer in order to be saved/updated in the database, and also for object retrieval to return to the front-end
- **Data Access Layer** – The Data Access Layer contains multiple classes, each for the different entities (players, games, photos) which provide database functionality. All of the entity DAL classes will inherit from a base ***DataAccessLayer.cs*** class. Database access is done solely through stored procedures because it allows precondition checking and business logic implementation inside the database. The Data Access Layer classes for each entity provides CRUD functionality as well as specific business logic functions. The Data Access Layer also contains all the .SQL scripts used to create the database, views, functions and stored procedures.
- **Model Factory** – The model factory is a class found inside the Data Access Layer which is used to build models from the rows returned from the database. When a DAL method is called which reads data from the database, the model factory will use the Data Reader and build a model from the data returned. This is done to allow the controllers to operate in an Object Oriented manner and have centralised object creation when retrieving information from the database, allowing for easy maintenance if the database entity changes.

- **Hubs** – Hubs is a folder which contains classes which use SignalR. ***ApplicationHub.cs*** is the SignalR hub which players connect to when lanuching the application. ***ApplicationHub.cs*** manages the client connections and disconnections using a connectionID which is generated automatically by SignalR. However, managing these connections and mapping of a connectionID to a player is done manually using persistent storage within the database. ***HubInterface.cs*** is a class which invokes all the live updates to connected clients for application updates. The ***HubInterface.cs*** takes in a ***IHubContext<ApplicationHub>*** which is the context for the SignalR hub, containing all the connected clients which can be receive SignalR commands. Each method within the ***HubInterface.cs*** is related to a live application update which occurs throughout the course of the game.

FRONT-END ARCHITECTURE

Most of the code for the app itself resides in ***src/app*** folder. This is where virtually all student-written code resides, except for ***styles.css***.

Within this folder, the breakdown is as follows:

- **Core** – contains essential files that are instantiated once for the whole app.
- **Main** – contains files for the main view in the game.
- **Shared** – contains any files that can be shared among other views and instantiated as many times as needed.
- **Start** – contains files to handle the creating and joining of games.
- **Summary** – contains files to display summary pages (lobby and scoreboard).
- **Voting** – contains the files for the voting modal.
- **App-routing.module.ts** – handles all the routing for the application.
- **App.component.html|ts|css** – handle the initial view that others are loaded from. This component always exists throughout the life of the app.
- **App.module.ts** – handles the files needed for the app and delegates to other modules.

Angular uses a very modular structure, made up of components, modules and service. Components are the building blocks of the view and code and handle most visible things. Services are singleton classes that provide services and app-wide functionality. Modules handle how the files and components work together, especially when loading and injecting components into other modules so that they are accessible.

All assets used in the project are stored in ***src/assets***. ***Src/environments*** is used for any app config that may change depending on whether it is in development or production.

FRONT-END DEPLOYMENT DETAILS

Before the frontend can be compiled, you must have Node.js installed on your system. Once it is set up, follow these steps:

1. Open up a command prompt and navigate to the base folder for the project (this is the one where the angular and package JSON files reside).
2. Run the command ***'npm install @angular/cli'*** to install the Angular compiler. If this does not work, you may have to install it globally with ***'npm install -g @angular/cli'.***
3. Run ***'npm install'*** to install dependencies for the project.
4. Run ***'npm start'*** to start the active compiler and debug environment. This will open a browser window with the app once it is finished.

Generic info for setting up an Angular dev environment can be found here:
https://angular.io/guide/quickstart

BACKEND DEPLOYMENT DETAILS

System Requirements:

- Visual Studio 2017
- Download and install the latest .NET Core 2.1 SDK and Runtime environments from https://www.microsoft.com/net/download

Setup

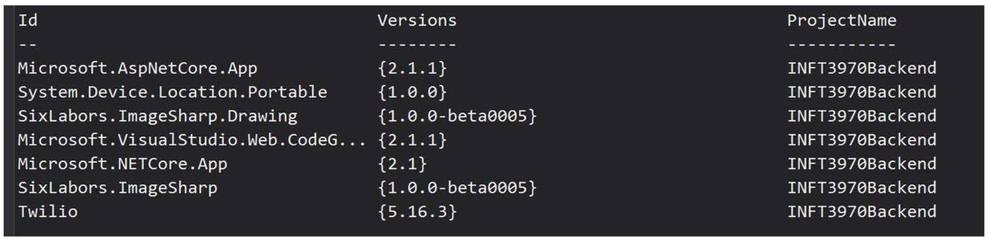
1. To deploy and run the backend, download the backend Visual Studio solution called ***INFT3970Backend***.

2. Run or import the ***CreateDB.sql*** script using SSMS, this can be found at: ***"~\INFT3970Backend\INFT3970Backend\DataAccess Layer\Database Scripts\CreateDB"*** which will create the database and SQL login information***.*** Ignore all warnings regarding "missing module" as this is because the stored procedures are being created out of order.

3. Ensure SQL Server is configured to allow dual authentication, allowing SQL server login credentials to be used, not integrated security.

4. Open the ***'INFT3970 Backend'*** in Visual Studio via the ***INFT3970Backend.sln*** file.

5. If NuGet packages do not automatically install or restore the required packages, please install the following NuGet Packages using ***NuGet Package Manager***.

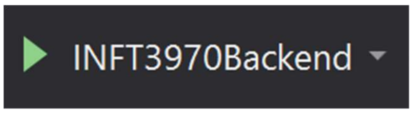


```
Id                                    Versions                                  ProjectName
--                                    --------                                  -----------
Microsoft.AspNetCore.App              {2.1.1}                                   INFT3970Backend
System.Device.Location.Portable       {1.0.0}                                   INFT3970Backend
SixLabors.ImageSharp.Drawing          {1.0.0-beta0005}                          INFT3970Backend
Microsoft.VisualStudio.Web.CodeG...   {2.1.1}                                   INFT3970Backend
Microsoft.NETCore.App                 {2.1}                                     INFT3970Backend
SixLabors.ImageSharp                  {1.0.0-beta0005}                          INFT3970Backend
Twilio                                {5.16.3}                                  INFT3970Backend
```

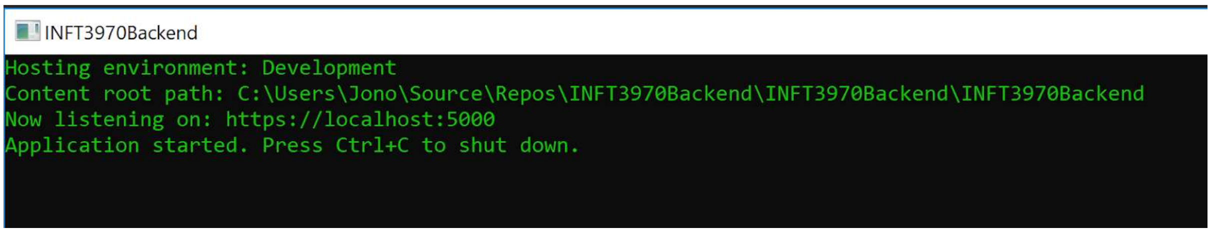
NOTE: If SixLabors.ImageSharp and SixLabours.ImageSharp.Drawing are not in NuGet Package manager please install them via the **Package Manager Console** using the following commands:

- Install-Package SixLabors.ImageSharp.Drawing -Version 1.0.0-beta0005
- Install-Package SixLabors.ImageSharp -Version 1.0.0-beta0005

6. Run the solution, ensuring that from the dropdown option next to the ***'run'*** button, you have selected ***'INFT3970Backend'*** and not the default ***'IIS Express'***.



7. This will open a console window which will state that the server is now running.



```
INFT3970Backend
Hosting environment: Development
Content root path: C:\Users\Jono\Source\Repos\INFT3970Backend\INFT3970Backend\INFT3970Backend
Now listening on: https://localhost:5000
Application started. Press Ctrl+C to shut down.
```

DEPLOYMENT TROUBLESHOOTING

By default, the backend will operate at https://localhost:5000, which is defined within the ***Program.cs*** file. If for some reason you cannot get the backend to run on https://localhost:5000, and you now have it running as a different URL or port number, the frontend configuration data is stored in ***src/environments***. The development URL is in the ***environment.ts*** file. These should be configured to communicate with the updated URL and port.

If you are certain that the backend is running, the frontend is pointing the correct URL, and the frontend still cannot connect to it, you may need to accept the SSL certificate. In Firefox, this can be accomplished by manually navigating to the URL of the server, and accepting the certificate when it prompts you about the security issue.

Chrome may be able to be configured to allow insecure self-signed certificates by entering the URL ***'chrome://flags/#allow-insecure-localhost'*** and changing the options from there.

TESTING THE APPLICATION

The easiest way to test the application is on a computer which has a webcam. The app is enabled to work in a desktop environment. In order to test the application follow the deployment details for the backend and frontend as listed above and navigate to the frontend URL. When testing the application, you are able to use mobile phone numbers as Twilio has a premium account which can send text messages to any phone number.

If you have any issues while setting up the backend, please contact **Jonathan Williams** via his university email, and he can provide guidance to solve any backend issues. If you have any problems with the frontend, please contact **David Low** via his university email, and he can provide guidance to solve any frontend issues.

Testing On A Mobile Device

If you wish to test on a mobile application, please contact **David Low** via his university email address to arrange the start up of our live production server for you to test the application without having to perform the lengthy process of setting up IIS. We recommend testing on an Android device as it provides the best UI experience. However, all the functionality is available on iOS.

iOS Testing Conditions

Please be aware there is a few restrictions when testing the application due to Apple's constraints for iOS:

1. Safari is the only web browser which can be used as WebRTC is not supported in others.
2. After taking a photo of another player, you must rotate the phone to landscape in order to select the player you have tagged.

GENERAL ISSUES

Scope creep is a common issue that plagues project teams in that the requirements and deliverables of the project change throughout the development and iterative cycles of the design. Our team throughout the timeframe of the project had clearly defined core goals that were essential to the project; these core goals included things such as the camera functionality, voting system, notification system, map functionality and others as state previously. These goals were decided upon early in the planning stages of the development, as were some stretch goals that we would possibly implement in the future if the time permitted. Some of the stretch goals included extra game modes such as Assassin mode, Battle Royal mode and Infected mode and power-ups. Our schedule only allowed that these additions be made once the core structure of the game was done and ready to present. The design was met with some issues and require modifying and some additions to support our base goals. However, our team never strayed too far from what the focus of the project, with much consideration being placed in our initial phases of planning to save valuable time towards the end of the project that could potentially be redesigning parts or trying to fix major bugs.

Throughout the implementation of the UI(approx. Weeks 8-9) our team discussed and prioritised which stretch goals could be feasible in the time that we left remaining but we always stayed true to the initial plan and schedule. However, there was some variation and missed deadlines as planned due to team members having other commitments such as other course assignments. However, these setbacks did not overly affect our project and final deliverable.

TECHNICAL ISSUES

Mobile Application Running On Multiple Platforms

Our initial concept for the game was to design a native application that would be deployed on a mobile phone. However, with the design requirements stating that the project must be fully functioning and polished. Our team decided that it would not be feasible to develop a mobile application, as it would take 1) Too much time to develop for an android and IOS device 2) No one in our team had much experience developing IOS applications. The solution to the problem was to develop a website that would be able to be accessed on a mobile phone and act as a native app would.

Technology learning curves

To make the most of the time given to us it was prudent to play to the strong points of our academic backgrounds, that being the majority of our group was proficient in software development. Our team had to research into different technologies to meet some of the features that we planned to implement. Our projects planning phase went heavily into researching each technology. Our team experienced some issues in regards to SignalR. Initially setting up the project to successfully use SignalR where business logic code could successfully invoke a method on a specific client was challenging and required in-depth research into the problem. The solution to this problem was dependency injection using IHubContext found inside each controller. The use of dependency injection allowed each controller upon each request an interface to the SignalR hub which contained all connected clients, allowing method invocation on a specific client if required throughout the business logic.

Geolocation and GPS

As a function of our game is to display player locations on a local map, we needed to implement location tracking. This proved to be difficult due to the limitations in a browser-based app, where location data can only be obtained while actively viewing the web page. This ultimately meant that our game functionality needed to work around this, in which we decided just to track the player's location when they take a photo. This information is used to check if they are within the game radius.

While implementing this, we dabbled with multiple map APIs, with difficulties in drawing a circular boundary over the map. Initially, we tried Google maps, where we were unable to draw a circle and had challenges displaying profile pictures on player locations.
Bing maps have both of these functions, although had many issues with rendering and had multiple mandatory text overlays. Ultimately we finalised with the OpenLayers map API, which provided all the functionality we desired in an open source accessible format.

Issues Displaying on iPhones

Ensuring the application worked on iPhones was challenging as Apple heavily restricts the functionality and permissions of 3rd party browsers such as Google Chrome. WebRTC, the primary technology used to display the live camera view and capture photos can only be used in Safari and are not available in 3rd party browsers.

Safari also proved challenging for CSS and UI. When developing the frontend and designing the UI, the UI would display fine on Android, across all different screen sizes and browsers. However, Safari would not display correctly or would incorrectly scale. One of the major issues was regarding the tagging popup. When a player takes a photo, they are presented with a popup where they can select whom they have tagged. Safari would not display this drop-down list unless the phone was in landscape mode.

Deploying to a Production Environment

One thing we wanted to set up early from the start was a proper, publicly-accessible production environment. Due to the nature of how our game is played, we needed to have a central server running where multiple people could access it from their phones to be able to test together.

One issue we had with deploying to a production environment is with WebRTC. WebRTC requires a secure connection when being used in a production environment, and will not allow traffic to be sent over HTTP. To solve this, we bought a cheap domain name. Initially, we were going to set up Let's Encrypt for free TLS encryption, but the host we chose offered a free certificate for a year, so we ended up using that instead.

Another issue we have had with the production environment is restricted access, as the site has been flagged as malware by one or two security companies, and was blocked on the university network for a couple of weeks. It is possible that the domain name that we purchased has been used maliciously in the past, and triggered some alerts, as it does have a previous history.

Since we were already going to host the app on a server, we decided to do it with proper production builds of the frontend and backend. We already decided to use SQL Server Express, so it required no changes to be set up. The server is running Windows Server 2016. Both the frontend and backend are served through IIS. To get Angular to work on IIS, we had to install a re-routing module, as Angular handles routing itself. Apart from that, it was relatively easy to serve, as the production build of Angular includes all JavaScript, HTML and CSS files to host directly from a folder. The backend was a little more challenging to get set up. Dot Net Core has a built-in server called Kestrel, so IIS needs to be set up as a reverse proxy to communicate with Kestrel, which was fine as IIS could handle the TLS certificate, and send the proxy request to Kestrel. IIS also has another module which can be used to serve .NET Core applications, so it was pretty straightforward to create the production build, then get IIS to host it from the folder.

The final component which we added was an automation server called Jenkins. This was a combination of experimenting and adding practicality. Instead of manually having to pull the updated code onto the server and build it ourselves, David set up Jenkins to watch specific branches on our repositories for the frontend and backend, and when changes were pushed it would automatically download them, re-build production builds for both ends, and set them up to continue being served. This made it immensely easier to test small changes on the dev server, instead of having to go through the whole process every single time manually.

MEETINGS

Online Meetings: Every Monday at 7:30 PM
Group Meetings: Every Thursday at 12:00 PM - 1:00 PM & 1:30 PM – 3:00 PM

Our teams had two consistent meetings each week throughout the whole project. The first being a video chat on each Monday of the week at 7:30 PM which on average ran for about 45 minutes, where we briefly discussed what we have done since our Thursday meeting and what we were planning on doing for the next week. It was also a great opportunity for questions to be raised and points to be brought up in time to be discussed and planned before going into the Thursday meetings.

Our second meeting throughout the weeks was a 60-minute meeting in person before the consultation time at 12:00 PM on Thursday. During this time we reviewed the work that each member had for the week and discussed problems that were present in the project. After the scheduled consultation time we then have an informal meeting and group work period in which we discussed issues brought up in the consultation and also assigned work to each member of the team for the week.

INDIVIDUAL ROLES & RETROSPECTIVE

Upon forming our group, we discovered that we have an almost perfectly diverse range of skill-sets and preferences. This meant that we could easily delegate portions of the project accordingly. The following are the individual roles of each member and a personal retrospective.

Dylan Levin

During the development of CamTag, I worked with Jonathan and Mathew to build the backend behind the game and to create and discuss the many business and game rules to go along with it. Throughout this process, I have learned new concepts and strengthened those which I know, particularly with our hybrid MVC class structure, and the implementation of new libraries such as WebRTC, Twilio and SignalR. This project has been a highly enjoyable endeavour, and I can surely speak for the team by saying that it has turned out fantastic and just as we imagined from the conception.

Jonathan Williams

I was lead backend programmer for the project and was responsible for the database design and stored procedure functionality. One of my primary roles on the backend was designing the data access layer to allow models to be effectively created and returned from the database to be used throughout the controllers and returned to the front end for display. I also spent most of my time building the request to serve the front end and focused on SignalR methods.

I have learned a lot about project management, teamwork and communication skills throughout the project. Working in a large team, all with different skill sets and some working in different areas of the application tested my communication skills to ensure all parties were on the same page on what tasks needed to be completed. Also, I have also learned a substantial amount about the use of SignalR and how it can be useful in other applications. The use of SignalR was challenging but also rewarding; it was difficult to understand and implement at first. However, as our team researched proper implementation and design guidelines we were able to overcome our initial challenges and implement it successfully.

I am thrilled with our team's result. I believe we produced a quality application that has real-world value and can easily be expanded upon. One thing I think our team could improve on is time management and meeting deadlines. Towards the end of the project, some of our scheduled deadlines were not met and resulted in development being pushed back. However, this is understandable as most of the team had commitments for other courses and each assessment for other classes can be unpredictable in how difficult and how much time they take.

David Low

I was primarily responsible for coding the frontend app, and ensuring that the development on backend and frontend would work together smoothly. I also kept track of the project schedule and progress.

I have learned a lot about managing an actual project and setting realistic expectations about what goals can be achieved within a specified period. I also learned a lot about deploying a project to a real production environment, and some of the issues and security considerations to be aware of when doing so. I'm really happy with how this aspect of the project went, as it was something not covered within the course, so it's great to see how streamlined the end result is.

Harry Pallett

As the user interface designer and developer on the project, I was responsible for creating mockups, turning those mockups into a real working interface, and user testing the implemented interface. Most of my time was spent working out how the interface would scale on different devices, as we hoped to have a unified experience across various platforms. The solution I settled on was to combine CSS Grid and media calls for various aspect ratios to layout our main screens. This resulted in very clean HTML markup as individual elements were positioned in a grid based on a Grid Area Template.

I spent very little time on the HTML markup as all I needed to implement was text, buttons, input options and areas for images / video. When I did spend additional time on the HTML markup, it was due to a lack of forward planning of modals which were mostly implemented using absolute positioning and viewport units. I am now able to very quickly design user interfaces with very minimal HTML and CSS, whether using grids or absolute positioning and viewport units. This will come in handy when developing further fluid designs. My hope is that I can transfer what I have learned throughout the project to my future design projects.

Sheridan Gomes

During the planning process, I gave my input on different aspects of the game and was involved in multiple discussions with all group members regarding the business rules and ideas for the game, also brainstorming the different modes and analyzing the game for limitations at may arise. During the development process, I worked on parts of the backend and some parts of the business logic.

The final year project has been an excellent experience for me in developing different skills that I will need in the future, also working in a team was helpful as I learned how to break complex tasks into parts and steps, plan and manage time, refine understanding through discussion and explanation, give and receive feedback on performance, challenge assumptions and develop stronger communication skills. Also while working on the project I learned about new technologies like SignalR and WebRTC, that will be valuable for the future use, my teammates helped me understand different aspects of these technologies that I had a hard time understanding.

Mathew Herbert

During the course of the semester, I was apart of the backend development team. Working in the team was Jonathan, Dylan and me, we spent a lot of time during the initial phases of project researching technologies to support the functionality of the game and then implementing them.

Notably, I gained a great deal more insight into what working apart of a larger team is like with communicating with other members who were involved in different aspects of the project. For example, working in the backend team to deliver the correct data and structures to the front end team. I hope to take away the lessons about communicating with a team from this project, as I believe we managed it efficiently and kept consistent throughout the course of the project. Over the time I was introduced into some frontend frameworks such as Angular which I would be interested in studying in the future

FUTURE DEVELOPMENT

Our team discussed the possibilities of continuing development after the project has completed. All team members see an excellent opportunity to expand upon our original idea and develop a fully functioning application which may become popular in the future. Our plans for future development is to transform the application in a native app for both Andriod and iOS. The reason for this is because it allows greater functionality regarding map display and location tracking as the application can obtain permission to track location when the application is closed. This will be beneficial for the Battle Royale game mode.

Developing for native platforms may be difficult and more time consuming, however, our application backend is interoperable as it communicates with the front-end via REST protocols. Using REST allows the backend structure to stay the same and have any front-end such as web applications or native apps, consume the REST responses and processing accordingly.

Our team has discussed also introducing other game modes and functionality into the native applications. The extra game modes and features were our stretch goals for this project which did not have enough time to be implemented. Other game modes include “Assassin”, where each player has a target, and they must eliminate their target. Another game mode is “Infected”, where one player is infected to start, and when the player tags another, they also become infected. In “Infected”, once all other players in the game are infected the game is over. An example of the extra features which could be implemented is “Geo Location Power Ups”, where players can go to a specific location in order to obtain a power-up. Power-ups could include; extra ammunition, camera zoom and a UAV where the player can view live locations of other players in the game.

APPENDICES

How To Play Guide

Tag Gamemode

The aim of the game is to score as many points as possible within the time limit. To score points, players must take photos of other players in the game. Every other player in the game except the tagging player and the tagged player must vote on the photo before voting time expires, deciding if the photo is successful or unsuccessful. If the majority of votes submitted by other players is successful, the tagging player earns a point. Otherwise, the tagging player does not earn any points. If all other players in the game do not vote, the photo is marked as successful by default. Within the game, the player has access to a camera in order to take photos, a notifications list to view necessary updates within the game, a map to see the last known locations of other players, a scoreboard and a help menu. After the time limit expires, the player with the most amount of points is declared the winner.

Battle Royale Gamemode

The aim of the game is to be the last player standing at the end of the game. Players play the game within a limited area which is continuously shrinking, forcing players to be within close proximity of each other. Players are eliminated throughout the game when players take photos of each other. When a player takes a photo of another player, if the majority decision of the vote is successful, the tagged player is eliminated from the game. Otherwise, the tagged player remains in the game. The voting system is the same as the core game mode. However, players still can vote even after they have been eliminated. Players have access to all the same functionality as the core game mode, such as the ability to take photos, view notifications, view the scoreboard and view a map which outlines the last known locations of each player. However, the map also displays the restricted playing area using a circle to outline the outer boundary. If a player attempts to take a photo outside of the playing area, they are disabled for a number of minutes.

Creating A Game

The player hosting the game is the person to begin the game, the game settings are initialised at this point, and the player can issue the room code for other people to join the game. The player can customise the game based on a number of options including:

- Game Mode.
- Game Duration.
- Maximum Amount of Players.
- Join game in progress (a setting to allow users to either join the game at any time or only when the game is in the lobby state).
- The amount of ammunition (photos) a player has at one time.
- The ammunition refill timer (how long it takes for ammunition to refill).

A step-by-step is as follows:

1. On the main menu, select *"Create Game".*
2. Enter in your details such as nickname and contact details.
3. Verify your contact details by entering the verification code sent to the contact.
4. Adjust the displayed game settings.
5. Provide all other players with the generated game code, allowing them to join the game.
6. Once all the other players have joined the game, click *"Begin Game"*.
7. Wait for the start delay period as set in the game settings and begin playing.

Joining A Game

A step-by-step is as follows:

1. On the main menu, select *"Create Game".*
2. Enter in your details such as nickname and contact details.
3. Verify your contact details by entering the verification code sent to the contact.
4. Enter a game code as provided by the host.
5. Wait for host to start the game, and then for the start delay period.
6. Play!

Casting Your Vote

Once a player uploads a photo the voting screen is displayed. Here is where you can make your decision. The green button means "successful", the red button means "unsuccessful".

Viewing Notifications

You will receive in-game notifications throughout the game which will inform you of important events which have occurred. To view your notifications click the notification icon found at the top left-hand corner of your screen. This will display your unread notifications, and you have the option to toggle to view all notifications received.

Taking a Photo

On the main camera view, take a photo by pressing the photo button located at the bottom of the screen. Taking a photo will launch a popup, where you can select the player whom you have tagged and submit the photo for voting. Once you have submitted the photo or cancelled, your ammo will be reduced. However, your ammo will replenish after a certain amount of time.

Viewing the Map and Scoreboard

The map can be displayed by pressing the map pin icon found at the bottom right-hand corner of your screen. This will launch the map where you can see the last known location of all other players in the game. To see the scoreboard, press the menu button found at the bottom left-hand corner of the screen, then select scoreboard. This will display each player's statistics such as number of kills and deaths.

Troubleshooting

The following are a few recommendations for solving a few errors when accessing and using the game:

If the browser cannot load the website:

- Make sure that the browser application on your phone is up to date with the current patches.
- Try using a different browser application if the current one is not working correctly.
- Check to see if the URL that you entered was the correct one.

If there are difficulties with an internet connection:

- Try restarting the wifi on your phone (if using wifi to access the internet).
- Check whether the mobile data is turned on for your phone.

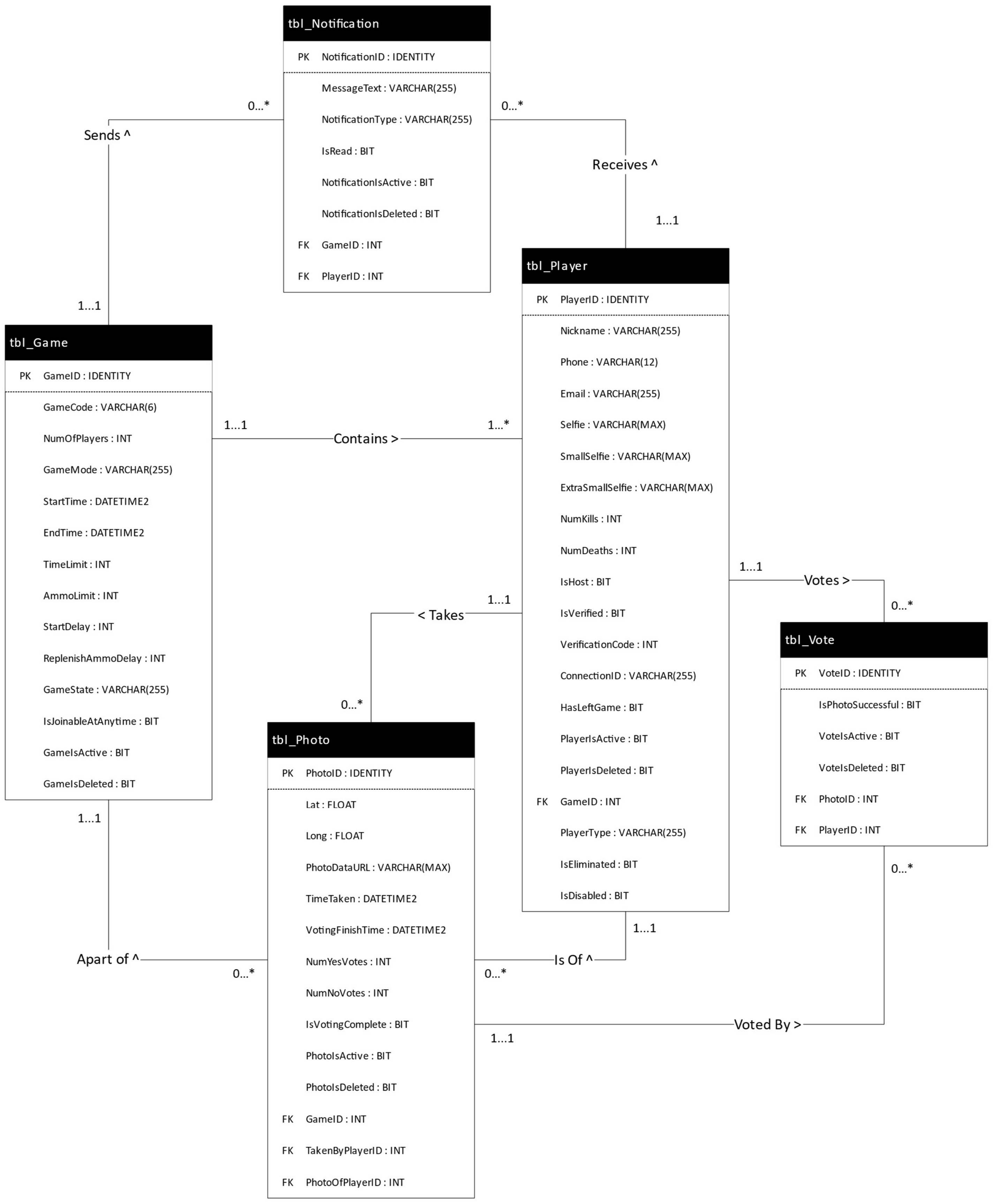
Can’t access the map or geolocation functionality:

- Ensure GPS functionality is turned on with the phone.
- Ensure GPS is enabled with the browser

Can’t join a game:

- Check to see if the room code is correct.
- Check to see if the maximum number of players has already been reached.
- Check to see if the game has been set to “Lobby join only” (in that players my only join the game if the game state is in the Lobby).

EER Diagram


tbl_Notification
PK NotificationID : IDENTITY
MessageText : VARCHAR(255)
NotificationType : VARCHAR(255)
IsRead : BIT
NotificationIsActive : BIT
NotificationIsDeleted : BIT
FK GameID : INT
FK PlayerID : INT
0...*
Sends ^
1...1
0...*
Receives ^
1...1
tbl_Player
PK PlayerID : IDENTITY
Nickname : VARCHAR(255)
Phone : VARCHAR(12)
Email : VARCHAR(255)
Selfie : VARCHAR(MAX)
SmallSelfie : VARCHAR(MAX)
ExtraSmallSelfie : VARCHAR(MAX)
NumKills : INT
NumDeaths : INT
IsHost : BIT
IsVerified : BIT
VerificationCode : INT
ConnectionID : VARCHAR(255)
HasLeftGame : BIT
PlayerIsActive : BIT
PlayerIsDeleted : BIT
FK GameID : INT
PlayerType : VARCHAR(255)
IsEliminated : BIT
IsDisabled : BIT
tbl_Game
PK GameID : IDENTITY
GameCode : VARCHAR(6)
NumOfPlayers : INT
GameMode : VARCHAR(255)
StartTime : DATETIME2
EndTime : DATETIME2
TimeLimit : INT
AmmoLimit : INT
StartDelay : INT
ReplenishAmmoDelay : INT
GameState : VARCHAR(255)
IsJoinableAtAnytime : BIT
GameIsActive : BIT
GameIsDeleted : BIT
1...1
Contains >
1...*
1...1
Votes >
0...*
1...1
< Takes
0...*
tbl_Vote
PK VoteID : IDENTITY
IsPhotoSuccessful : BIT
VoteIsActive : BIT
VoteIsDeleted : BIT
FK PhotoID : INT
FK PlayerID : INT
tbl_Photo
PK PhotoID : IDENTITY
Lat : FLOAT
Long : FLOAT
PhotoDataURL : VARCHAR(MAX)
TimeTaken : DATETIME2
VotingFinishTime : DATETIME2
NumYesVotes : INT
NumNoVotes : INT
IsVotingComplete : BIT
PhotoIsActive : BIT
PhotoIsDeleted : BIT
FK GameID : INT
FK TakenByPlayerID : INT
FK PhotoOfPlayerID : INT
1...1
Apart of ^
0...*
0...*
Is Of ^
1...1
0...*
Voted By >
1...1

Data Dictionary

tbl_Game: A game of CamTag.

Attribute	Description	Data Type	Default	Null
GameID {PK}	Primary Key, auto incrementing.	INT	NA	N
GameCode {AK}	The random password / code for the game which players must enter to join the game. Used to stop random players entering a game. This code will be given to the host of the game, and the host will pass on the code to other players to join.	VARCHAR(6)	ABC123	N
NumOfPlayers	The number of players who have joined the game. Includes players who are not verified.	INT	0	N
GameMode	The type of game being played: CORE = The core game of tag, time limit based. BR = Battle Royale	VARCHAR(255)	“CORE”	N
StartTime	The time the game starts. This will be the time when the game is moved into a “PLAYING” state, and the players can now take photos and play the game.	DATETIME2	GETDATE() + 10 mins	N
EndTime	The time the game will end. This is the time the game will be completed.	DATETIME2	24hrs after start date	N
TimeLimit	The time limit of the game. Stored in milliseconds.	INT	86400000	N
AmmoLimit	The maximum ammo (number of photos) each player can have at one time.	INT	3	N
StartDelay	The time delay which the game transitions from STARTING to PLAYING to let players run away and hide before the game begins. Stored in milliseconds.	INT	600000	N
ReplenishAmmoDelay	The time delay which a player’s ammo will be replenished after taking a photo. Stored in milliseconds.	INT	600000	N
GameState	The current state of the game, outlines if the game is playing or completed etc. IN LOBBY = The game is currently in the lobby, waiting for players to join. STARTING = The game is currently starting and will begin once the StartTime has elapsed. PLAYING = The game is currently playing, and players can take photos of others. COMPLETED = The game is completed.	VARCHAR(255)	IN LOBBY	N
IsJoinableAtAnyTime	A flag value which indicates if the game can be joined at any time. If yes, this allows players to join the game while the game is STARTING or PLAYING. If no, players can only join the game when the GameState is IN LOBBY.	BIT	0	N

GameIsActive	A flag value which indicates this record is apart of a game which is not completed and still playing.	BIT	1	N
GameIsDeleted	A flag value which indicates if this record is currently deleted and should be discarded.	BIT	0	N

tbl_Game Constraints

Attribute	Constraint
GameCode	Game code length must be 6.
NumOfPlayers	A maximum number of players is 16.
GameMode	Can only be CORE or BR.
StartTime	Must be less the EndTime.
GameState	Can only be IN LOBBY, STARTING, PLAYING or COMPLETED.
TimeLimit	The minimum time limit is 10 minutes and the maximum time limit is 24 hours.
AmmoLimit	Minimum is 1, the maximum is 9.
ReplenishAmmoDelay	Minimum is 1 minute, the maximum is 1 hour.
StartDelay	Minimum is 1 minute, maximum is 10 minutes.

tbl_Photo: A photo posted by a player in a game of CamTag.

Attribute	Description	Data type	Default	Null
PhotoID {PK}	Primary Key, auto incrementing.	INT	NA	N
Lat	The latitude the photo was captured at.	FLOAT	NA	Y
Long	The longitude the photo was captured at.	FLOAT	NA	Y
PhotoDataURL	The base64 dataURL of the photo.	VARCHAR(MAX)	NA	N
TimeTaken	The time the image was taken.	DATETIME2	GETDATE()	N
VotingFinishTime	The time the peer review voting will end. The default is 15mins from the TimeTaken.	DATETIME2	15mins after	N
NumYesVotes	The number of YES votes from other players in the game that agree that the photo submitted is a successful photo the PhotoOfPlayerID.	INT	0	N
NumNoVotes	The number of NO votes from other players in the game that DO NOT agree that the photo submitted is a successful photo the PhotoOfPlayerID.	INT	0	N
IsVotingComplete	A flag value which indicates if the voting has been completed. Either all other players have successfully made a vote or the time has passed the finished time.	BIT	0	N
PhotoIsActive	A flag value which indicates this record is apart of a game which is not completed and still playing.	BIT	1	N
Photo IsDeleted	A flag value which indicates if this record is currently deleted and should be discarded.	BIT	0	N
GameID {FK}	The game which the photo is apart of.	INT	NA	N
TakenByPlayerID {FK}	The ID of the player who took the photo.	INT	NA	N
PhotoOfPlayerID {FK}	The ID of the player who this picture is of.	INT	NA	N

tbl_Photo Constraints

Attribute	Constraint
VotingFinishTime	Must be greater than TimeTaken.
NumYesVotes	Must be greater than 0.
NumNoVotes	Must be greater than 0.

tbl_Player: A player inside a game of CamTag.

Attribute	Description	Data Type	Default	Null
PlayerID {PK}	Primary Key, auto incrementing.	INT	NA	N
Nickname	The nickname the player provides themselves to distinguish themselves in a game.	VARCHAR(255)	“Player”	N
Phone	The phone number of the player. This is the phone number where notifications will be sent to. Can be null because the player can also use an email address to receive notifications.	VARCHAR(12)	NA	Y
Email	The email address of the player. The email address where notifications will be sent to. Can be null because the player can also use a phone number to receive notification.	VARCHAR(255)	NA	Y
Selfie	The base64 dataURL of the selfie photo. Size of the photo is 512x512px	VARCHAR(MAX)	NA	N
SmallSelfie	A resized version of Seflie, still in base64 format. The size of the photo is 128x128px	VARCHAR(MAX)	NA	N
ExtraSmallSelfie	A resized version of Seflie, still in base64 format. The size of the photo is 32x32px.	VARCHAR(MAX)	NA	N
AmmoCount	The amount of ammo the player currently has in the game.	INT	3	N
NumKills	The number of successful kills or “Tags” a player has made in the game.	INT	0	N
NumDeaths	The number of times the player has been killed or “Tagged” by another player.	INT	0	N
IsHost	A flag value which indicates if this player is the host of the game they are playing. A host of a game is the player who started/created a game. Only the host player has permission to start a game and remove unverified players from their game.	BIT	0	N
IsVerified	A flag value which indicates if the players chosen phone number or email address has been verified successfully. A player verifies their contact details by entering a verification code.	BIT	0	N
VerificationCode	The code which the player must confirm in order to verify themselves and their contact details.	INT	NA	Y
ConnectionID	The connection ID produces by SignalR when a client connects to the SignalR hub. The connectID will be generated on the client side and sent to the SignalR hub which will then be stored in the DB	VARCHAR(255)	NULL	Y
HasLeftGame	A flag value which outlines if the player has left the game.	BIT	0	N
PlayerIsActive	A flag value which indicates this record is apart of a game which is not completed and still playing.	BIT	1	N
PlayerIsDeleted	A flag value which indicates if this record is currently deleted and should be discarded.	BIT	0	N
GameID {FK}	The game which the player is apart of.	INT	NA	N
PlayerType	The type of player, either a CORE player or a BR player, used for determining which kind of game they are playing.	VARCHAR(255)	CORE	N
IsEliminated	If the player is a BR player, this value indicates if the player has been eliminated from the game by another player successfully taking a photo of them.	BIT	0	N
IsDisabled	If the player is a BR player, this value indicates if the player has been disabled for attempting to take a photo outside of the playing zone.	BIT	0	N

tbl_Player Constraints

Attribute	Constraint
NumKills	Must be greater than or equal to 0.
NumDeaths	Must be greater than or equal to 0.
AmmoCount	Must be greater than or equal to 0
Email + Phone	The player must have an email address or phone number; both values cannot be null at the same time.

tbl_Vote: The table which solves the many to many relationship between a player and photo, where a player can vote on multiple photos and a photo can be voted on by multiple players.

Attribute	Description	Data Type	Default	Null
VoteID {PK}	Primary Key, auto incrementing.	INT	NA	N
IsPhotoSuccessful	A flag value which indicates the user's vote selection. 0 = No this photo is not valid. 1 = Yes the photo is valid.	BIT	0	N
VoteIsActive	A flag value which indicates this record is apart of a game which is not completed and still playing.	BIT	1	N
VoteIsDeleted	A flag value which indicates if this record is currently deleted and should be discarded.	BIT	0	N
PhotoID {FK}	The ID of the photo being voted on.	INT	NA	N
PlayerID {FK}	The ID of the player making the vote.	INT	NA	N

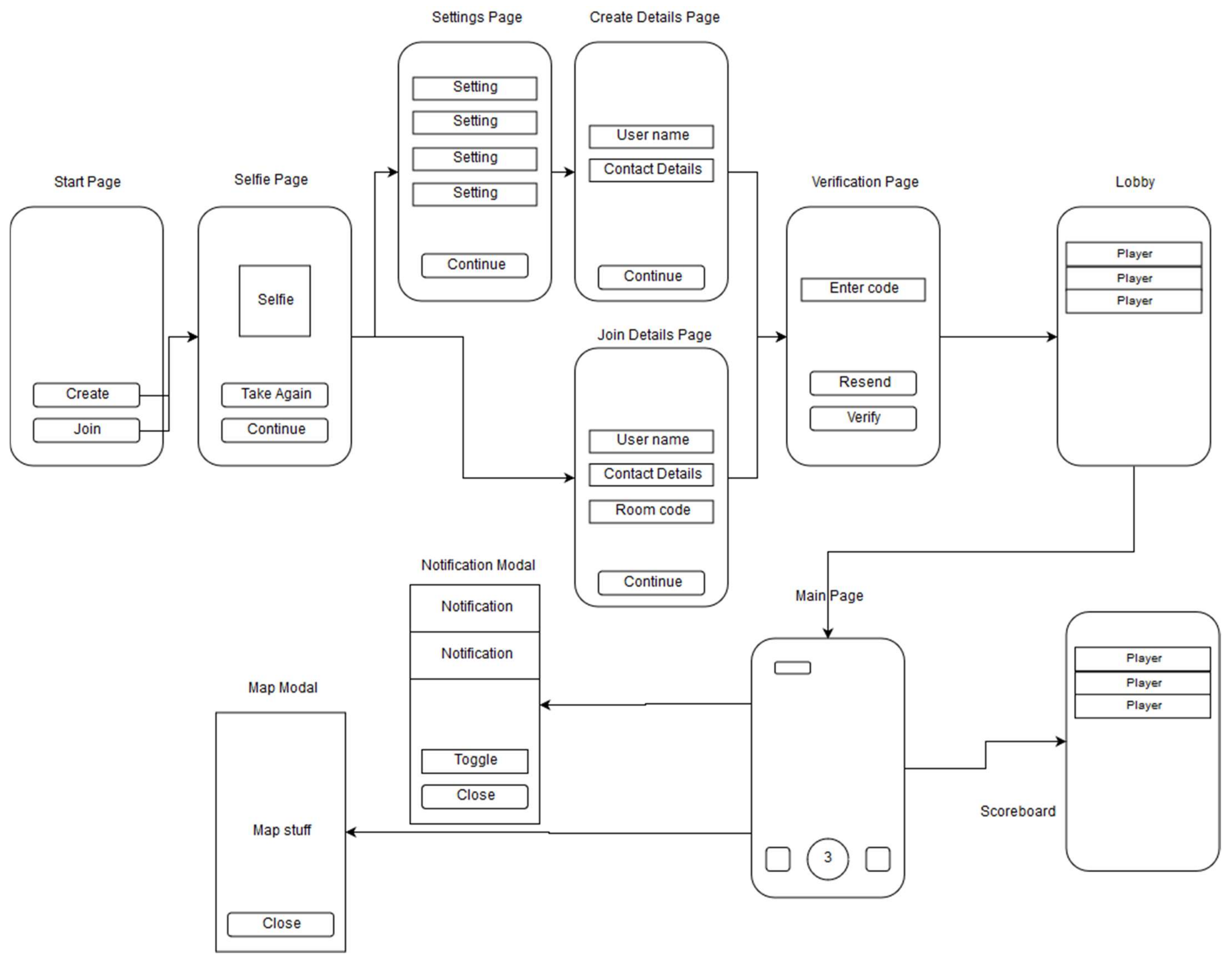
tbl_Notification: In-game notifications received by the user.

Attribute	Description	Data Type	Default	Null
NotificationID {PK}	Primary Key, auto incrementing.	INT	NA	N
MessageText	The text which will be displayed to the user when viewing the notification.	VARCHAR(255)	"Notification"	N
NotificationType	The type of notification such as: SUCCESS = Successfully tagged another player. FAIL = Unsuccessful attempt at tagging another player. JOIN = A new player joined the game. LEAVE = A player has left the game. AMMO = Ammo count has been replenished from empty. DISABLED = The BR player has been disabled for taking a photo outside the zone RE-ENABLED = The BR player has been re-enabled.	VARCHAR(255)	SUCCESS	N
IsRead	A flag value which outlines if the user has read the notification.	BIT	0	N
NotificationIsActive	A flag value which indicates this record is apart of a game which is not completed and still playing.	BIT	1	N
NotificationIsDeleted	A flag value which indicates if this record is currently deleted and should be discarded.	BIT	0	N
GameID {FK}	The ID which the notification is apart of.	INT	NA	N
PlayerID {FK}	The ID of the player who the notification is for.	INT	NA	N

tbl_Notification Constraints

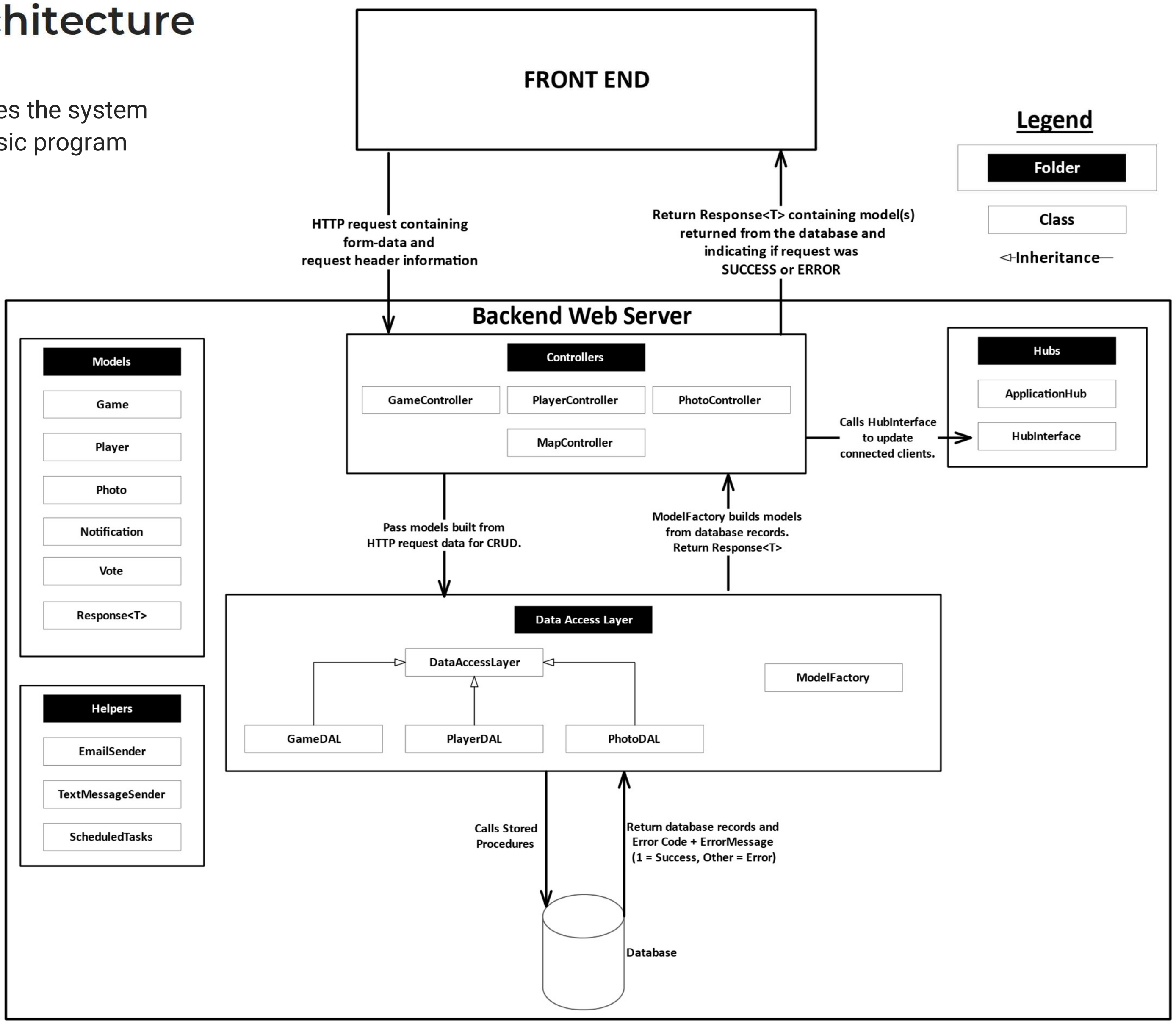
Attribute	Constraint
NotificationType	Can only be SUCCESS, FAIL, JOIN, LEAVE, AMMO, DISABLED or RE-ENABLED.

UI Storyboard



System Architecture

This diagram outlines the system architecture and basic program flow for a request.



FRONT END
HTTP request containing form-data and request header information
Return Response<T> containing model(s) returned from the database and indicating if request was SUCCESS or ERROR
Legend
Folder
Class
◁Inheritance—
Backend Web Server
Models
Game
Player
Photo
Notification
Vote
Response<T>
Controllers
GameController
PlayerController
PhotoController
MapController
Calls HubInterface to update connected clients.
Hubs
ApplicationHub
HubInterface
Pass models built from HTTP request data for CRUD.
ModelFactory builds models from database records. Return Response<T>
Data Access Layer
DataAccessLayer
ModelFactory
GameDAL
PlayerDAL
PhotoDAL
Helpers
EmailSender
TextMessageSender
ScheduledTasks
Calls Stored Procedures
Return database records and Error Code + ErrorMessage (1 = Success, Other = Error)
Database

API Interface Guide

Get All Players In Game

Route	**api/game/getAllPlayersInGame/{id:int}/{isPlayerID:bool}/{filter}/{orderBy}**
HTTP	GET
Controller	GameController.cs
Description	Gets all the players in a game with multiple filter parameters. **FILTER VALUES:** • ***All*** = get all the players in the game which aren't deleted • ***Active*** = get all players in the game which aren't deleted and is active • ***Ingame*** = get all players in the game which aren't deleted, is active, have not left the game and have been verified. • ***Ingameall*** = get all players in the game which aren't deleted, is active, and have been verified(includes players who have left the game) • ***Taggable*** = get all players who are taggable. • **Host** = gets all players for the host lobby view, all players (including not verified) and players who have not left the game. **ORDER BY VALUES:** • ***AZ*** = Order by name in alphabetical order • ***ZA*** = Order by name in reverse alphabetical order • ***KILLS***= Order from highest to lowest in number of kills

Request Data

Data Location	Key/ID	Data Type
URL	/{id}	int
URL	/{isPlayerID}	bool
URL	/{filter}	string
URL	/{orderBy}	string

Return Data	*Response<Game>* - A Game object and a list of Players in that game. NULL if an error occurred.
Example Request	https://localhost:5000/api/game/getAllPlayersInGame/100000/true/INGAME/AZ

Example Response

```
{
   "data": {
      "gameID": 100000,
      "gameCode": "tcf124",
      "numOfPlayers": 4,
      "gameMode": "CORE",
      "startTime": "2018-09-23T19:01:34.4733333",
      "endTime": "2018-09-24T18:51:34.4733333",
      "gameState": "PLAYING",
        "isJoinableAtAnytime": false,
        "isActive": true,
        "isDeleted": false,
        "players": [
            {
                "playerID": 100004,
                "nickname": "David",
                "phone": "",
                "email": "team6.camtag@gmail.com",
                "selfieDataURL": "localhost",
                "ammoCount": 3,
                "numKills": 0,
                "numDeaths": 0,
                "numPhotosTaken": 0,
                "isHost": false,
                "isVerified": true,
                "isActive": true,
                "isDeleted": false,
                "connectionID": "",
                "hasLeftGame": false,
                "gameID": 100000,
                "game": null,
                "isConnected": false
            }
        ]
    },
    "type": "SUCCESS",
    "errorMessage": "",
    "errorCode": 1
}
```

Error Codes Returned

Error Code	Reason
DATABASE_CONNECT_ERROR	Error connection to database.
DATA_INVALID	Filter or order by is null / empty.
ITEM_DOES_NOT_EXIST	The game player list returned is empty.
BUILD_MODEL_ERROR	Error building the Game/Player objects.
PLAYER_DOES_NOT_EXIST	The ID passed in does not exist
GAME_DOES_NOT_EXIST	The ID passed in does not exist

Join Game

Route	**api/player/joinGame**
HTTP	POST
Controller	PlayerController.cs
Description	Joins a player to a game matching the gameCode value, creating a new player record and returning the created player object.

Request Data

Data Location	Key/ID	Data Type
Body - JSON	gameCode	string
Body - JSON	nickname	string
Body - JSON	contact	string
Body - JSON	imgUrl	string

Return Data	*Response<Player>* - Returns the player object created with the game data. NULL if an error occurred.

Example Request

https://localhost:5000/api/player/joinGame
Request Body (JSON Object):

```
{
   “imgUrl”: “data:image/jpeg;base64,/9j/...”,
   “gameCode”: “tcf124”,
   “nickname”: ”Bob”,
   “Contact”: “bob@gmail.com”
}
```

Example Response

```
{
   "data": {
      "playerID": 100002,
      "nickname": "Jono2696jhkh",
      "phone": "",
      "email": "jono@2.19",
      "selfieFilePath": "no selfie",
      "numKills": 0,
      "numDeaths": 0,
      "numPhotosTaken": 0,
      "isHost": false,
      "isVerified": false,
      "isActive": true,
      "connectionID": "",
      "isConnected": false,
      "game": {
         "gameID": 100000,
         "gameCode": "oqvzyn",
         "numOfPlayers": 2,
         "gameMode": "CORE",
         "startTime": "2018-09-06T09:45:57.27",
         "endTime": "2018-09-07T09:45:57.27",
         "gameState": "STARTING",
         "isJoinableAtAnytime": false,
         "isActive": true,
         "players": null
      }
   },
   "type": "SUCCESS",
   "errorMessage": "",
   "errorCode": 1
```

	}

Error Codes Returned

Error Code	Reason
DATABASE_CONNECT_ERROR	Error connection to database.
INSERT_ERROR	An error occurred creating the database record.
BUILD_MODEL_ERROR	Error is building the created Player model.
MODELINVALID_PLAYER	The player details passed in are invalid.
GAME_DOES_NOT_EXIST	The game code passed in does not exist / is not an active gamecode.
CANNOT_PERFORM_ACTION	The game is not joinable; the game is already playing and IsJoinable = false. Or the game is full; there are already 16 players inside the game (both unverified and verified players).
ITEM_ALREADY_EXISTS	Nickname, phone or email is already taken.

Verify Player

Route	**api/player/verify**
HTTP	POST
Controller	PlayerController.cs
Description	Verifies a player record, the player enters their verification code received, and the code they entered is validated against the code stored in the database. If the code the player entered matches the code in the database, the player record is set to verified. Outlining that the player has access to the email or phone number entered when joining the game.
Request Data	Data Location / Key/ID / Data Type: **Form-data** / verificationCode / string **Header** / playerID / int
Return Data	*Response* - Success or Error
Example Request	https://localhost:5000/api/player/verify **Request Headers:** Key / Value **playerID** / 100000 **Request Body (Form-Data):** Key / Value **verificationCode** / 35579

Example Response	{
	"data": null,
	"type": “SUCCESS”,
	"errorMessage": "",
	"errorCode": 1
	}

Error Codes Returned	see below

Error Code	Reason
DATABASE_CONNECT_ERROR	Error connection to database.
INSERT_ERROR	An error occurred updating the record.
MODELINVALID_PLAYER	The playerID passed in is invalid.
DATA_INVALID	The verification code is incorrect format. Not a number between 10000 - 99999
PLAYER_DOES_NOT_EXIST	The playerID does not exist within the database.
GAME_STATE_INVALID	The game is already completed.
CANNOT_PERFORM_ACTION	The player is already verified.
ITEM_DOES_NOT_EXIST	The verification code is incorrect.

Resend Verification Code

Route	**api/player/resend**
HTTP	POST
Controller	PlayerController.cs
Description	Generates a new verification code for the player, updates the verification code in the database and resends the new code to the player's contact information (Email or Phone).
Request Data	Data Location / Key/ID / Data Type: **Header** / playerID / int
Return Data	*Response* - Success or Error
Example Request	https://localhost:5000/api/player/resend **Request Headers:** Key / Value: **playerID** / 100000
Example Response	{"type": "SUCCESS", "errorMessage": "", "errorCode": 1}
Error Codes Returned	See table below

Request Data:

Data Location	Key/ID	Data Type
Header	playerID	int

Request Headers:

Key	Value
playerID	100000

Error Codes Returned:

Error Code	Reason
DATABASE_CONNECT_ERROR	Error connection to database.
INSERT_ERROR	An error occurred updating the record.
MODELINVALID_PLAYER	The playerID passed in is invalid.
BUILD_MODEL_ERROR	Error building the created Player model.
PLAYER_DOES_NOT_EXIST	The playerID does not exist within the database.
GAME_STATE_INVALID	The game is already completed.
CANNOT_PERFORM_ACTION	The player is already verified.

Create Game

Route	**api/game/createGame**
HTTP	POST
Controller	GameController.cs
Description	Creates a new game according to player settings and joins the player to that game lobby, returning the created Player object. **Settings:** timeLimit: How long the game plays - Minimum = 10min, Maximum = 24hrs ammoLimit - How much ammo each player has - Minimum = 1, Maximum = 9 startDelay - How long before the game begins - Minimum = 1min, Maximum = 10min replenishAmmoDelay - How long before ammo replenished - Minimum = 1min, Max = 1 hour
Request Data	**NOTE:** all INT values are millisecond values.

Data Location	Key/ID	Data Type
Body - JSON	nickname	string
Body - JSON	contact	string
Body - JSON	imgUrl	string
Body - JSON	timeLimit	int
Body - JSON	ammoLimit	int
Body - JSON	startDelay	int
Body - JSON	replenishAmmoDelay	int
Body - JSON	gameMode	string
Body - JSON	isJoinableAtAnyTime	bool
Body - JSON	latitude	double
Body - JSON	longitude	double
Body - JSON	radius	int

Return Data	*Response<Player>* - Returns the player object created with the game data. NULL if an error occurred.

Example Request

Request Body (JSON Object):
```
{
  "imgUrl": "data:image/jpeg;base64,",
  "nickname": "JonoWilliams",
  "contact": "0457558322",
  "timeLimit": 86400000,
  "ammoLimit": 3,
  "startDelay": 60000,
  "replenishAmmoDelay": 60000,
  "gameMode": "BR",
  "isJoinableAtAnyTime": true,
  "latitude": 100,
  "longitude": 150,
  "radius": 50
}
```

Example Response
```
{
  "data": {
    "playerID": 100006,
    "nickname": "JonoWilliams",
    "phone": "+61457558322",
    "email": null,
    "selfieDataURL": "data:image/jpeg;base64,",
    "ammoCount": 3,
    "numKills": 0,
    "numDeaths": 0,
    "numPhotosTaken": 0,
    "isConnected": false,
    "gameID": 100001,
    "isHost": true,
    "isVerified": false,
    "isActive": true,
    "isDeleted": false,
    "connectionID": "",
    "hasLeftGame": false,
    "game": {
        "gameID": 100001,
        "gameCode": "uf27j3",
        "numOfPlayers": 1,
        "gameMode": "CORE",
        "gameState": "IN LOBBY",
        "timeLimit": 86400000,
        "ammoLimit": 3,
        "replenishAmmoDelay": 600000,
        "startDelay": 600000,
        "startTime": null,
        "endTime": null,
        "isJoinableAtAnytime": true,
        "isActive": true,
        "isDeleted": false,
        "players": null
    }
  },
  "type": "SUCCESS",
  "errorMessage": "",
  "errorCode": 1
}
```

Error Codes Returned

Error Code	Reason
DATABASE_CONNECT_ERROR	Error connection to database.
INSERT_ERROR	An error occurred creating the database record.
BUILD_MODEL_ERROR	Error building the created Player model.
MODELINVALID_PLAYER	The player details passed in are invalid.
ITEM_ALREADY_EXISTS	Nickname, phone or email is already taken.

Get Notifications

Route	**api/player/getNotifications/{playerID:int}/{all:bool}**
HTTP	GET
Controller	PlayerController.cs
Description	Returns a list of either unread or all notifications for the respective playerID, based on the ‘all’ boolean value.

Request Data

Data Location	Key/ID	Data Type
URL	/{playerID}	String
URL	/{all}	Bool

Return Data	*Response<List<Notification>>* - Returns a notification list. NULL if an error occurred.
Example Request	https://localhost:5000/api/player/getNotifications/100000/false

Example Response

```
{
        "data": [
        {
        "notificationID": 100000,
        "messageText": "'ScoMo' has joined the game.",
        "type": "JOIN        ",
        "isRead": false,
        "isActive": true,
        "gameID": 100000,
        "playerID": 100000
        }
        ],
        "type": "SUCCESS",
        "errorMessage": "",
        "errorCode": 1
}
```

Error Codes Returned

Error Code	Reason
DATABASE_CONNECT_ERROR	Error connection to database.
MODELINVALID_PLAYER	The playerID passed in is invalid.
BUILD_MODEL_ERROR	Error building the notification list model.
PLAYER_DOES_NOT_EXIST	The ID passed in does not exist in the DB
PLAYER_INVALID	The ID passed in is not inside an active game.
GAME_STATE_INVALID	The game is not in a PLAYING state.

Upload Photo

Route	**api/photo/upload**
HTTP	POST
Controller	PhotoController.cs
Description	Uploads a photo to the database. Sends out notifications to players that a photo must now be voted on. Returns a response which indicates success or error. NULL data is returned.

Request Data

Data Location	Key/ID	Data Type
Body - JSON	imgUrl	String
Body - JSON	takenByID	String
Body - JSON	photoOfID	String
Body - JSON	latitude	String
Body - JSON	longitude	String

Return Data	*Response* - Returns a response which indicates success or error. NULL data is returned.

Example Request

https://localhost:5000/api/photo/upload

Request Body (JSON Object):

```
{
   “imgUrl”: “data:image/jpeg;base64,/9j/...”,
   “takenByID”: “100000”,
   “photoOfID”: ”100001”,
   “latitude”: “123.456”,
   “longitude”: “-35.1”
}
```

Example Response	{"type": "SUCCESS", "errorMessage": "", "errorCode": 1}

Error Codes Returned

Error Code	Reason
DATABASE_CONNECT_ERROR	Error connection to database.
INSERT_ERROR	An error occurred creating the database record.
MODELINVALID_PHOTO	The photo details passed in are invalid.
PLAYER_DOES_NOT_EXIST	TakenByID or PhotoOfID is not inside DB
PLAYER_INVALID	TakenByID or PhotoOfID is not inside an active game.
DATA_INVALID	TakenByID and PhotoOfID are not in the same game.
GAME_STATE_INVALID	The game is not in a PLAYING state.
BUILD_MODEL_ERROR	An error trying to build the photo model.

Get Votes To Complete

Route	**api/photo/vote**
HTTP	GET
Controller	PhotoController.cs
Description	Gets the list of PlayerVotePhoto records which have not been completed by the player. This is the list of photos which the player has not voted on yet.
Request Data	Data Location / Key/ID / Data Type: **Header** / playerID / int
Return Data	*Response<List<PlayerVotePhoto>>* - Returns a list of the PlayerVotePhoto records, each record contains all information about the player, game, and photo. NULL if error occurred.
Example Request	https://localhost:5000/api/photo/vote **Request Headers:** Key / Value: **playerID** / 100000
Example Response	see below

Request Data:

Data Location	Key/ID	Data Type
Header	playerID	int

Request Headers:

Key	Value
playerID	100000

Example Response:

```
{
  "data": [
    {
      "voteID": 100003,
      "isPhotoSuccessful": null,
      "isActive": true,
      "playerID": 100000,
      "photoID": 100002,
      "player": {
        "playerID": 100000,
        "nickname": "Jono",
        "phone": "+61457558322",
        "email": "",
        "selfieFilePath": "localhost",
        "numKills": 0,
        "numDeaths": 0,
        "numPhotosTaken": 0,
        "isHost": false,
        "isVerified": true,
        "isActive": true,
        "connectionID": "B2nzXbZ1eCI7Vr93-PlJqQ",
        "isConnected": true,
        "hasLeftGame": false,
        "game": {
          "gameID": 100000,
          "gameCode": "tcf124",
          "numOfPlayers": 6,
          "gameMode": "CORE",
          "startTime": "2018-09-13T09:02:55.7366667",
          "endTime": "2018-09-14T09:02:55.7366667",
          "gameState": "STARTING",
          "isJoinableAtAnytime": false,
          "isActive": true,
          "players": null
        }
      },
  "photo": {
    "photoID": 100002,
    "lat": 23,
    "long": 21,
    "photoDataURL": base64 DataURL of Image Goes Here
    "timeTaken": "2018-09-13T09:15:22.0033333",
    "votingFinishTime": "2018-09-13T09:30:22.0033333",
    "numYesVotes": 0,
    "numNoVotes": 0,
    "isVotingComplete": false,
    "isActive": true,
    "gameID": 100000,
    "takenByPlayerID": 100003,
    "photoOfPlayerID": 100004,
    "game": {
      "gameID": 100000,
      "gameCode": "tcf124",
      "numOfPlayers": 6,
      "gameMode": "CORE",
      "startTime": "2018-09-13T09:02:55.7366667",
      "endTime": "2018-09-14T09:02:55.7366667",
      "gameState": "STARTING",
      "isJoinableAtAnytime": false,
      "isActive": true,
      "players": null
    },
    "takenByPlayer": {
      "playerID": 100003,
      "nickname": "Harry",
      "phone": "+61478542569",
      "email": "",
      "selfieFilePath": "localhost",
      "numKills": 0,
      "numDeaths": 0,
      "numPhotosTaken": 0,
      "isHost": false,
      "isVerified": false,
      "isActive": true,
      "connectionID": "",
      "isConnected": false,
      "hasLeftGame": false,
      "game": {
        "gameID": 100000,
        "gameCode": "tcf124",
        "numOfPlayers": 6,
        "gameMode": "CORE",
        "startTime": "2018-09-13T09:02:55.7366667",
        "endTime": "2018-09-14T09:02:55.7366667",
        "gameState": "STARTING",
        "isJoinableAtAnytime": false,
        "isActive": true,
        "players": null
      }
    },
    "photoOfPlayer": {
      "playerID": 100004,
      "nickname": "David",
      "phone": "+61478585269",
      "email": "",
      "selfieFilePath": "localhost",
      "numKills": 0,
      "numDeaths": 0,
      "numPhotosTaken": 0,
                    "isHost": false,
                    "isVerified": true,
                    "isActive": true,
                    "connectionID": "",
                    "isConnected": false,
                    "hasLeftGame": false,
                    "game": {
                        "gameID": 100000,
                        "gameCode": "tcf124",
                        "numOfPlayers": 6,
                        "gameMode": "CORE",
                        "startTime": "2018-09-13T09:02:55.7366667",
                        "endTime": "2018-09-14T09:02:55.7366667",
                        "gameState": "STARTING",
                        "isJoinableAtAnytime": false,
                        "isActive": true,
                        "players": null
                    }
                }
            }
        }
    ],
    "type": "SUCCESS",
    "errorMessage": "",
    "errorCode": 1
}
```

Error Codes Returned

Error Code	Reason
DATABASE_CONNECT_ERROR	Error connection to database.
MODELINVALID_PLAYER	The playerID passed in is invalid.
BUILD_MODEL_ERROR	Error building the vote list model.
PLAYER_DOES_NOT_EXIST	The ID passed in does not exist in the DB
PLAYER_INVALID	The ID passed in is not inside an active game.
GAME_STATE_INVALID	The game is not in a PLAYING state.

Vote On Photo

Route	**api/photo/vote**
HTTP	POST
Controller	PhotoController.cs
Description	Cast a vote on the photo. Vote that yes this photo is a successful photo or unsuccessful.

Request Data

Data Location	Key/ID	Data Type
Header	playerID	int
Header	voteID	int
form-data	decision	String (should only be “true” or “false”)

Return Data	*Response* - Success or Error

Example Request

https://localhost:5000/api/photo/vote

Request Headers:

Key	Value
playerID	100000
voteID	100001

Request Body (Form-data)

Key	Value
decision	“true”

Example Response	{"type": "SUCCESS", "errorMessage": "", "errorCode": 1}

Error Codes Returned

Error Code	Reason
DATABASE_CONNECT_ERROR	Error connection to database.
INSERT_ERROR	Error updating the record in DB.
MODELINVALID_VOTE	The playerID or voteID passed in is invalid.
PLAYER_DOES_NOT_EXIST	The ID passed in does not exist in the DB
PLAYER_INVALID	The ID passed in is not inside an active game.
GAME_STATE_INVALID	The game is not in a PLAYING state.
ITEM_DOES_NOT_EXIST	The voting record does not exist, the voteID does not exist or is already completed.
CANNOT_PERFORM_ACTION	Voting on the photo has already been completed.

Leave Game

Route	**api/player/leaveGame**
HTTP	POST
Controller	PlayerController.cs
Description	Leaves a player from the game via the passed playerID.

Request Data

Data Location	Key/ID	Data Type
Header	playerID	int

Return Data	*Response* - Success or Error

Example Request

https://localhost:5000/api/player/leaveGame

Request Headers:

Key	Value
playerID	100000

Example Response	{"type": "SUCCESS", "errorMessage": "", "errorCode": 1}

Error Codes Returned

Error Code	Reason
DATABASE_CONNECT_ERROR	Error connection to database.
INSERT_ERROR	Error updating the record in DB.
MODELINVALID_PLAYER	The playerID passed in is invalid.
BUILD_MODEL_ERROR	Error building the photo's completed after the player left model.
PLAYER_DOES_NOT_EXIST	The ID passed in does not exist in the DB
PLAYER_INVALID	The ID passed in is not inside an active game.
GAME_STATE_INVALID	The game is already completed.

Set Notifications as Read

Route	**api/player/setNotificationsRead**
HTTP	POST
Controller	PlayerController.cs
Description	Updates the database IsRead attribute for notifications that have been read by player.
Request Data	<table><tr><th>Data Location</th><th>Key/ID</th><th>Data Type</th></tr><tr><td>**Body - JSON**</td><td>playerID</td><td>int</td></tr><tr><td>**Body - JSON**</td><td>notificationArray</td><td>int</td></tr></table>
Return Data	*Response* - Success or Error

Example Request	https://localhost:5000/api/player/setNotificationsRead
	
	{
	"playerID":"100000",
	"notificationArray":["100002", "100003"]
	}

Example Response	{"type": "SUCCESS", "errorMessage": "", "errorCode": 1}
Error Codes Returned	<table><tr><th>Error Code</th><th>Reason</th></tr><tr><td>DATABASE_CONNECT_ERROR</td><td>Error connection to database.</td></tr><tr><td>INSERT_ERROR</td><td>Error updating the record in DB.</td></tr><tr><td>PLAYER_DOES_NOT_EXIST</td><td>The ID passed in does not exist in the DB</td></tr><tr><td>PLAYER_INVALID</td><td>The ID passed in is not inside an active game.</td></tr><tr><td>GAME_STATE_INVALID</td><td>The game the is not in a PLAYING state.</td></tr></table>

Use Ammo

Route	**api/player/useAmmo**
HTTP	POST
Controller	PlayerController.cs
Description	Decrements a Players ammo count. Schedules the ammo to be replenished after a certain period of time specified by the game rules.

Request Data

Data Location	Key/ID	Data Type
Header	playerID	int
form-data	latitude	double
form-data	longitude	double

Return Data	*Response<Player>* - The updated Player object after the ammo count has been reduced. NULL will be returned if an error occurred.

Example Request

https://localhost:5000/api/player/useAmmo

Request Headers:

Key	Value
playerID	100000
latitude	70
longitude	90

Example Response

```
{
    "data": {
        "playerID": 100012,
        "nickname": "Jonomundo26",
        "phone": "+61457558322",
        "email": null,
        "selfieDataURL": "data:image/jpeg;base64,",
        "ammoCount": 1,
        "numKills": 0,
        "numDeaths": 0,
        "numPhotosTaken": 2,
        "isConnected": false,
        "gameID": 100002,
        "isHost": true,
        "isVerified": true,
        "isActive": true,
        "isDeleted": false,
        "connectionID": "",
        "hasLeftGame": false,
        "isEliminated": false,
        "isDisabled": true,
        "playerType": "BR",
        "game": {
            "gameID": 100002,
            "gameCode": "56kop1",
            "numOfPlayers": 3,
            "gameMode": "BR",
            "gameState": "PLAYING",
            "timeLimit": 86400000,
            "ammoLimit": 3,
            "replenishAmmoDelay": 60000,
            "startDelay": 60000,
            "startTime": "2018-10-16T12:51:45.4333333",
            "endTime": "2018-10-16T14:51:45.4333333",
            "isJoinableAtAnytime": true,
            "isActive": true,
            "isDeleted": false,
            "latitude": 70,
            "longitude": 150,
            "radius": 50,
            "players": null
        }
    },
    "type": "ERROR",
    "errorMessage": "Not inside the zone.",
    "errorCode": 101
}
```

Error Codes Returned

Error Code	Reason
DATABASE_CONNECT_ERROR	Error connection to database.
INSERT_ERROR	Error updating the record in DB.
MODELINVALID_PLAYER	The playerID passed in is invalid.
BUILD_MODEL_ERROR	Error building the player model.
PLAYER_DOES_NOT_EXIST	The ID passed in does not exist in the DB
PLAYER_INVALID	The ID passed in is not inside an active game.
GAME_STATE_INVALID	The game is already completed.
CANNOT_PERFORM_ACTION	The ammo count is already at 0.
BR_NOTINZONE	The player is outside of the zone.

Get Ammo Count

Route	**api/player/ammo**
HTTP	GET
Controller	PlayerController.cs
Description	Gets the Players ammo count. Does not decrement the ammo, just gets the current ammo count value for the player in the Database.
Request Data	<table><tr><th>Data Location</th><th>Key/ID</th><th>Data Type</th></tr><tr><td>**Header**</td><td>playerID</td><td>int</td></tr></table>
Return Data	*Response<int>* - The ammo count for the Player. Negative int if an error occurs.
Example Request	https://localhost:5000/api/player/ammo **Request Headers:**<table><tr><th>Key</th><th>Value</th></tr><tr><td>**playerID**</td><td>100000</td></tr></table>

Example Response	{
	"data": 2,
	"type": "SUCCESS",
	"errorMessage": "",
	"errorCode": 1
	}

Error Codes Returned	<table><tr><th>Error Code</th><th>Reason</th></tr><tr><td>DATABASE_CONNECT_ERROR</td><td>Error connection to database.</td></tr><tr><td>MODELINVALID_PLAYER</td><td>The playerID passed in is invalid.</td></tr><tr><td>PLAYER_DOES_NOT_EXIST</td><td>The ID passed in does not exist in the DB</td></tr><tr><td>PLAYER_INVALID</td><td>The ID passed in is not inside an active game.</td></tr></table>

Begin Game

Route	**api/game/begin**
HTTP	POST
Controller	GameController.cs
Description	Begins a game, updates the GameStartTime to 10 minutes in the future to allow the players to disperse. It will also set the GameEndTime to be the end time specified during the game setup. This method also schedules threads to run which will update the game state to PLAYING and also schedule a thread to COMPLETE the game.

Request Data

Data Location	Key/ID	Data Type
Header	playerID	int

Return Data

Response<Game> - The updated Game object after been updated in the database. NULL if an error occurred

Example Request

https://localhost:5000/api/game/begin

Request Headers:

Key	Value
playerID	100000

Example Response

```
{
   "data": {
      "gameID": 100000,
      "gameCode": "tcf124",
      "numOfPlayers": 4,
      "gameMode": "CORE",
      "startTime": "2018-09-19T13:12:39.0433333",
      "endTime": "2018-09-20T13:02:39.0433333",
      "gameState": "STARTING",
      "isJoinableAtAnytime": false,
      "isActive": true,
      "isDeleted": false,
      "players": null
   },
   "type": "SUCCESS",
   "errorMessage": "",
   "errorCode": 1
}
```

Error Codes Returned

Error Code	Reason
DATABASE_CONNECT_ERROR	Error connection to database.
INSERT_ERROR	Error updating the record in DB.
MODELINVALID_PLAYER	The playerID passed in is invalid.
BUILD_MODEL_ERROR	Error building the player model.
PLAYER_DOES_NOT_EXIST	The ID passed in does not exist in the DB
PLAYER_INVALID	The ID passed in is not inside an active game.
DATA_INVALID	The playerID passed in is not the host player.
GAME_STATE_INVALID	The game is not IN LOBBY state.
CANNOT_PERFORM_ACTION	Not enough players to begin the game.

Map / Get Last Known Locations

Route	**api/map**
HTTP	GET
Controller	MapController.cs
Description	Gets all the latest photos for each user in the game. Each photo is the last one the user has taken. If the game is a battle royale game, the current radius is also calculated

Request Data

Data Location	Key/ID	Data Type
Header	playerID	int

Return Data	*Response<MapResponse>* - Returns a list of photos and the data regarding the latitude, longitude and current radius if the game is a battle royale.
Example Request	https://localhost:5000/api/map/getLastPhotoLocations/ + playerID,

Request Headers:

Key	Value
playerID	100007

Example Response

```
{
   "data": {
      "photos": [],
      "isBR": true,
      "radius": 36.72211447916667,
      "latitude": 70,
      "longitude": 150
   },
   "type": "SUCCESS",
   "errorMessage": "",
   "errorCode": 1
}
```

Error Codes Returned

Error Code	Reason
DATABASE_CONNECT_ERROR	Error connection to database.
MODELINVALID_PLAYER	The playerID passed in is invalid.
PLAYER_DOES_NOT_EXIST	The ID passed in does not exist in the DB
PLAYER_INVALID	The ID passed in is not in an active game.
GAME_STATE_INVALID	The game is not in a PLAYING state.
BUILD_MODEL_ERROR	Error build the photo list model.

Get Game Status / Application Status

Route	**api/game/status**
HTTP	GET
Controller	GameController.cs
Description	Get the current status of the game / web application. Used when a user reconnects back to the web application in order to the front end to be updated with the current game / application state so the front end can redirect the user accordingly.
Request Data	Data Location | Key/ID | Data Type **Header** | playerID | int
Return Data	*Response<GameStatusResponse>* - Returns a GameStatusResponse object which outlines the current GameState, if the player has any votes to complete, if the player has any new notifications and the updated player record. NULL if an error occurred. NOTE: Will only return the player record if the GameState is 'PLAYING', otherwise, the player will be NULL.
Example Request	https://localhost:5000/api/game/status Data Location | Key/ID | Data Type **Header** | playerID | int

Example Response	{
	  "data": {
	    "gameState": "PLAYING",
	    "hasVotesToComplete": true,
	    "hasNotifications": true,
	    "player": {
	      "playerID": 100000,
	      "nickname": "Jono",
	      "phone": "",
	      "email": "team6.camtag@gmail.com",
	      "selfieDataURL": "localhost",
	      "ammoCount": 3,
	      "numKills": 0,
	      "numDeaths": 0,
	      "numPhotosTaken": 0,
	      "isHost": true,
	      "isVerified": true,
	      "isActive": true,
	      "isDeleted": false,
	      "connectionID": "",
	      "isConnected": false,
	      "hasLeftGame": false,
	      "gameID": 100000,
	      "game": {
	        "gameID": 100000,
	        "gameCode": "tcf124",
	        "numOfPlayers": 4,
	        "gameMode": "CORE",
	        "startTime": null,
	        "endTime": null,
	        "gameState": "PLAYING",

```
                "isJoinableAtAnytime": false,
                "isActive": true,
                "isDeleted": false,
                "players": null
            }
        }
    },
    "type": "SUCCESS",
    "errorMessage": "",
    "errorCode": 1
}
```

Error Codes Returned

Error Code	Reason
DATABASE_CONNECT_ERROR	Error connection to database.
MODELINVALID_PLAYER	The playerID passed in is invalid.
PLAYER_DOES_NOT_EXIST	The ID passed in does not exist in the DB
BUILD_MODEL_ERROR	Error building the GameStatusResponse model.

Get Unread Notification Count

Route	**api/player/unread**
HTTP	GET
Controller	PlayerController.cs
Description	Gets the count of unread notifications for the player.
Request Data	

Data Location	Key/ID	Data Type
Header	playerID	int

Return Data	*Response<int>* - The count of unread notifications. Negative int if an error occurs.
Example Request	https://localhost:5000/api/player/unread **Request Headers:**

Key	Value
playerID	100000

Example Response

```
{
   "data": 2,
   "type": "SUCCESS",
   "errorMessage": "",
   "errorCode": 1
}
```

Error Codes Returned

Error Code	Reason
DATABASE_CONNECT_ERROR	Error connection to database.
MODELINVALID_PLAYER	The playerID passed in is invalid.
PLAYER_DOES_NOT_EXIST	The ID passed in does not exist in the DB
PLAYER_INVALID	The ID passed in is not inside an active game.
GAME_STATE_INVALID	The game the is not in a PLAYING state.

Remove Unverified Player

Route	**api/player/remove**
HTTP	POST
Controller	PlayerController.cs
Description	Removes the unverified player from the lobby / game. Can only be done while the GameState is IN LOBBY and can only be performed by the host player.
Request Data	Data Location / Key/ID / Data Type: **form-data** / playerID / int; **form-data** / playerIDToRemove / int
Return Data	*Response* - Success or Error
Example Request	https://localhost:5000/api/player/remove **Request Headers:** Key / Value: **playerID** / 100000; **playerIDToRemove** / 100003
Example Response	{"type": "SUCCESS", "errorMessage": "", "errorCode": 1}
Error Codes Returned	(see below)

Request Data:

Data Location	Key/ID	Data Type
form-data	playerID	int
form-data	playerIDToRemove	int

Request Headers:

Key	Value
playerID	100000
playerIDToRemove	100003

Error Codes Returned:

Error Code	Reason
DATABASE_CONNECT_ERROR	Error connection to database.
MODELINVALID_PLAYER	The playerID or playerIDToRemove passed in is invalid.
PLAYER_DOES_NOT_EXIST	One of the ID's passed in does not exist in the DB
PLAYER_INVALID	One of the ID's passed in is not active.
GAME_STATE_INVALID	The game the is not IN LOBBY state.
DATA_INVALID	The ID's are not in the same game.
CANNOT_PERFORM_ACTION	The playerID is not the host of the game.
ITEM_DOES_NOT_EXIST	The player to remove is already verified.

Error Codes Guide

Global Error Codes

The below error codes can be found across the entire web application; they are errors that can occur across many different application processes.

Error Code	Value	Description
DATABASE_CONNECT_ERROR	0	An error occurred while trying to connect to the database. An exception was thrown when trying to open the connection or run the stored procedure.
INSERT_ERROR	2	An error occurred while trying to INSERT or UPDATE a record inside the database. Possibly caused by a constraint on a database table resulting in the INSERT or UPDATE to not be completed.
BUILD_MODEL_ERROR	3	An exception was thrown while trying to parse the data returned from the database and build a strongly typed model class.
ITEM_ALREADY_EXISTS	4	The item you are trying to INSERT or UPDATE already exists, cannot insert or update a unique duplicate value.
DATA_INVALID	5	The data submitted is in an invalid format. EG expected and INT, but Parsing failed.
ITEM_DOES_NOT_EXIST	6	The given or returned item does not exist in the database.
CANNOT_PERFORM_ACTION	7	The attempted action is invalid or illegal.
GAME_DOES_NOT_EXIST	8	The GameCode or GameID passed in does not exist inside the database or is not a valid option/value in the request context.
GAME_STATE_INVALID	9	The current game state is invalid. The action trying to perform cannot be completed in the current game state.
PLAYER_DOES_NOT_EXIST	10	The PlayerID passed in does not exist inside the database.
PLAYER_INVALID	11	The PlayerID passed in is invalid in the request context. EG The player is not inside the game.
MODELINVALID_PLAYER	12	The Player model is invalid.
MODELINVALID_GAME	13	The Game model is invalid.
MODELINVALID_PHOTO	14	The Photo model is invalid.
MODELINVALID_VOTE	15	The Vote model is invalid.

Battle Royale Error Codes

Error Code	Value	Description
BR_NOTINZONE	101	The player requesting to use ammo / take a photo is outside of the playing zone in a battle royale game.