**Test Procedures**

**<ARMAMENT>**

****

**REVISION HISTORY**

|  |  |  |  |
| --- | --- | --- | --- |
| Revision # | Author | Revision Date | Comments |
| 1.0 | Keith Bosworth | March 15, 2019 | initiated |
| 1.1 | Keith Bosworth | March 16, 2019 | Skeleton of document sections, copied System Overview |
| 1.2 | Keith Bosworth | March 17, 2019 | Acceptance Tests Filled |
| 1.3 | Alex Cohn | March 17, 2019 | Acceptance Tests Reviewed |
| 1.4 | Alex Cohn | March 17, 2019 | Unit Tests |
| 1.5 | Andrew Pitt, Jeremy Scott | March 17, 2019 | Integration Tests |
| 1.6 | Ben Mankin | March 17. 2019 | Revisions & Edits |
| 2.0 | Alex Cohn | April 27, 2019 | Revisions & Edits |
| 2.1 | Keith Bosworth | April 28, 2019 | Add tests |
| 2.2 | Ben Mankin | April 28, 2019 | Add tests |
| 2.3 | Ben Mankin | April 29, 2019 | Accepted Revisions |

**Table of Contents**

[**System Overview**](#_30j0zll) **4**

[**Document Overview**](#_1fob9te) **9**

[**Unit Tests**](#_tyjcwt) **6**

[GameManager Class Methods](#_3dy6vkm) 6

[Private Methods](#_1t3h5sf) 6

[Gun Class Methods](#_4d34og8) 8

[Public Methods](#_2s8eyo1) 8

[Launcher Class Methods](#_3rdcrjn) 9

[Public Methods](#_26in1rg) 9

[PlayerManager Class Methods](#_lnxbz9) 9

[Public Methods](#_35nkun2) 10

[Private Methods](#_1ksv4uv) 13

[WallDropAnimator Class Methods](#_44sinio) 15

[Public Methods](#_2jxsxqh) 15

[**Integration Tests**](#_z337ya) **15**

[PhotonIntegrationScript Methods](#_3j2qqm3) 15

[PlayfabIntegrationScript Methods](#_1y810tw) 15

[**Use Case Integrations**](#_4i7ojhp) **16**

[**Acceptance Tests**](#_2xcytpi) **17**

[**References**](#_147n2zr) **47**

## System Overview

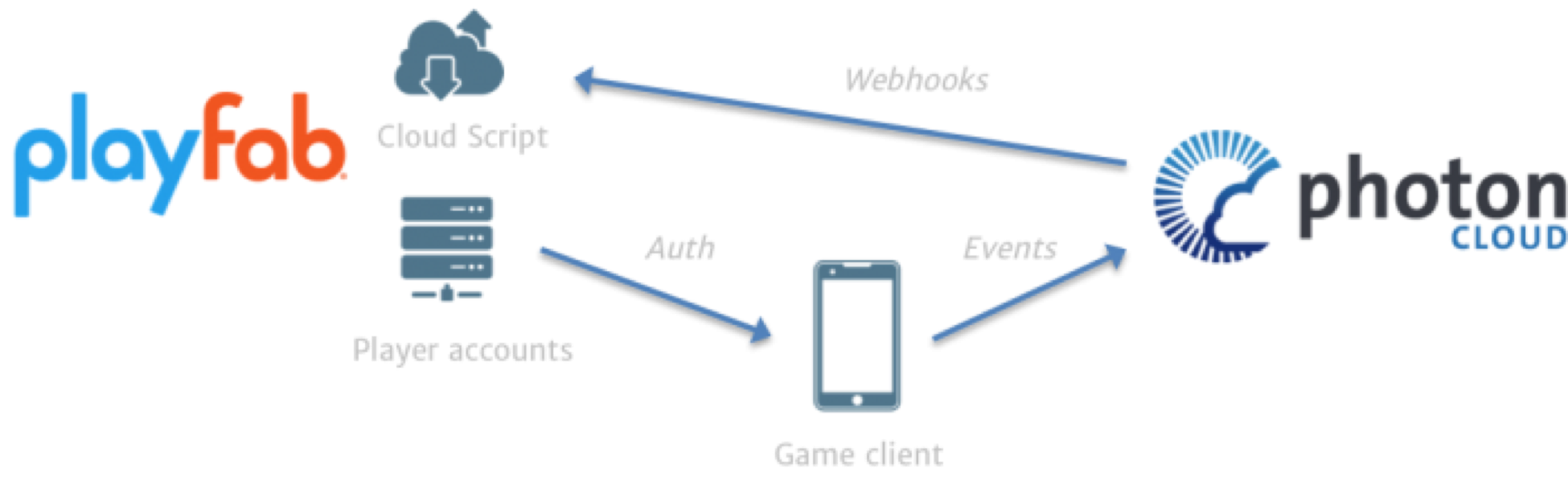
This section provides an overview of the Armament project: an original, networked, multiplayer, team-based video game built for PC, Mac, iOS, and Android platforms. Armament is built with the Unity game engine and API, Photon Networking servers and API, and PlayFab backend database and API.

Gameplay will consist of two stages played in succession: first, an *Armament* stage where players gather weapons and resources, and subsequently, a *Battle* stage where combatants fight for control of the arena using the resources they’ve acquired. Original sounds and art will be created for the project in addition to existing assets.

Armament will be designed and driven with the Unity engine and API, which contains numerous scripts and libraries that provide abstraction for the low-level details of physics rendering, graphics processing, animation, A.I, platform-specific builds, and system analytics.,

Player information is stored in a database provided by PlayFab. Users can register accounts and authenticate from both PC and mobile devices. Once logged in, players will see the Launcher where they can choose to play a game, or they can check the statistics stored in the database, which may be accessed through the leaderboard. If they choose to play, their statistics during that game will be updated to the leaderboard. Players can also add friends through the Launcher, which will allow them to invite those friends to private games. This feature gives players the ability to stay in touch with players they enjoyed playing with.

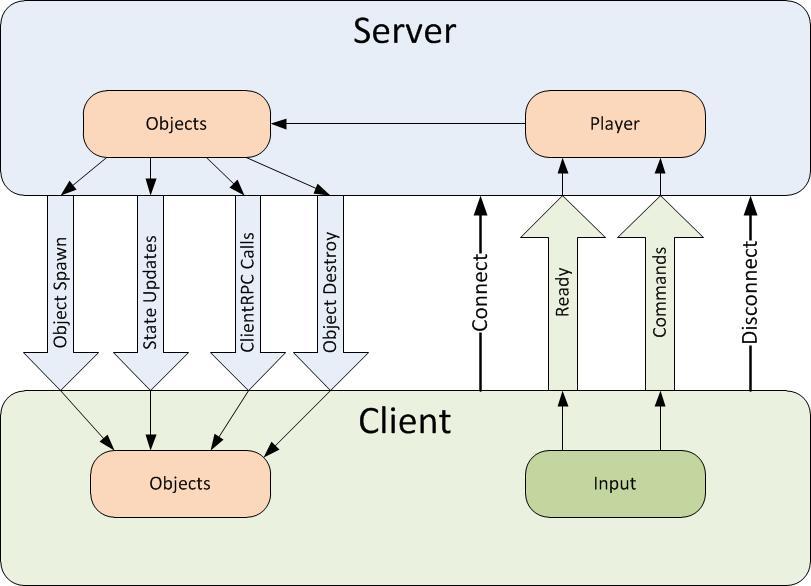
The interaction between Unity clients, the Photon cloud, and PlayFab can be seen below in the abstraction provided by **Figure 1.**

****

**Figure 1.** General overview of Unity client integration with Photon and PlayFab from a mobile perspective (similar for PC).

To play the role of the serving or host device, one client will assume the role of master client, which will synchronize remote clients by broadcasting RPC through Photon servers. A master client is a specialized type of local client that synchronizes remote clients using Photon servers as a means of communication, which sends and receives changes in specific game state based on client input. This implementation is headless in that it only passes RPC calls and choice data through Photon servers, as opposed to processing graphics and physics data within the Photon server space.

In order for Armament to synchronize across every connected device, copies of each object created in the game will exist in memory on both the master and remote clients. The master client device assumes the role of the authority to keep track of changes made to various GameObjects, as well as communicating those changes to the clients. The client/server relationship, RPC calls, data flow, GameObject storage, and state changes can be seen in **figure 2** below:



**Figure 2.** Overall data flow, object storage, and state changes from server (master client) to remote and local clients.

Data will be broadcasted from client to client using RCP and other modes of communication and data synchronization via the Photon server, and through the internet via UDP and TCP. The Photon networking framework provides a robust networking API built specifically for Unity projects in order to meet networking requirements. Clients send input, which is received by listening to various RPC events (for example OnMouseClick() called from within an RPC wrapper to listen to mouse clicks supplied from user input). The Photon Networking API provides abstraction of low-level socket code. **Figure 3** describes the user flow to begin playing online with other players.



**Figure 3.** User flow to begin playing Armament

When joining a game, players must use the in-game menu to connect to a Photon *name server*, which gives them access to a *master server*. Master servers are geographically located around the globe to provide low ping times to all clients, regardless of their location. Master clients will then place the clients in a *master server for matchmaking purposes.* . When a player finds a match they will communicate with other clients via a Game Server that is responsible for hosting the game room they are in.

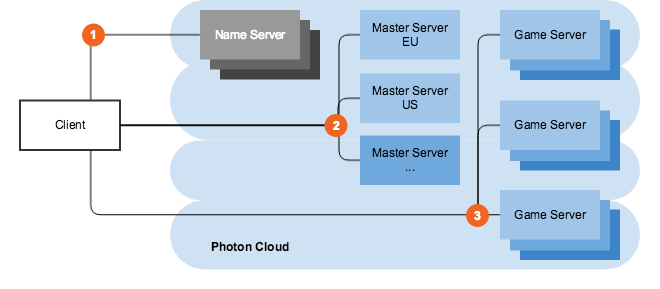
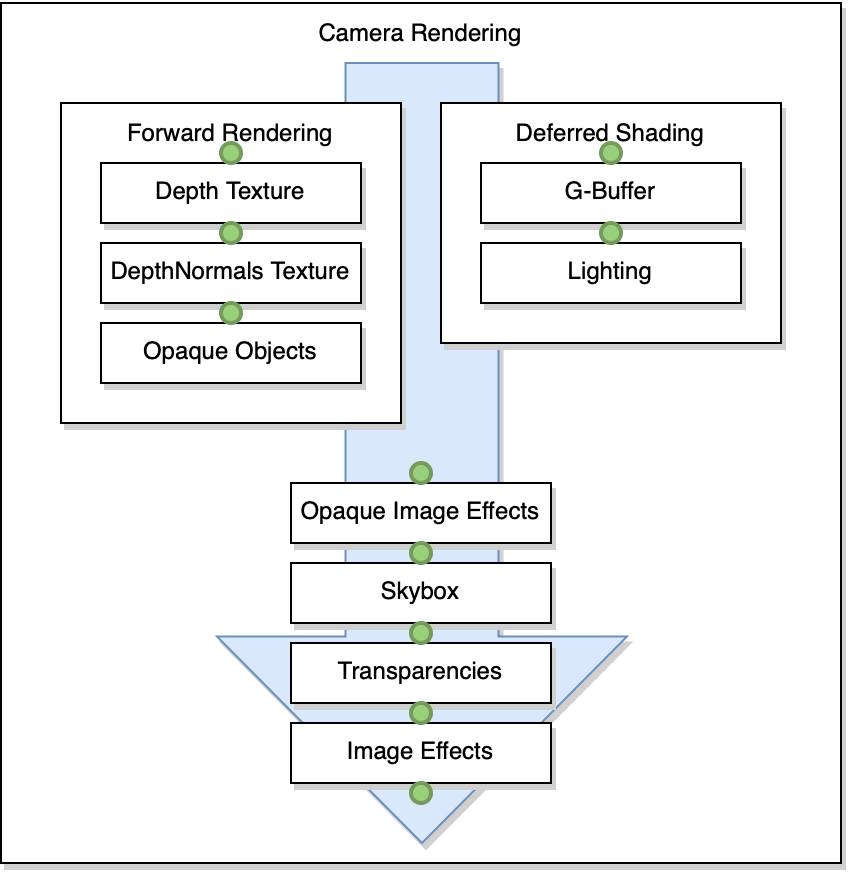


Figure 4.Master server and game server layout..

In the case that the host connection quality terminates or becomes suboptimal, the Photon API offers a host migration service which is called in order to move server identity to the next available device. Sending RPC through the Photon server adds one layer of security for clients in that the IPs of remote clients are managed by the Photon server instead of seen directly by the master.

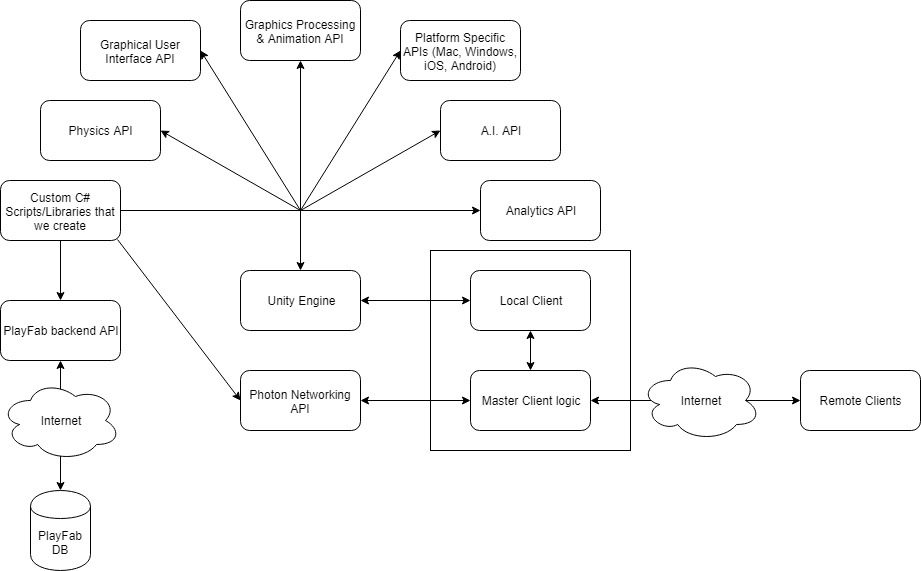
## The onus of graphics processing is placed on the client devices, which may flex the master client’s processing capabilities somewhat, but overall is not an impediment considering how picky our master client is in terms of choosing what to broadcast to clients through Photon servers. On the client side, the Unity engine will render a camera placed in the virtual environment, resulting in the data flow represented in figure 5 below. After graphics processing, certain state changes to the GUI will be communicated as attributes via various synchronization mechanisms provided by photon to all clients over the Internet.



**Figure 5**. Camera rendering data flow

## During gameplay, players have the option to toggle an A.I. controller. When this controller activated the game automatically takes control over the player avatar’s movement and actions. If the A.I. controller is activated during the *Armament* stage, the A.I. player will target (i.e., go to) known gun spawn points in order to pick up guns. Along the way, the A.I. may recognize that a gun, which it is not currently targeting for pickup and not yet picked up by another player, has come into view. . When it sees a valid gun target, it will run towards it and attempt to pick it up. During the Battle stage, the A.I. will wait for opponents (i.e., players on the other team) to come into view or shoot at it. Either event will trigger the A.I. player to target the opponent. The A.I. player will pursue its targeted opponent even if the player tries to run away. If the A.I. player gets the target in its crosshairs, it will immediately shoot..The A.I. player always calculates shortest path to its destination whether or not its target’s position changes. A high-level representation of the interlocking systems is described below in figure 4.

## System Block Diagram



**Figure 4.** A high-level view of the components in Armament.

## 

## Glossary

* **Master client**: the client that is designated to act as a pseudo server for all other clients. The master client becomes responsible for making decisions and coordinating actions that would typically be the responsibility of a server in a server-client model. Any client that joins a game room can potentially become the master client at some point. By default, the master client is chosen in the order of who entered the game room first.
* **Remote client**: all clients that are not currently the master client.
* Name server: the first server that every client contacts, which provides the list of available regions.
* **Master server**: every region has a completely separate master server for matchmaking.Game server: hosts game rooms
* **Launcher**: the first scene presented to the user upon starting the game. In this scene, a user has the ability to log in to their account, choose their gameplay options, and enter a game room to begin playing the game.
* **GameObject**:
* **Prefab**: a “prefabricated” set of GameObjects that are linked in a parent-child relationship. Prefabs have components that affect the way they behave (as it appears to the user) after graphics processing.
* **Components**: scripts which can be added to GameObjects in order to change the GameObject behavior, appearance, and properties. The Unity API contains certain fundamental components that can be added to a project. Developers also have the capability to write original scripts as components.

## Document Overview

The test procedure document outlines the various tests to be completed to verify the extent of completion for requirements applied to the various features developed for Armament. These tests will also be used to determine the stability of the overall system. The tests described in this document can be categorized at a high level into three areas:

Unit Tests

Automated tests applied to each method used in Armement. These tests automatically provide a certain input and require some well defined expected output in order to pass.

Integration Tests

Automated Tests designed against all use cases described for Armament (see Requirements Document). These tests are more specifically aimed at ensuring the defined use cases can be accomplished via our separate pieces of code, as well as ensuring these separate pieces integrate well together without breaking the system.

Acceptance Tests

A combination of automated and manual tests designed against requirements for features in Armament. Similar to unit tests because they prescribe some set of input and expected output that is considered passing. Acceptance tests specifically attempt to quantify whether specific requirements have been met by the system.

## 

## Unit Tests

### GameManager Class Methods

#### Private Methods

* void UpdateNavMesh()
  + Test 1:
    - Preconditions
      * navMesh is not null
    - Postconditions
      * navMesh is built
* void LoadArena()
  + Test 1:
    - Preconditions
      * client is Master Client
    - Postconditions
      * Arena has loaded
* void UpdatePlayerPropertiesDisplay()
  + Test 1:
    - Preconditions
      * Custom Room Properties and Player Properties are set
    - Postconditions
      * GUI Text reflects proper output of properties
* void RemoveUnclaimedItems()
  + Test 1:
    - Preconditions
      * Unclaimed items exist in the arena
      * Items are visible to players
    - Postconditions
      * Unclaimed items are no longer visible to players
      * Unclaimed items are turned into vanished items
* void ReturnVanishedItems()
  + Test 1:
    - Preconditions
      * Vanished items exist in the arena
      * Vanished items are not visible to players
    - Postconditions
      * Vanished items are visible to players
      * Vanished items are turned into unclaimed items
* void EndRound()
  + Test 1:
    - Postconditions
      * StartRound is called
* void DestroyAllItems()
  + Test 1:
    - Preconditions
      * Items are in the arena
    - Postconditions
      * No items are in the arena
* void BalanceTeams()
  + Test 1:
    - Preconditions
      * Teams are unbalanced by more than 1 player
    - Postconditions
      * Teams are balanced by no more than 1 player
* void StartRound()
  + Test 1:
    - Postconditions
      * Stage 1 timer is started
      * Position of team dividing wall is reset
      * Position of players is reset (to spawn points)
      * ReturnVanishedItems is called
* void SpawnNewItems()
  + Test 1:
    - Preconditions
      * Client is master client
    - Postconditions
      * New items are spawned
* void SpawnWall()
  + Test 1:
    - Preconditions
      * Client is master client and no wall is spawned
    - Postconditions
      * New wall is spawned
* GameObject InstantiatePlayerForActor()
  + Test 1:
    - Preconditions
      * Client does not have a locally controlled player
    - Postconditions
      * Client has a locally controlled player
* void RemoveGunOwnerships(Player player)
  + Test 1:
    - Preconditions
      * Player has gun ownership information in the Room.CustomProperties hashtable
    - Postconditions
      * Player has no gun ownership information in the Room.CustomProperties hashtable

### Gun Class Methods

#### Public Methods

* public void Shoot()
  + Test 1:
    - Preconditions
      * Gun is ready to shoot
      * Gun is aimed at a target
    - Postconditions
      * PlayGunShotSound is called
      * Gun causes target to take damage
      * impact effect has been instantiated locally at hit point
* public void PlayGunShotSound()
  + Test 1:
    - Post conditions
      * Gunshot sound is played
* public bool IsReadyToShoot()
  + Test 1:
    - Preconditions
      * Gun is owned by a player and is the active showing gun on the player avatar.
    - Postconditions
      * Gun cooldown timer has expired.
      * Boolean returned giving the status of the timer.
* public int GetTypeOfGun
  + Test 1:
    - Preconditions
      * Gun is instantiated in the scene.
    - Postconditions
      * The integer representing the type of the gun has been returned
      * The value is correct.

### FragGrenade Class Methods

#### Public Methods

* [PunRPC] void Explode()
  + Test 1:
    - Preconditions
      * Timer has reached 0 and the grenade is ready to explode.
    - Postconditions
      * Explosion has been spawned.
      * Damage and physics force has been applied to all targets within the explosion radius.
* [PunRPC] void DestroyRPC()
  + Test 1:
    - Preconditions
      * Grenade has just exploded, or round has just ended, and the grenade should be destroyed.
    - Postconditions
      * GameObject is no longer visible in arena.
      * GameObject is no longer present in the network.
* void Throw()
  + Test 1:
    - Preconditions
      * playerWhoOwnsThisGrenade is set to the MonoBehaviourPun that belongs to the player who owns the grenade, and thrown is false (the grenade has not yet been thrown).
    - Postconditions
      * Set thrown equal to true.
      * Physics force has been applied to the rigidbody, and the grenade is flying through the air.

### Medkit Class Methods

#### Public Methods

* **void RestoreHealth():**
  + Test 1:
    - Preconditions:
      * Player has pressed H on the keyboard, and PlayerManager.ProcessInputs() has received the input keycode.
      * Player does not have maximum health.
    - Postconditions:
      * Medkit spawned and deactivated very quickly, since it is not necessary to see the medkit box GameObject when using it.
      * Player has regained 30 health.
      * If player has more than the max health, set their health to the max health (this logic is handled in PlayerManager, but remains relevant enough to mention here).
* **[PunRPC] void DestroyRPC():**
  + Test 1:
    - Preconditions:
      * DestroyRPC has been called: either a player has picked up a medkit from the arena.
    - Postconditions:
      * This has checked whether or not the device is the master client. If it is the master client, it will destroy the GameObject, sending the RPC call to all remote clients.
* **void Use():**
  + Test 1:
    - Preconditions:
      * playerWhoOwnsThisMedkit that is set to the PlayerManager, which represents the owner of the medkit, is not null, and playerUsedMedkit is false.
    - Postconditions:
      * Set playerUsedMedkit equal to true, which sets the condition to precipitate player to regain 30 health.

### Launcher Class Methods

#### Public Methods

* public int GetAvatarSliderValueFromSlider()
  + Test 1:
    - Preconditions:
      * avatarSelectSlider != null
    - Returns:
      * value of avatarSelectSlider
* public void Connect()
  + Test 1:
    - PostConditions
      * isConnecting flag is set
      * progressLabel is updated
      * PhotonNetwork.ConnectUsingSettings is called
* public void JoinRandomRoom()
  + Test 1:
    - Preconditions
      * PhotonNetwork.IsConnected is true
    - Postconditions
      * progressLabel is updated
      * Connect is called
* public void JoinOrCreateRoom()
  + Test 1:
    - Postconditions
      * JoinOrCreateRoom(null) is called
* public void JoinOrCreateRoom(string[] expectedUsers)
  + Test 1:
    - Preconditions
      * PhotonNetwork.IsConnected is true
      * roomNameInputField.text is set
    - PostCondition
      * progressLabel is update
      * PhotonNetwork.JoinOrCreateRoom is called

### PlayerManager Class Methods

#### Public Methods

* public void CallShootRPC()
  + Test 1:
    - Preconditions:
      * Player has a gun (ActiveGun)
      * ActiveGun is ready to shoot
    - Postconditions:
      * PunRPC Shoot method is called over the network
* public void ToggleAIControl()
  + Test 1:
    - Preconditions:
      * Client has locally controlled player not under AI control
    - Postconditions
      * Client’s player is being controlled by AI script
      * Top Panel is blue to indicate AI control
  + Test 2:
    - Preconditions:
      * Client has locally controlled player under AI control
    - Postconditions:
      * Client’s player is under manual control
      * Top Panel is gray to indicate manual control
* public void PlaySound (AudioClip audioclip)
  + Test 1:
    - Input:
      * Audio clip to be played
    - Postconditions
      * Audio clip is played
* public void SetTeam(string team)
  + Test 1:
    - Input: team name
    - Postconditions
      * Player.SetCustomProperties is called
      * Player.CustomProperties is appropriately set
* public string GetTeam()
  + Test 1:
    - Preconditions:
      * Player exists and has information in Player.CustomProperties about what team it is on
    - Returns:
      * String representing the team the player is on
* public void TakeDamage(float amount)
  + Test 1:
    - Input: amount of damage to cause to the player
    - Preconditions
      * Player is controlled by this client
      * Health > 0
    - Postconditions
      * Health has decreased by amount
* public void TakeDamage(float amount, PlayerManager playerWhoCausedDamage)
  + Test 1:
    - Inputs:
      * Amount of damage to cause to the player
      * Player who caused the damage
    - Preconditions
      * Player is controlled by this client
      * Health > 0
      * Health < input amount
    - Postconditions
      * Health has decreased by amount
      * AddKill is called on player who caused damage
* public void RestoreHealth(float amount)
  + Test 1:
    - Inputs:
      * Amount of health to restore
    - Preconditions
      * Player’s health + amount <= MAX\_HEALTH
    - Postconditions
      * Player’s heatlh = Player’s health + amount
* public void ResetHealth()
  + Test 1:
    - Preconditions
      * Health < MAX\_HEALTH
    - Postconditions
      * Health equals MAX\_HEALTH
* public void ResetShield()
  + Test 1:
    - Preconditions:
      * Player’s shield != MAX\_SHIELD
    - Postconditions:
      * Player’s shield = MAX\_SHIELD
* public void MovePlayer(Transform t)
  + Test 1:
    - Input: transform to set for the player
    - Preconditions
      * Player’s transform does not equal input transform
    - Postconditions
      * Player’s transform equals the input transform
* public void Respawn()
  + Test 1:
    - Preconditions
      * Client owns player
    - Postconditions
      * ResetHealth is called
      * Player is moved to spawn point
* public void StartDeadSpectatorMode()
  + Test 1:
    - Preconditions
      * Locally controlled player is alive
      * Client views player’s camera
      * Controllers are enabled
    - Postconditions
      * Locally controlled player is dead
      * Client views split screen scene cameras
      * Controllers are disabled
* public void StopDeadSpectatorMode()
  + Test 1:
    - Preconditions
      * Locally controlled player is dead
      * Client views split screen scene cameras
      * Controllers are disabled
    - Postconditions
      * Locally controlled player is alive
      * Client views player’s camera
      * Controllers are enabled
* public void SetActiveGun(int gunViewID)
  + Test 1:
    - Input: Photon View ID of the Gun to set as active gun
    - PostConditions
      * (Gun)PlayerManager.activeGun equals Gun with Photon View ID equal to input gunViewID
* public void PickUpGun(int gunViewID)
  + Test 1:
    - Input: Photon View ID of the Gun to pick up
    - Preconditions
      * Gun to be picked up does not equal (Gun)activeGun
    - Postconditions
      * Picked up gun’s collider is disabled
      * Picked up gun’s PhotonView is disabled
      * Picked up gun is in Player’s GameObject’s inventory
      * Picked up gun is disabled
      * Picked up gun’s position and rotation is set appropriately
      * Picked up gun’s fpsCam is equal to the Player’s camera
      * Picked up gun’s owner is equal to this Player
* public void DropAllItems
  + Test 1:
    - Preconditions
      * Locally controlled player has items
    - Postconditions
      * Locally controlled player has no items

#### Private Methods

* void DropGun(Gun gun)
  + Test 1:
    - Input: Gun to drop
    - Preconditions
      * Gun to drop is not null
      * Gun is equal to activeGun
      * Gun is not only gun in Player’s inventory
    - PostConditions
      * Gun’s owner in Room.CustomProperties is set to VALUE\_UNCLAIMED\_ITEM
      * Gun’s GameObject is sibling to Player’s GameObject in the GameObject Hierarchy
      * Gun’s position and rotation is set appropriately (relative to player’s position and rotate)
      * Gun’s collider is enabled
      * Gun’s PhotonView is enabled
      * activeGun is set to next gun in Player’s inventory
  + Test 2:
    - Input: Gun to drop
    - Preconditions
      * Gun to drop is not null
      * Gun is equal to activeGun
      * Gun is only gun in Player’s inventory
    - PostConditions
      * Gun’s owner in Room.CustomProperties is set to VALUE\_UNCLAIMED\_ITEM
      * Gun’s GameObject is sibling to Player’s GameObject in the GameObject Hierarchy
      * Gun’s position and rotation is set appropriately (relative to player’s position and rotate)
      * Gun’s collider is enabled
      * Gun’s PhotonView is enabled
      * activeGun is set to null
* void AddDeath()
  + Test 1
    - Postconditions
      * Player’s Deaths value is incremented in Player.CustomProperties
* void AddKill()
  + Test 1
    - Postconditions
      * Player’s Kills value is incremented in Player.CustomProperties
* void SetMode(string modeValue)
  + Test 1:
    - Input
      * modeValue string representing whether the player is alive or dead
    - Postconditions
      * Player’s mode is set in Player.CustomProperties

### WallDropAnimator Class Methods

#### Public Methods

* public void ResetWallPosition()
  + Test 1:
    - Preconditions
      * Wall is not in original position
    - Postconditions
      * Wall is in original position

### CameraAnimationHandler Class Methods

#### Public Methods

* void MoveLeft()
  + Test 1
    - Preconditions:
      * Launcher scene has loaded successfully
    - Postconditions
      * Main camera object animates successfully and rotates 90 degrees to the left of its previous resting position
* void MoveRight()
  + Test 1
    - Preconditions:
      * Launcher scene has loaded successfully
    - Postconditions
      * Main camera object animates successfully and rotates 90 degrees to the right of its previous resting position
* void MoveBack()
  + Test 1
    - Preconditions:
      * Launcher scene has loaded successfully
    - Postconditions
      * Main camera object animates successfully and rotates 180 degrees to the left of its previous resting position
* void MoveUp()
  + Test 1
    - Preconditions:
      * Launcher scene has loaded successfully
    - Postconditions
      * Main camera object animates successfully and rotates 90 degrees to the right and 90 degrees upwards of its previous resting position
* void MoveDown()
  + Test 1
    - Preconditions:
      * Launcher scene has loaded successfully
    - Postconditions
      * Main camera object animates successfully and rotates 90 degrees to the right and 90 degrees downwards of its previous resting position
* void ResetTriggers()
  + Test 1
    - Preconditions:
      * Launcher scene has loaded successfully
      * A camera animation has just concluded
    - Postconditions
      * Camera triggers are all set to false

### MobileButtonController Class Methods

#### Public Methods

* void ToggleChat()
  + Test 1
    - Preconditions:
      * Mobile Input is enabled
      * isChatEnabled boolean set to false
    - Postcondition
      * Chat CrossPlatformInput is set to ‘down’
      * isChatEnabled boolean set to true
  + Test 2
    - Precondition
      * Mobile Input is enabled
      * isChatEnabled boolean set to true
    - Postcondition
      * Chat CrossPlatformInput is set to ‘up’
      * isChatEnabled boolean set to false
* void ToggleStats()
  + Test 1
    - Preconditions:
      * Mobile Input is enabled
      * isStatsEnabled boolean set to false
    - Postcondition
      * Stats CrossPlatformInput is set to ‘down’
      * isStatsEnabled boolean set to true
  + Test 2
    - Preconditions:
      * Mobile Input is enabled
      * isStatsEnabled boolean set to true
    - Postcondition
      * Stats CrossPlatformInput is set to ‘up’
      * isStatsEnabled boolean set to false
* void Jump()
  + Test 1
    - Preconditions:
      * Mobile Input is enabled
    - Postconditions:
      * Jump CrossPlatformInput is set to ‘up’
* void UseHealth()
  + Test 1
    - Preconditions:
      * Mobile Input is enabled
    - Postconditions:
      * Jump CrossPlatformInput is set to ‘up’
* void ThrowGrenade()
  + Test 1
    - Preconditions:
      * Mobile Input is enabled
    - Postconditions:
      * Grenade CrossPlatformInput is set to ‘up’
* void ToggleAi()
  + Test 1
    - Preconditions:
      * Mobile Input is enabled
    - Postconditions:
      * AI CrossPlatformInput is set to ‘up’
* void CycleGun()
  + Test 1
    - Preconditions:
      * Mobile Input is enabled
    - Postconditions:
      * Cycle Gun CrossPlatformInput is set to ‘up’

### GamePlayFabController Class Methods

#### Public Methods

* void OnGetAccountInfoSuccess(GetAccountInfoResult result)
  + Test 1
    - Preconditions:
      * User is logged in
    - Postconditions
      * User’s returned username is consistent with what is stored in the database
* void getStats()
  + Test 1
    - Preconditions:
      * User is logged in
      * User has at least one stat in the database
    - Postconditions
      * User’s returned stats are consistent with what is in the database
* void OnGetStatistics(GetPlayerStatisticsResult result)
  + Test 1
    - Preconditions:
      * User is logged in
      * User has at least one stat to update
    - Postconditions
      * User’s relevant stats are updated on the database
      * User’s updated stats should be consistent in the result
* void IncrementKillCount()
  + Test 1
    - Preconditions:
      * User is logged in
      * User reduces another player’s health to below 0
    - Postconditions
      * User’s KillCount stat is updated by 1 in the database.
* void IncrementDeathCount()
  + Test 1
    - Preconditions:
      * User is logged in
      * User’s health is reduced to below 0.
    - Postconditions
      * User’s DeathCount stat is updated by 1 in the database.
* void IncrementRoundWins()
  + Test 1
    - Preconditions:
      * User is logged in
      * User is not on the team with 0 living players
    - Postconditions
      * User’s RoundWins stat is updated by 1 in the database.

### AICharacterController Class Methods

#### Public Methods

* public void SetTarget(Transform target)
  + Test 1
    - Inputs: Transform of a desired target
    - Preconditions:
      * AI is toggled on
    - Postconditions
      * this.target equals target

#### Private Methods

* void CreateDistanceGraph()
  + Test 1
    - Preconditions:
      * GameManager instance has a list of spawned weapons
    - Postconditions
      * graph is populated with correct distances
* Transform FindNextPointToTarget()
  + Test 1
    - Preconditions:
      * dist is populated with distances to all the weapon spawn points
    - Returns
      * Transform of the next point to target
* bool EnemyIsInCrosshairs(out PlayerManager enemy)
  + Test 1
    - Preconditions:
      * Enemy is in crosshairs
    - Returns
      * true if enemy is in crosshairs. Enemy: the PlayerManager of the player in crosshairs
* void ShootGun()
  + Test 1
    - Preconditions:
      * player has active gun
    - Postconditions
      * active gun is shot

### CountdownTimer Class Methods

#### Private Methods

* void CheckForTimerInfo()
  + Test 1
    - Preconditions:
      * stage 1 timer info is set in Room.CustomProperties
    - Postconditions
      * isTimer1Running = true
      * Timer1TimeIsUp = false
      * startTime1 = timer info from Room.CustomProperties
  + Test 2
    - Preconditions:
      * stage 2 timer info is set in Room.CustomProperties
    - Postconditions
      * isTimer2Running = true
      * Timer2TimeIsUp = false
      * startTime2 = timer info from Room.CustomProperties

## Integration Tests

#### PhotonIntegrationScript Methods

* TestPhotonConnection()
* TestPlayerPropertySync()

#### PlayfabIntegrationScript Methods

* TestPlayfabConnection()
* TestAccountCreation()

#### Use Case Integrations

|  |  |
| --- | --- |
| UC2: As a player, I want in-game UI, so I have visual feedback on what happens in-game | **What are we integrating?**  Unity api (via UI), Photon api, and Playfab  **We will be looking for:**  changes reflected in GUI that are indicative of events that happened in game (for example when one player kills another) |
| UC3: As a player, I want the ability to login to an account, so that my online presence persists across gameplay sessions | **What are we integrating?**  Testing Unity integration with Playfab  **We will be looking for:**  a genuine user account created that can be viewed from the backend Database system logs. We will then be looking for the ability to log back in after logging out. |
| UC6: As a player, I want user-friendly menu pages, so that I can easily navigate the functionality provided by the game | **What are we integrating?**  Unity API (UI-menus) with Photon  **We will be looking for:**  the presence of live networked information reflected on all clients. |
| UC7: As a player I want the option to play AI opponents so that I can enjoy the game even if I cannot find enough human players | **What are we integrating?**  Unity machine learning API with Photon Networking AP**I**  **We will be looking for:**  AI behavior that meets certain criteria for playing with humans. |
| UC10: As a player, I want matchmaking, so that it is easy to find a game to play. | **What are we integrating?**  Unity API (UI) with Photon  **We will be looking for:**  the presence of live networked information reflected on all clients relevant to matchmaking. |

## Acceptance Tests

|  |  |
| --- | --- |
| Test Case ID | TC0001 |
| Title | User Login |
| Requirement ID | RQ0001 |
| Priority/Risk | Critical |
| Technique | Manual |
| Description | When a user is not already logged on, ensure that the user can log on with any valid character combination for the username and password that links to a stored user. |
| Data Source | testdata.txt |
| Procedure | 1. Launch Armement.exe 2. Enter valid username into username field 3. Enter valid password into password field 4. Click login button |
| Expected Result | User account linked to provided credentials is recognized by the system, and that user is logged in on the running instance of Armament. |

|  |  |
| --- | --- |
| Test Case ID | TC0002 |
| Title | User Registration |
| Requirement ID | RQ0002 |
| Priority/Risk | Critical |
| Technique | Manual |
| Description | When a user is not already logged on, ensure that the user can register a new user account with any valid character combination for the username and password. |
| Data Source | testdata.txt |
| Procedure | 1. Launch Armement.exe 2. Click New User button 3. Enter valid username into username field 4. Enter valid password into password field 5. Enter valid email into email field 6. Enter valid display name into display name field 7. Click register button |
| Expected Result | Credentials are validated by system, a new user account is created, and the user is logged in to the new account |

|  |  |
| --- | --- |
| Test Case ID | TC0004 |
| Title | Game Arena Loads |
| Requirement ID | RQ0004 |
| Priority/Risk | Critical |
| Technique | Manual |
| Description | When a user enters a room, the arena successfully loads |
| Data Source | N/A |
| Procedure | 1. Successfully complete steps in TC0001 2. Click ‘Jump In’ |
| Expected Result | User avatar is loaded into the correct Unity scene. |

|  |  |
| --- | --- |
| Test Case ID | TC0005 |
| Title | User Movement |
| Requirement ID | RQ0005 |
| Priority/Risk | Critical |
| Technique | Manual |
| Description | When a user loads into an arena, they are able to successfully move the player avatar within the arena. |
| Data Source | N/A |
| Procedure | 1. Successfully complete steps from TC0004 2. Press all assigned keys |
| Expected Result | Actions linked to key assignments are visibly executed by player avatar. |

|  |  |
| --- | --- |
| Test Case ID | TC0007 |
| Title | Weapon Pickups |
| Requirement ID | RQ0007 |
| Priority/Risk | Critical |
| Technique | Manual |
| Description | Users can pick up weapons loaded into the scene. |
| Data Source | N/A |
| Procedure | 1. Successfully complete steps from TC0004 2. Walk into weapons in the environment |
| Expected Result | Weapons are removed from the scene, they are visible inside the players inventory, and they are equippable from the inventory. |

|  |  |
| --- | --- |
| Test Case ID | TC0008 |
| Title | Weapon Usage |
| Requirement ID | RQ0008 |
| Priority/Risk | Critical |
| Technique | Manual |
| Description | Users can equip weapons, and successfully use all provided firing functionality. |
| Data Source | N/A |
| Procedure | 1. Successfully complete steps from TC0007 2. Equip each weapon from the inventory 3. Fire primary fire functions. |
| Expected Result | Weapons are successfully equipped from inventory and visible on player avatar. Primary fire function is visibly initiated in the scene when used. |

|  |  |
| --- | --- |
| Test Case ID | TC0009 |
| Title | Item Pickups |
| Requirement ID | RQ0009 |
| Priority/Risk | High |
| Technique | Manual |
| Description | Users can pick up items loaded into the scene. |
| Data Source | N/A |
| Procedure | 1. Successfully complete steps from TC0004 2. Pick up items in the environment |
| Expected Result | Items are removed from the scene, they are visible inside the players inventory, and they are usable from the inventory. |

|  |  |
| --- | --- |
| Test Case ID | TC0010 |
| Title | Item Usage |
| Requirement ID | RQ0010 |
| Priority/Risk | Critical |
| Technique | Manual |
| Description | Users can successfully use items from inventory. |
| Data Source | N/A |
| Procedure | 1. Successfully complete steps from TC0009 2. Use each item from the inventory |
| Expected Result | Items are successfully used from the inventory. The effect defined for a given item is applied to the user. |

|  |  |
| --- | --- |
| Test Case ID | TC0011 |
| Title | Create Multiplayer Match |
| Requirement ID | RQ0011 |
| Priority/Risk | Critical |
| Technique | Manual |
| Description | Users can successfully create a multiplayer room for other players to join. |
| Data Source | testdata.txt |
| Procedure | 1. Successfully complete steps from TC0001 2. Click “Matchmaking” 3. Fill in appropriate info provided in Data Source 4. Click “Join Specific Room” |
| Expected Result | Player 1 is successfully loaded into the arena. |

|  |  |
| --- | --- |
| Test Case ID | TC0012 |
| Title | Join Multiplayer Match |
| Requirement ID | RQ0012 |
| Priority/Risk | Critical |
| Technique | Manual |
| Description | Users can successfully join a multiplayer room that has already been created. |
| Data Source | testdata.txt |
| Procedure | 1. Successfully complete steps from TC0011 2. On another machine, complete steps from TC0001 as a different user. 3. Click “Matchmaking” 4. Enter name of room that first player created 5. Click “Join Specific Room” |
| Expected Result | The room created by the first player is visible to the second player. The second player can successfully join this match, and is loaded into the same arena as the first player. |

|  |  |
| --- | --- |
| Test Case ID | TC0013 |
| Title | Multiplayer Action Sync |
| Requirement ID | RQ0013 |
| Priority/Risk | Critical |
| Technique | Manual |
| Description | Actions initiated by players are synchronized into other player views in under 1 second |
| Data Source | testdata.txt |
| Procedure | 1. Successfully complete steps from TC0012 2. Each player should ensure test logging is turned on. 3. Player 1 shoots player 2. 4. Player 1 picks up an item/weapon in view of Player 2. |
| Expected Result | Logging will log Player 1 shooting action and the PhotonNetwork.Time onto Player 1’s screen.  Logging will log Player 2 receiving damage and PhotonNetwork.Time onto Player 2’s screen.  Logging will log PhotonNetwork.Time on Client 2 when Player 2’s Health is set on Client 2.  Logging will log PhotonNetwork.Time on Client 1 when Player 2’s Health update is received on Client 1.  Logging will log PhotonNetwork.Time on Client 1 when Player 1’s pickup action occurs.  Logging will log PhotonNetwork.Time on Client 2 when Player 1’s pickup action is received.  Each successive local time from each log on separate machines should never exceed 1 second of delay. |

|  |  |
| --- | --- |
| Test Case ID | TC0014 |
| Title | Variable Player Avatars |
| Requirement ID | RQ0014 |
| Priority/Risk | High |
| Technique | Manual |
| Description | Players on different teams are displayed using different avatars. |
| Data Source | testdata.txt |
| Procedure | 1. Successfully complete steps from TC0012 2. Visually confirm different avatars are being used. |
| Expected Result | Player 1 should appear as an avatar appropriate for their team. Player 2 should appear as an avatar appropriate for their team. |

|  |  |
| --- | --- |
| Test Case ID | TC0015 |
| Title | HUD Updates |
| Requirement ID | RQ0015 |
| Priority/Risk | High |
| Technique | Manual |
| Description | Health changes are reflected on the user interface |
| Data Source | N/A |
| Procedure | 1. Successfully complete steps from TC0013 |
| Expected Result | Player 2’s health meter should reflect current health accurately after being shot. |

|  |  |
| --- | --- |
| Test Case ID | TC0017 |
| Title | Timer Stage Transition Occurs |
| Requirement ID | RQ0017 |
| Priority/Risk | Critical |
| Technique | Manual |
| Description | A stage transition occurs if the collection phase timer runs out. |
| Data Source | N/A |
| Procedure | 1. Successfully complete steps from TC0012. 2. Allow timer to run out. |
| Expected Result | Arena barrier should automatically collapse. Items and weapons should disappear from arena. These changes should be visible to all players in the arena. |

## 

## 

## 

## 

## 

## 

|  |  |
| --- | --- |
| Test Case ID | TC0018 |
| Title | Player-Enabled Stage Transition Occurs |
| Requirement ID | RQ0018 |
| Priority/Risk | Critical |
| Technique | Manual |
| Description | A stage transition occurs if a player activates the barrier override. |
| Data Source | N/A |
| Procedure | 1. Successfully complete steps from TC0012. 2. Shoot Barrier. |
| Expected Result | Arena barrier should collapse. Items and weapons should disappear from arena. These changes should be visible to all players in the arena. |

## 

|  |  |
| --- | --- |
| Test Case ID | TC0019 |
| Title | AI Players Usable |
| Requirement ID | RQ0019 |
| Priority/Risk | Medium |
| Technique | Manual |
| Description | A match can be setup with computer players, or ‘bots’. |
| Data Source | testdata.txt |
| Procedure | 1. Successfully complete steps from TC0012 2. Press “T” (or “AI” on mobile) |
| Expected Result | Player avatar should become an AI bot. |

## 

|  |  |
| --- | --- |
| Test Case ID | TC0020 |
| Title | Chat is Synced |
| Requirement ID | RQ0020 |
| Priority/Risk | Medium |
| Technique | Manual |
| Description | Messages sent through chat are synced across clients. |
| Data Source | N/A |
|  | 1. Successfully complete steps from TC00012 2. Press “Enter” to open chat box. 3. Type a message 4. Press “Enter” to send message |
| Expected Result | All clients in the match should receive the message. The message should be visible if they open their chat box. |

|  |  |
| --- | --- |
| Test Case ID | TC0021 |
| Title | Leaderboard is Synced |
| Requirement ID | RQ0021 |
| Priority/Risk | Medium |
| Technique | Manual |
| Description | Updates to player stats occur at appropriate times and sync across clients |
| Data Source | N/A |
|  | 1. Successfully complete steps from TC00012 2. Player A kills Player B 3. Exit Match 4. Player A checks kill leaderboard and ensures their kill count incremented. 5. Player B checks kill leaderboard and ensures Player A’s kill count incremented |
| Expected Result | Players kill count stat should update globally |

## 

|  |  |
| --- | --- |
| Test Case ID | TC0022 |
| Title | Add Friend Works |
| Requirement ID | RQ0022 |
| Priority/Risk | Medium |
| Technique | Manual |
| Description | Adding a username to your friends list successfully adds that user to your friend group |
| Data Source | testdata.txt |
|  | 1. Successfully complete steps from TC0001 2. Click ‘Friends/Leaderboards’ 3. Type Username provided in data source into ‘Enter Friend’s Username’ field 4. Click “Add Friend” 5. Click “Add Friend” again |
| Expected Result | First Add Friend should result in success message. Second add friend should notify that friend is already in group. |

## 

|  |  |
| --- | --- |
| Test Case ID | TC0023 |
| Title | Team Preference Works |
| Requirement ID | RQ0023 |
| Priority/Risk | Low |
| Technique | Manual |
| Description | Selecting team preference before joining a match results in player being assigned to that team if team is not full |
| Data Source | N/A |
|  | 1. Successfully complete steps from TC0001 2. Click ‘Matchmaingk’ 3. Change Team Preference 4. Click ‘Join Random’ |
| Expected Result | When loaded into match, user should be on the appropriate team |

## References

* Unity user Manual 2018.3 <https://docs.unity3d.com/Manual/index.html>
* Photon documentation <https://doc.photonengine.com/en-us/pun/v2/getting-started/pun-intro>
* Playfab documentation <https://api.playfab.com/docs/general-getting-started>