Loyalty Point Integration

Loyalty Points are the currency that players accumulate as a benefit of playing an application and can be used to purchase Rewards from PLAYSTUDIOS Reward partners. The system is centrally configured with earn rules that are triggered at specific points of gameplay or once-daily events such as a login. A game server must manage player Loyalty state and grant Loyalty Points according to earn rules, daily caps, lifetime caps, and initial balances.

The following playAWARDS server-to-server APIs will support the Loyalty Point system:

Loyalty Economy Configuration Query API

The playAWARDS Loyalty Economy Configuration Query API is a server-to-server API that is used to obtain the current rule sets regarding Loyalty earn methods, Loyalty caps, and initial balances for new players. A PLAYSTUDIOS economist will create and update Loyalty Economy configurations for each application partner taking into account the earn methods available in that particular application.

The game server should utilize the playAWARDS Loyalty Economy Configuration Query API to obtain the Loyalty Economy configuration at server startup and periodically during operation.

Loyalty API

The playAWARDS Loyalty API is a server-to-server API that can be used to add or deduct Loyalty Points.

MyVIP API

The playAWARDS myVIP API is a server-to-server API that can be used to add myVIP points or to query a player's current status in the myVIP system. A player's Tier Status is part of the criteria used in determining the amount of Loyalty awarded to a player during various earn methods.

Player Loyalty State

The game server environment should maintain a database (document store) that persists a player's Loyalty state. The suggested contents of this document are:

```
    /// <summary>
    /// Suggested database model for a player's game server Loyalty state.  The
game server should implement a repository
    /// with create, read, update, and delete operations on a container of these
items.
    /// </summary>
    public class PlayerLoyaltyState
    {
        /// <summary>
        /// Constructor
        /// </summary>
        /// <param name="playerId"></param>
        /// <param name="currentDayAmount"></param>
```

```
        /// <param name="currentAnonymousOrUnsyncedAmount"></param>
        /// <param name="dailyInstanceCounts"></param>
        /// <param name="createdAt"></param>
        /// <param name="lastResetAt"></param>
        /// <param name="lastSyncedAt"></param>
        public PlayerLoyaltyState(string playerId, decimal currentDayAmount,
decimal currentAnonymousOrUnsyncedAmount, IDictionary<string, int>
dailyInstanceCounts, DateTime createdAt, DateTime lastResetAt, DateTime?
lastSyncedAt)
        {
            PlayerId = playerId;
            CurrentDayAmount = currentDayAmount;
            CurrentAnonymousOrUnsyncedAmount = currentAnonymousOrUnsyncedAmount;
            DailyInstanceCounts = dailyInstanceCounts;
            CreatedAt = createdAt;
            LastResetAt = lastResetAt;
            LastSyncedAt = lastSyncedAt;
        }

        /// <summary>
        /// The unique id of the player within the partner's game server
        /// </summary>
        public string PlayerId { get; }

        /// <summary>
        /// The amount of PLAYAWARDS Loyalty Points obtained on the current day
        /// </summary>
        public decimal CurrentDayAmount { get; }

        /// <summary>
        /// The amount of Loyalty Points for an anonymous player or unsynced
authenticated player
        /// Anonymous Loyalty balances must be managed within the game server but
can be transferred to playAWARDS
        /// when the anonymous player merges their account with an authenticated
one
        /// </summary>
        public decimal CurrentAnonymousOrUnsyncedAmount { get; }

        /// <summary>
        /// A dictionary for tracking daily instances of Loyalty earn methods.
Certain earn methods can
        /// only earn points N times per day according to the
LoyaltyConfiguration.
        /// </summary>
        public IDictionary<string, int> DailyInstanceCounts { get; }

        /// <summary>
        /// The datetime that this item was created
        /// </summary>
        public DateTime CreatedAt { get; }

        /// <summary>
        /// The datetime that this item was last reset.  This is utilized to
```

```
    dictate resets of the CurrentDayAmount and
            /// DailyInstanceCounts when the date changes
            /// </summary>
            public DateTime LastResetAt { get; }

            /// <summary>
            /// The possible datetime that this player's Loyalty balance was last sent
    to playAWARDS.  This
            /// only applies to authenticated players.
            /// </summary>
            public DateTime? LastSyncedAt { get; }
        }
```

Suggested Game Server Interfaces

In order to facilitate integration, the game server should implement the following abstractions to the playAWARDS Loyalty Point system:

- `ILoyaltyConfigurationClient`: A singleton service that runs on the game server; contains the logic required to obtain and cache the Loyalty Economy configuration.
- `ILoyaltyRuleEngine`: A singleton service that standardizes queries into the Loyalty Economy configuration in terms of the player attributes and earn method attributes. The implementation should utilize **ILoyaltyConfigurationClient** to handle the specifics of obtaining the current **LoyaltyConfigurationQueryDTO**.
- `ILoyaltyClient`: A singleton service that runs on the game server; contains the logic required to call the playAWARDS Loyalty API to add Loyalty Points and query an authenticated player's Loyalty balance.
- `IMyVipClient`: A singleton service that runs on the game server; contains the logic required to call the playAWARDS myVIP API to add myVIP points or to query a player's current status in the myVIP system.
- `IPlayerAttributeService`: A singleton service that runs on the game server; contains the logic required to obtain a player's **game level**, **platform** (Android, iOS, or Web), and **identity type** (**Anonymous** or **Authenticated**). This abstraction is suggested purely for the convenience of integration with the **IPlayerLoyaltyLogicService.**
- `IPlayerLoyaltyStateRepository`: A singleton service that runs on the game server; serves as an abstraction to the **PlayerLoyaltyState** database. This service contains the logic required to create, read, update, and delete instances of **PlayerLoyaltyState**.
- `IPlayerLoyaltyLogicService`: A singleton service that serves to unify operations that create and update a **PlayerLoyaltyState**. The aforementioned abstractions should be utilized within the **IPlayerLoyaltyLogicService** to simplify the overall logic and to encapsulate details of each service from one another.

ILoyaltyConfigurationClient

ILoyaltyConfigurationClient is an abstraction of a service that returns the PLAYSTUDIOS Loyalty Economy Configuration. The game server should implement this as a REST API client responsible for making server-to-server requests to the playAWARDS Loyalty Economy Configuration Query API. It is recommended that the implementation utilizes a memory cache or external distributed cache that invalidates periodically so that the configuration is refreshed as the game server runs.

```
    public interface ILoyaltyConfigurationClient
    {
        /// <summary>
        /// Return the current LoyaltyConfiguration
        /// </summary>
        /// <returns></returns>
        Task<LoyaltyConfigurationQueryDTO> GetLoyaltyConfigurationAsync();
    }
```

Refer to the `LoyaltyConfigurationQueryDTO` for more information.

ILoyaltyRuleEngine

ILoyaltyRuleEngine is an abstraction of a service that standardizes queries into the Loyalty Economy configuration in terms of the player attributes and earn method attributes. Implementing this as a standalone service allows the rule query logic to be tested in isolation from the rest of the Loyalty system.

```
    /// <summary>
    /// Abstraction of a service that makes rule queries into the
LoyaltyConfiguration
    /// </summary>
    public interface ILoyaltyRuleEngine
    {
        /// <summary>
        /// Query the earn rules
        /// </summary>
        /// <param name="partnerApplicationId"></param>
        /// <param name="earnMethod"></param>
        /// <param name="earnCriteria"></param>
        /// <param name="occurrenceCount"></param>
        /// <param name="platform"></param>
        /// <param name="identityType"></param>
        /// <param name="myVipTier"></param>
        /// <param name="gameLevel"></param>
        /// <returns></returns>
        Task<LoyaltyEarnRuleQueryDTO> GetEarnRuleAsync(string earnMethod,
LoyaltyEarnCriteria earnCriteria, int occurrenceCount, LoyaltyPlatform platform,
LoyaltyIdentityType identityType, int? myVipTier, int? gameLevel);

        /// <summary>
        /// Query the daily cap rules
        /// </summary>
        /// <param name="partnerApplicationId"></param>
        /// <param name="identityType"></param>
        /// <param name="myVipTier"></param>
        /// <returns></returns>
        Task<LoyaltyDailyCapRuleDTO> GetDailyCapRuleAsync(LoyaltyIdentityType
identityType, int? myVipTier);

        /// <summary>
```

```
        /// Query the lifetime cap rules
        /// </summary>
        /// <param name="partnerApplicationId"></param>
        /// <param name="identityType"></param>
        /// <param name="myVipTier"></param>
        /// <returns></returns>
        Task<LoyaltyLifetimeCapRuleDTO>
GetLifetimeCapRuleAsync(LoyaltyIdentityType identityType, int? myVipTier);

        /// <summary>
        /// Query the initial balance rules
        /// </summary>
        /// <param name="partnerApplicationId"></param>
        /// <param name="identityType"></param>
        /// <param name="myVipTier"></param>
        /// <returns></returns>
        Task<LoyaltyInitialBalanceRuleDTO>
GetInitialBalanceRuleAsync(LoyaltyIdentityType identityType, int? myVipTier);

        /// <summary>
        /// Query the maximum amount of times that an authed account can be merged
with an anonymous account
        /// </summary>
        /// <param name="partnerApplicationId"></param>
        /// <returns></returns>
        Task<int> GetMaxAccountMergesAsync();

        /// <summary>
        /// Query the minimum amount of Loyalty required between sync operations
        /// </summary>
        /// <param name="partnerApplicationId"></param>
        /// <returns></returns>
        Task<long> GetMinSyncAmountAsync();
    }
```

The following is a sample implementation that demonstrates how queries are structured to handle nullable criteria and ordered by priority score. The sample implementation utilizes the `ILoyaltyConfigurationClient` abstraction.

```
    /// <summary>
    /// Sample implementation of ILoyaltyRuleEngine
    /// </summary>
    public class LoyaltyRuleEngine : ILoyaltyRuleEngine
    {
        private readonly ILoyaltyConfigurationClient LoyaltyConfigurationClient;

        /// <summary>
        /// Constructor
        /// </summary>
        /// <param name="loyaltyConfigurationClient"></param>
        public LoyaltyRuleEngine(ILoyaltyConfigurationClient
```

```
loyaltyConfigurationClient)
        {
            LoyaltyConfigurationClient = loyaltyConfigurationClient;
        }

        /// <summary>
        /// Query the daily cap rules
        /// </summary>
        /// <param name="partnerApplicationId"></param>
        /// <param name="identityType"></param>
        /// <param name="myVipTier"></param>
        /// <returns></returns>
        public async Task<LoyaltyDailyCapRuleDTO>
GetDailyCapRuleAsync(LoyaltyIdentityType identityType, int? myVipTier)
        {
            var config = await
LoyaltyConfigurationClient.GetLoyaltyConfigurationAsync();

            // rule query must handle nullable criteria in where clauses
            var matchingRules = from rule in config.DailyCapRules
                                where (rule.IdentityType ==
LoyaltyIdentityType.Any || rule.IdentityType == identityType)
                                && (rule.MyVipTierMin == null ||
(myVipTier.HasValue && rule.MyVipTierMin <= myVipTier))
                                && (rule.MyVipTierMax == null ||
(myVipTier.HasValue && rule.MyVipTierMax >= myVipTier))
                                orderby rule.PriorityScore descending
                                select rule;

            return matchingRules.First(); // throws if no rules match criteria
        }

        /// <summary>
        /// Query the earn rules
        /// </summary>
        /// <param name="earnMethod"></param>
        /// <param name="earnCriteria"></param>
        /// <param name="occurrenceCount"></param>
        /// <param name="platform"></param>
        /// <param name="identityType"></param>
        /// <param name="myVipTier"></param>
        /// <param name="gameLevel"></param>
        /// <returns></returns>
        public async Task<LoyaltyEarnRuleQueryDTO> GetEarnRuleAsync(string
earnMethod, LoyaltyEarnCriteria earnCriteria, int occurrenceCount, LoyaltyPlatform
platform, LoyaltyIdentityType identityType, int? myVipTier, int? gameLevel)
        {
            var config = await
LoyaltyConfigurationClient.GetLoyaltyConfigurationAsync();

            // rule query must handle nullable criteria in where clauses
            var matchingRules = from rule in config.EarnMethods[earnMethod]
                                where (!rule.OccurrenceMin.HasValue ||
rule.OccurrenceMin.Value <= occurrenceCount)
```

```
                                && (!rule.OccurrenceMax.HasValue ||
rule.OccurrenceMax.Value >= occurrenceCount)
                                && (!rule.GameLevelMin.HasValue ||
(gameLevel.HasValue && rule.GameLevelMin.Value <= gameLevel.Value))
                                && (!rule.GameLevelMax.HasValue ||
(gameLevel.HasValue && rule.GameLevelMax.Value >= gameLevel.Value))
                                && (!rule.MyVipTierMin.HasValue ||
(myVipTier.HasValue && rule.MyVipTierMin.Value <= myVipTier.Value))
                                && (!rule.MyVipTierMax.HasValue ||
(myVipTier.HasValue && rule.MyVipTierMax.Value >= myVipTier.Value))
                                && (rule.Platform == LoyaltyPlatform.Any ||
rule.Platform == platform)
                                && (rule.IdentityType == LoyaltyIdentityType.Any
|| rule.IdentityType == identityType)
                                && (rule.EarnCriteria == earnCriteria)
                                orderby rule.PriorityScore descending
                                select rule;

            return matchingRules.First(); // throws if no rules match criteria
        }

        /// <summary>
        /// Query the initial balance rules
        /// </summary>
        /// <param name="identityType"></param>
        /// <param name="myVipTier"></param>
        /// <returns></returns>
        public async Task<LoyaltyInitialBalanceRuleDTO>
GetInitialBalanceRuleAsync(LoyaltyIdentityType identityType, int? myVipTier)
        {
            var config = await
LoyaltyConfigurationClient.GetLoyaltyConfigurationAsync();

            // rule query must handle nullable criteria in where clauses
            var matchingRules = from rule in config.InitialBalanceRules
                                where (rule.IdentityType ==
LoyaltyIdentityType.Any || rule.IdentityType == identityType)
                                && (rule.MyVipTierMin == null ||
(myVipTier.HasValue && rule.MyVipTierMin <= myVipTier))
                                && (rule.MyVipTierMax == null ||
(myVipTier.HasValue && rule.MyVipTierMax >= myVipTier))
                                orderby rule.PriorityScore descending
                                select rule;

            return matchingRules.First(); // throws if no rules match criteria
        }

        /// <summary>
        /// Query the lifetime cap rules
        /// </summary>
        /// <param name="identityType"></param>
        /// <param name="myVipTier"></param>
        /// <returns></returns>
        public async Task<LoyaltyLifetimeCapRuleDTO>
```

```
GetLifetimeCapRuleAsync(LoyaltyIdentityType identityType, int? myVipTier)
        {
            var config = await
LoyaltyConfigurationClient.GetLoyaltyConfigurationAsync();

            // rule query must handle nullable criteria in where clauses
            var matchingRules = from rule in config.LifetimeCapRules
                                where (rule.IdentityType ==
LoyaltyIdentityType.Any || rule.IdentityType == identityType)
                                && (rule.MyVipTierMin == null ||
(myVipTier.HasValue && rule.MyVipTierMin <= myVipTier))
                                && (rule.MyVipTierMax == null ||
(myVipTier.HasValue && rule.MyVipTierMax >= myVipTier))
                                orderby rule.PriorityScore descending
                                select rule;

            return matchingRules.First(); // throws if no rules match criteria
        }

        /// <summary>
        /// Query the maximum amount of times that an authed account can be merged
with an anonymous account
        /// </summary>
        /// <returns></returns>
        public async Task<int> GetMaxAccountMergesAsync()
        {
            var config = await
LoyaltyConfigurationClient.GetLoyaltyConfigurationAsync();

            return config.MaxAccountMerges;
        }

        /// <summary>
        /// Query the minimum amount of Loyalty required between sync operations
        /// </summary>
        /// <returns></returns>
        public async Task<long> GetMinSyncAmountAsync()
        {
            var config = await
LoyaltyConfigurationClient.GetLoyaltyConfigurationAsync();

            return config.MinSyncAmount;
        }
    }
```

ILoyaltyClient

ILoyaltyClient is an abstraction of a service that can add Loyalty Points to a player's balance and obtain the player's current Loyalty balance. The game server should implement this as a REST API client responsible for making server-to-server requests to the playAWARDS Loyalty API using the `playAWARDSAccessToken`. The `playAWARDSAccessToken` is acquired with a server-to-server call at the time that the player session starts.

```
    /// <summary>
    /// Abstraction of a service that provides operations on the player's
PLAYSTUDIOS Loyalty Point balance.  The game server
    /// should implement this as a REST API client responsible for making server-
to-server requests
    /// to the playAWARDS Loyalty API using the assigned playAWARDS application
secret and the playAWARDSAccessToken for a player.
    ///
    /// It is recommended that the implementation utilizes a cache for each player
that invalidates periodically (time TBD)
    /// as well as anytime that points are added.
    /// </summary>
    public interface ILoyaltyClient
    {
        /// <summary>
        /// Add PLAYSTUDIOS Loyalty Points for the player
        /// </summary>
        /// <param name="playerId"></param>
        /// <param name="loyaltyDelta"></param>
        /// <returns></returns>
        Task<decimal> AddLoyaltyAsync(string playerId, decimal loyaltyDelta);

        /// <summary>
        /// Get the PLAYSTUDIOS Loyalty Point balance for the player
        /// </summary>
        /// <param name="playerId"></param>
        /// <returns></returns>
        Task<decimal> GetLoyaltyBalanceAsync(string playerId);

        /// <summary>
        /// Get the amount of PLAYSTUDIOS Loyalty earned over the lifetime of the
player
        /// </summary>
        /// <param name="playerId"></param>
        /// <returns></returns>
        Task<decimal> GetLifetimeLoyaltyEarnedAsync(string playerId);

        /// <summary>
        /// Get the amount of times that a player has merged accounts over the
lifetime of the player
        /// </summary>
        /// <param name="playerId"></param>
        /// <returns></returns>
        Task<int> GetLifetimeAccountMergesAsync(string playerId);
    }
```

IMyVipClient

IMyVipClient is an abstraction of a service that provides operations on the player's PLAYSTUDIOS myVIP point balance. The game server should implement this as a REST API client responsible for making server-to-

server requests to the playAWARDS myVip API using the `playAWARDSAccessToken` for a player. The `playAWARDSAccessToken` is acquired with a server-to-server call at the time that the player session starts.

```
    /// <summary>
    /// Abstraction of a service that provides operations on the player's
PLAYSTUDIOS myVIP point balance.  The game server
    /// should implement this as a REST API client responsible for making server-
to-server requests
    /// to the playAWARDS myVip API using the assigned playAWARDS application
secret and the playAWARDSAccessToken for a player.
    ///
    /// It is recommended that the implementation utilizes a cache for each player
that invalidates periodically (time TBD)
    /// as well as any time that points are added.
    /// </summary>
    public interface IMyVipClient
    {
        /// <summary>
        /// Gets the current PLAYSTUDIOS myVIP information for the player
        /// </summary>
        /// <param name="playerId"></param>
        /// <returns></returns>
        Task<VipCurrentStatusResponse> GetCurrentStatusAsync(string playerId);

        /// <summary>
        /// Add PLAYSTUDIOS myVIP points for the player
        /// </summary>
        /// <param name="playerId"></param>
        /// <param name="points"></param>
        /// <returns></returns>
        Task<VipAddPointsResponse> AddPointsAsync(string playerId, int points);

        /// <summary>
        /// Gets the current PLAYSTUDIOS myVIP configuration for all possible
tiers
        /// </summary>
        /// <param name="playerId"></param>
        /// <returns></returns>
        Task<VipGetConfigResponse> GetConfigAsync(string playerId);

        /// <summary>
        /// Performs a check to determine if the player has reached the end of a
        /// PLAYSTUDIOS Tier cycle
        /// </summary>
        /// <param name="playerId"></param>
        /// <returns></returns>
        Task<VipCheckEndOfStatusResponse> CheckEndOfStatusAsync(string playerId);
    }
}
```

playAWARDS myVip Responses and DTOs

```
[DataContract]
    public sealed class VipCurrentStatusResponse
    {
        [DataMember] public readonly VipStatus Status;

        [JsonConstructor]
        public VipCurrentStatusResponse(VipStatus status)
        {
            Status = status;
        }
    }

    [DataContract]
    public sealed class VipAddPointsResponse
    {
        /// <summary>
        /// Information about the change.
        /// </summary>
        [DataMember]
        public readonly VipEvent Event;

        /// <summary>
        /// The player's current status after the event has taken place.
        /// </summary>
        [DataMember]
        public readonly VipStatus CurrentStatus;

        [JsonConstructor]
        public VipAddPointsResponse(VipEvent vipEvent, VipStatus currentVipStatus)
        {
            Event = vipEvent;
            CurrentStatus = currentVipStatus;
        }
    }

    [DataContract]
    public sealed class VipGetConfigResponse
    {
        [DataMember] public readonly VipConfig Config;

        [JsonConstructor]
        public VipGetConfigResponse(VipConfig config)
        {
            Config = config;
        }
    }

    [DataContract]
    public sealed class VipCheckEndOfStatusResponse
    {
        /// <summary>
        /// The type of tier modification (if any).
```

```
        /// </summary>
        [DataMember]
        public readonly string EventType;

        /// <summary>
        /// The player's current status after the event has taken place.
        /// </summary>
        [DataMember]
        public readonly VipStatus CurrentStatus;

        [JsonConstructor]
        public VipCheckEndOfStatusResponse(string eventType, VipStatus
currentStatus)
        {
            EventType = eventType;
            CurrentStatus = currentStatus;
        }
    }

    [DataContract]
    public sealed class VipConfig
    {
        /// <summary>
        /// All total tier entries in the VIP table.
        /// </summary>
        [DataMember]
        public readonly Entry[] TierEntries;

        [DataMember]
        public readonly DisplayEntry[] Display;

        [JsonConstructor]
        public VipConfig(IEnumerable<DisplayEntry> display, IEnumerable<Entry>
tierEntries)
        {
            Display = display.ToArray();
            TierEntries = tierEntries.ToArray();
        }

        public VipConfig(VipConfig config)
        {
            TierEntries = config.TierEntries.Select(entry => new
Entry(entry)).ToArray();
        }

        [DataContract]
        public sealed class DisplayEntry
        {
            [DataMember]
            public readonly string TierEntryProperty;

            [DataMember]
            public readonly string DisplayName;
```

```
            [DataMember]
            public readonly int DisplayPriority;
            [JsonConstructor]
            public DisplayEntry(string tierEntryProperty, string displayName, int
displayPriority)
            {
                TierEntryProperty = tierEntryProperty;
                DisplayName = displayName;
                DisplayPriority = displayPriority;
            }
        }

        [DataContract]
        public sealed class Entry
        {
            /// <summary>
            /// The tier rank.
            /// </summary>
            [DataMember]
            public readonly int Tier;

            /// <summary>
            /// The total amount of days for a player to work on this tier goal.
            /// </summary>
            [DataMember]
            public readonly int DurationDays;

            /// <summary>
            /// The total amount of VIP points that must be accumulated to at
least retain this tier rank at the end of the duration of working towards the
goal.
            /// </summary>
            [DataMember]
            public readonly decimal RetainAmount;

            /// <summary>
            /// The total amount of VIP points that must be accumulated to tier up
to the next tier rank within the DurationDays amount of time.
            /// </summary>
            [DataMember]
            public readonly decimal TierUpAmount;

            [DataMember]
            public readonly bool HasPersonalHost;

            [JsonConstructor]
            public Entry(int tier, int durationDays, decimal retainAmount, decimal
tierUpAmount, bool hasPersonalHost)
            {
                Tier = tier;
                DurationDays = durationDays;
                RetainAmount = retainAmount;
                TierUpAmount = tierUpAmount;
                HasPersonalHost = hasPersonalHost;
```

```
            }

            public Entry(VipConfig.Entry entry)
            {
                Tier = entry.Tier;
                DurationDays = entry.DurationDays;
                RetainAmount = entry.RetainAmount;
                TierUpAmount = entry.TierUpAmount;
                HasPersonalHost = entry.HasPersonalHost;
            }
        }
    }

    [DataContract]
    public sealed class VipStatus
    {
        /// <summary>
        /// Player's tier.
        /// </summary>
        [DataMember]
        [JsonProperty]
        public int Tier { get; private set; }

        /// <summary>
        /// The total length of time the player has to work on their current tier
goal.
        /// </summary>
        [DataMember]
        [JsonProperty]
        public int DurationDays { get; private set; }

        /// <summary>
        /// The time the player started working on the current tier goal.
        /// </summary>
        [DataMember]
        [JsonProperty]
        public System.DateTime StartTime { get; private set; }

        /// <summary>
        /// The deadline when the player's current tier goal will end.
        /// </summary>
        [DataMember]
        [JsonProperty]
        public System.DateTime EndTime { get; private set; }

        /// <summary>
        /// The amount of VIP points the player must accumulate to at least retain
the current tier at the end of the current tier goal.
        /// </summary>
        [DataMember]
        [JsonProperty]
        public decimal RetainAmount { get; private set; }

        /// <summary>
```

```
        /// The amount of VIP points the player must accumulate to tier up to the
next tier.
        /// </summary>
        [DataMember]
        [JsonProperty]
        public decimal TierUpAmount { get; private set; }


        [DataMember]
        [JsonProperty]
        public bool HasPersonalHost { get; private set; }

        /// <summary>
        /// The player's current progress toward the current tier goal.
        /// </summary>
        [DataMember]
        [JsonProperty]
        public decimal CurrentAmount { get; private set; }

        public VipStatus(PlayerVipStatus playerVipStatus)
        {
            // Make sure to show the display status for all values below.
            Tier = playerVipStatus.TierLevelId;
            DurationDays = playerVipStatus.DurationDays;
            StartTime = playerVipStatus.StartTime;
            EndTime = playerVipStatus.EndTime;
            RetainAmount = playerVipStatus.RetainAmount;
            TierUpAmount = playerVipStatus.TierUpAmount;
            CurrentAmount = playerVipStatus.CurrentAmount;
            HasPersonalHost = playerVipStatus.HasPersonalHost;
        }

        public VipStatus(int internalTier, int tier, int durationDays, DateTime
startTime, DateTime endTime, decimal retainAmount, decimal tierUpAmount, decimal
currentAmount)
        {
            Tier = tier;
            DurationDays = durationDays;
            StartTime = startTime;
            EndTime = endTime;
            RetainAmount = retainAmount;
            TierUpAmount = tierUpAmount;
            CurrentAmount = currentAmount;
        }

        /// <summary>
        /// For deserialization only.
        /// </summary>
        [JsonConstructor]
        private VipStatus()
        {
        }
    }
```

```
    [DataContract]
    public sealed class PlayerVipStatus
    {
        [DataMember]
        public readonly int DurationDays;

        [DataMember]
        public readonly DateTime StartTime;

        [DataMember]
        public readonly DateTime EndTime;

        [DataMember]
        public readonly decimal RetainAmount;

        [DataMember]
        public readonly decimal TierUpAmount;

        [DataMember]
        public readonly decimal CurrentAmount;

        [DataMember]
        public readonly int TierLevelId;

        [DataMember]
        public readonly bool HasPersonalHost;

        [JsonConstructor]
        public PlayerVipStatus(int durationDays, DateTime startTime, DateTime
endTime, decimal retainAmount, decimal tierUpAmount, decimal currentAmount)
        {
            DurationDays = durationDays;
            StartTime = startTime;
            EndTime = endTime;
            RetainAmount = retainAmount;
            TierUpAmount = tierUpAmount;
            CurrentAmount = currentAmount;
            TierLevelId = tierLevelId;
        }

    }

    [DataContract]
    public sealed class VipEvent
    {
        /// <summary>
        /// The type of tier modification.
        /// </summary>
        [DataMember]
        public readonly string EventType;

        /// <summary>
        /// The total amount of VIP points that have changed on the current event.
        /// </summary>
```

```
        [DataMember]
        public readonly decimal VipPointsDelta;

        [JsonConstructor]
        public VipEvent(string eventType, decimal vipPointsDelta)
        {
            EventType = eventType;
            VipPointsDelta = vipPointsDelta;
        }

        public VipEvent(PlayerVipEvent playerVipEvent)
        {
            EventType = playerVipEvent.EventType.ToString();
            VipPointsDelta = playerVipEvent.VipPointsDelta;
        }
    }

    public enum TierEventType
    {
        Up,
        Down,
        RetainThreshold,
        RetainEndOfStatus,
        NoChange,
    }

    [DataContract]
    public sealed class PlayerVipEvent
    {
        [DataMember]
        public readonly TierEventType EventType;

        [DataMember]
        public readonly decimal VipPointsDelta;

        [DataMember]
        public readonly PlayerVipStatus CurrentStatus;

        [JsonConstructor]
        public PlayerVipEvent(TierEventType eventType, decimal vipPointsDelta,
PlayerVipStatus currentStatus)
        {
            EventType = eventType;
            VipPointsDelta = vipPointsDelta;
            CurrentStatus = currentStatus;
        }
    }
```

IPlayerAttributeService

IPlayerAttributeService is an abstraction of a service that contains the logic required to obtain a player's `GameLevel`, `MyVipTier`, `Platform` (Android, iOS, or Web), and `IdentityType` (**Anonymous** or

Authenticated). These values contribute to rule filtering within the `ILoyaltyRuleEngine`. The implementation should utilize the `IMyVipClient` abstraction to interface with the playAWARDS myVIP API, as well as any services that the game server needs to obtain the game level, platform, and identity type of the player.

```
    /// <summary>
    /// Abstraction of a service that provides attributes of the player for
matching rules within the PLAYSTUDIOS LoyaltyConfigurationQueryDTO
    /// </summary>
    public interface IPlayerAttributeService
    {
        /// <summary>
        /// Gets the player's game level if known (nullable)
        /// </summary>
        /// <param name="playerId"></param>
        /// <returns></returns>
        Task<int?> GetGameLevelAsync(string playerId);

        /// <summary>
        /// Gets the player's myVIP level if known (nullable)
        /// </summary>
        /// <param name="playerId"></param>
        /// <returns></returns>
        Task<int?> GetMyVipTierAsync(string playerId);

        /// <summary>
        /// Gets the player's LoyaltyPlatform (Android, iOS, or Web)
        /// </summary>
        /// <param name="playerId"></param>
        /// <returns></returns>
        Task<LoyaltyPlatform> GetPlatformAsync(string playerId);

        /// <summary>
        /// Gets the player's IdentityType (Anonymous or Authenticated)
        /// </summary>
        /// <param name="playerId"></param>
        /// <returns></returns>
        Task<LoyaltyIdentityType> GetIdentityTypeAsync(string playerId);
    }

    /// <summary>
    /// Enumeration type for each possible platform
    /// </summary>
    [JsonConverter(typeof(StringEnumConverter))]
    public enum LoyaltyPlatform
    {
        /// <summary>
        /// Android Platform
        /// </summary>
        Android,

        /// <summary>
```

```
        /// iOS Platform
        /// </summary>
        iOS,

        /// <summary>
        /// Web Platform
        /// </summary>
        Web
    }

    /// <summary>
    /// Enumeration type for a player's identity type.  Different rules and
policies may be specified for Anonymous vs Authenticated players
    /// </summary>
    [JsonConverter(typeof(StringEnumConverter))]
    public enum LoyaltyIdentityType
    {
        /// <summary>
        /// Anonymous Players
        /// </summary>
        Anonymous,

        /// <summary>
        /// Authenticated Players
        /// </summary>
        Authenticated
    }
```

IPlayerLoyaltyStateRepository

IPlayerLoyaltyStateRepository serves as an abstraction to the **PlayerLoyaltyState** database. This service contains the logic required to create, read, update, and delete instances of `PlayerLoyaltyState`. Implementation details will depend on the game server's underlying database.

```
    /// <summary>
    /// Suggested abstraction of a service that provides operations on a
repository of PlayerLoyaltyState items.  The game
    /// server should maintain a database of PlayerLoyaltyState and implement a
repository interface to that database
    /// </summary>
    public interface IPlayerLoyaltyStateRepository
    {
        /// <summary>
        /// Create a new PlayerLoyaltyState item for the player with the provided
initial balance
        /// </summary>
        /// <param name="playerId"></param>
        /// <param name="initialBalance"></param>
        /// <returns></returns>
        Task<PlayerLoyaltyState> CreatePlayerLoyaltyStateAsync(string playerId,
PlayerLoyaltyState initialState);
```

```
        /// <summary>
        /// Read the player's PlayerLoyaltyState
        /// </summary>
        /// <param name="playerId"></param>
        /// <returns></returns>
        Task<PlayerLoyaltyState> ReadPlayerLoyaltyStateAsync(string playerId);

        /// <summary>
        /// Update the player's PlayerLoyaltyState (without additional result)
        /// </summary>
        /// <param name="playerId"></param>
        /// <param name="transform"></param>
        /// <returns></returns>
        Task<PlayerLoyaltyState> UpdatePlayerLoyaltyStateAsync(string playerId,
Func<PlayerLoyaltyState, PlayerLoyaltyState> transform);

        /// <summary>
        /// Update the player's PlayerLoyaltyState (without additional result)
(async transform)
        /// </summary>
        /// <param name="playerId"></param>
        /// <param name="transform"></param>
        /// <returns></returns>
        Task<PlayerLoyaltyState> UpdatePlayerLoyaltyStateAsync(string playerId,
Func<PlayerLoyaltyState, Task<PlayerLoyaltyState>> asyncTransform);

        /// <summary>
        /// Update the player's PlayerLoyaltyState and return a result
        /// </summary>
        /// <param name="playerId"></param>
        /// <param name="transform"></param>
        /// <returns></returns>
        Task<Tuple<PlayerLoyaltyState, TResult>>
UpdatePlayerLoyaltyStateAsync<TResult>(string playerId, Func<PlayerLoyaltyState,
Tuple<PlayerLoyaltyState, TResult>> transformWithResult);

        /// <summary>
        /// Update the player's PlayerLoyaltyState and return a result (async
transform)
        /// </summary>
        /// <param name="playerId"></param>
        /// <param name="transform"></param>
        /// <returns></returns>
        Task<Tuple<PlayerLoyaltyState, TResult>>
UpdatePlayerLoyaltyStateAsync<TResult>(string playerId, Func<PlayerLoyaltyState,
Task<Tuple<PlayerLoyaltyState, TResult>>> asyncTransformWithResult);

        /// <summary>
        /// Delete the player's PlayerLoyaltyState
        /// </summary>
        /// <param name="playerId"></param>
        /// <returns></returns>
```

```
        Task<PlayerLoyaltyState> DeletePlayerLoyaltyStateAsync(string playerId);
    }
```

The following is the suggested format for `PlayerLoyaltyState`:

```
    /// <summary>
    /// Suggested database model for a player's game server Loyalty state.  The
game server should implement a repository
    /// with create, read, update, and delete operations on a container of these
items.
    /// </summary>
    public class PlayerLoyaltyState
    {
        /// <summary>
        /// Constructor
        /// </summary>
        /// <param name="playerId"></param>
        /// <param name="currentDayAmount"></param>
        /// <param name="anonymousLoyaltyBalance"></param>
        /// <param name="unsyncedAuthenticatedLoyaltyBalance"></param>
        /// <param name="dailyOccurrenceCounts"></param>
        /// <param name="createdAt"></param>
        /// <param name="lastResetAt"></param>
        /// <param name="lastSyncedAt"></param>
        public PlayerLoyaltyState(string playerId, decimal currentDayAmount,
decimal anonymousLoyaltyBalance, decimal unsyncedAuthenticatedLoyaltyBalance,
IDictionary<string, int> dailyOccurrenceCounts, DateTimeOffset createdAt,
DateTimeOffset lastResetAt, DateTimeOffset? lastSyncedAt)
        {
            PlayerId = playerId;
            CurrentDayAmount = currentDayAmount;
            AnonymousLoyaltyBalance = anonymousLoyaltyBalance;
            UnsyncedAuthenticatedLoyaltyBalance =
unsyncedAuthenticatedLoyaltyBalance;
            DailyOccurrenceCounts = dailyOccurrenceCounts;
            CreatedAt = createdAt;
            LastResetAt = lastResetAt;
            LastSyncedAt = lastSyncedAt;
        }

        /// <summary>
        /// The unique id of the player within the partner's game server
        /// </summary>
        public string PlayerId { get; }

        /// <summary>
        /// The amount of PLAYSTUDIOS Loyalty Points obtained on the current day
        /// </summary>
        public decimal CurrentDayAmount { get; }

        /// <summary>
        /// The amount of Loyalty Points for an anonymous player (kept aside from
```

```
UnsynchedAuthenticatedLoyaltyAmount due to account merge limits)
        /// </summary>
        public decimal AnonymousLoyaltyBalance { get; }

        /// <summary>
        /// The amount of Loyalty Points unsynced for an authenticated player
(kept aside from AnonymousLoyaltyAmount due to account merge limits)
        /// </summary>
        public decimal UnsyncedAuthenticatedLoyaltyBalance { get; }

        /// <summary>
        /// A dictionary for tracking daily occurrences of Loyalty earn methods.
Certain earn methods can
        /// only earn points N times per day according to the
LoyaltyConfiguration.
        /// </summary>
        public IDictionary<string, int> DailyOccurrenceCounts { get; }

        /// <summary>
        /// The datetime that this item was created
        /// </summary>
        public DateTimeOffset CreatedAt { get; }

        /// <summary>
        /// The datetime that this item was last reset.  This is utilized to
dictate resets of the CurrentDayAmount and
        /// DailyOccurrenceCounts when the date changes
        /// </summary>
        public DateTimeOffset LastResetAt { get; }

        /// <summary>
        /// The possible datetime that this player's Loyalty balance was last sent
to playAWARDS.  This
        /// only applies to authenticated players.
        /// </summary>
        public DateTimeOffset? LastSyncedAt { get; }
    }
```

IPlayerLoyaltyLogicService

IPlayerLoyaltyLogicService is an abstraction of a service that unifies operations and any dependencies that are involved in creating and updating a `PlayerLoyaltyState`. This should be implemented using the suggested game server interfaces for:

- `ILoyaltyRuleEngine` (which should utilize an `ILoyaltyConfigurationClient` in its implementation)
- `ILoyaltyClient`
- `IMyVipClient`
- `IPlayerAttributeService`
- `IPlayerLoyaltyStateRepository`
- `IDateTimeProvider` (in order to facilitate testing)

The following is a sample **IDateTimeProvider** file:

```
    //
    // Summary:
    //     Interface that abstracts DateTime operations
    public interface IDateTimeProvider
    {
        //
        // Summary:
        //     Get the current local DateTime.
        DateTime GetDateTimeNow();
        //
        // Summary:
        //     Get the current local DateTimeOffset (DateTime and timezone
offset).
        DateTimeOffset GetDateTimeOffsetNow();
        //
        // Summary:
        //     Get the current UTC DateTimeOffset (DateTime and zero timezone
offset).
        DateTimeOffset GetDateTimeOffsetUTCNow();
        //
        // Summary:
        //     Get the current UTC DateTime.
        DateTime GetDateTimeUTCNow();
    }
```

The integration of the PLAYSTUDIOS Loyalty Point system can be broken down into the following operations.

```
    /// <summary>
    /// Abstraction of a service that provides the interaction point between a
game server and the
    /// PLAYSTUDIOS Loyalty Point system.  The game server should implement this
service utilizing
    /// implementations of the suggested interfaces for:
    /// * ILoyaltyRuleEngine
    /// * ILoyaltyConfigurationClient
    /// * ILoyaltyClient
    /// * IPlayerAttributeService
    /// * IPlayerLoyaltyStateRepository
    /// </summary>
    public interface IPlayerLoyaltyLogicService
    {
        /// <summary>
        /// Create a PlayerLoyaltyState for a new player
        /// </summary>
        /// <param name="playerId"></param>
        /// <returns></returns>
        Task<PlayerLoyaltyResult> CreateAnonymousPlayerAsync(string playerId);
```

```
        /// <summary>
        /// Create a PlayerLoyaltyState for a new player
        /// </summary>
        /// <param name="playerId"></param>
        /// <returns></returns>
        Task<PlayerLoyaltyResult> CreateAuthenticatedPlayerAsync(string playerId);

        /// <summary>
        /// Start a new play session for a player
        /// </summary>
        /// <param name="playerId"></param>
        /// <returns></returns>
        Task<PlayerLoyaltyResult> StartPlaySessionAsync(string playerId);

        /// <summary>
        /// Earn Loyalty for a game specific event
        /// </summary>
        /// <param name="playerId"></param>
        /// <param name="earnMethod"></param>
        /// <returns></returns>
        Task<PlayerLoyaltyResult> EarnLoyaltyFromSingleEventAsync(string playerId,
string earnMethod);

        /// <summary>
        /// Earn Loyalty for a game specific event that represents an aggregation
(of chips, of minutes of ads viewed, etc)
        /// </summary>
        /// <param name="playerId"></param>
        /// <param name="earnMethod"></param>
        /// <param name="count"></param>
        /// <returns></returns>
        Task<PlayerLoyaltyResult> EarnLoyaltyFromMultipleEventAsync(string
playerId, string earnMethod, long count);

        /// <summary>
        /// Send any unsynched local Loyalty Points to playAWARDS
        /// </summary>
        /// <param name="playerId"></param>
        /// <returns></returns>
        Task<PlayerLoyaltyResult> SyncLoyaltyAsync(string playerId);

        /// <summary>
        /// Return eligibility of a player to merge anonymous Loyalty to an
authenticated account
        /// </summary>
        /// <param name="playerId"></param>
        /// <returns></returns>
        Task<bool> CanMergeLoyaltyAsync(string playerId);

        /// <summary>
        /// Merge an anonymous player's Loyalty Points into an authenticated
player's Loyalty state in playAWARDS
        /// </summary>
        /// <param name="playerId"></param>
```

```
        /// <returns></returns>
        Task<PlayerLoyaltyResult> MergeLoyaltyAsync(string playerId);
    }
```

Implementation Notes

Implementation of the **IPlayerLoyaltyLogicService** involves integration of the aforementioned services to:

- Obtain the current Loyalty Economy Configuration (`ILoyaltyConfigurationClient`).
- Query the Loyalty Economy Configuration for earn rules, daily cap rules, lifetime cap rules, initial balance rules, conditions that dictate anonymous account merges, and conditions that dictate the sync frequency of authenticated players (`ILoyaltyRuleEngine`).
- Obtain attributes of the player such as game level, Tier (`IMyVipClient`), platform, and identity type (`IPlayerAttributeService`).
- Create, read, update, (and delete) a player's Loyalty state stored by the game server (`IPlayerLoyaltyStateRepository`).
- Perform ongoing updates to the player's Loyalty state to:
    - Track an anonymous player's total Loyalty balance (prior to a merge operation into an authenticated player's balance, subject to the `MaxAccountMerges` condition).
    - Track an authenticated player's unsynced Loyalty balance (prior to a sync operation, subject to the `MinSyncAmount` condition).
    - Track a player's daily Loyalty amount (to enforce daily caps).
    - Track occurrence counts of Loyalty earn methods (to match earn method rules dictated by daily occurrence counts).
    - Reset the player's daily Loyalty amount and occurrence counts of Loyalty earn methods at the start of a new day.
- Obtain and add points to an authenticated player's PLAYSTUDIOS Loyalty balance (`ILoyaltyClient`).
- Obtain a player's lifetime Loyalty earned and number of lifetime account merges (`ILoyaltyClient`).

The implementation of `IPlayerLoyaltyLogicService` must interact with game server code involving:

- Creation of new players.
- The start of a player session.
- Detecting a new day for the purposes of resetting daily occurrence counters and daily cap room.
- Earning Loyalty from a specific single event (such as daily login or game-specific challenge/milestone).
- Earning Loyalty from some multiple event (fractional Loyalty **X** multiplied by **N** items, such as spending in-game currency/chips, or **N** seconds of ads viewed).
- Merging an anonymous player with an authenticated player (might require reaching out to the playAWARDS Account API).
- Interfacing with playAWARDS APIs for account creation or merging, and operations to Loyalty and myVip balances.

The details of unifying these services in a game server implementation will differ depending on game server language and framework and the presence of existing services that provide these operations.

Sample Implementation

The following is a sample implementation in C# .NET Core (use of equivalent async/await or asynchronous task-based logic in the game server's framework is strongly recommended). Logging and integration with game server telemetry and event monitoring is recommended but will depend on implementation by the game server.

Specific details of exception handling will differ depending on game server framework, but the game server should consider and handle the possibility of fault scenarios caused by:

- Failure to match rules in the Loyalty Economy Configuration earn rules, daily cap rules, lifetime cap rules, initial balance rules.
- Attempting to merge to an authenticated account that has reached **MaxAccountMerges**.
- Timeouts or non-successful responses from the playAWARDS API.
- Conflicts creating or updating player Loyalty state in the repository/database implementation.

```
    /// <summary>
    /// Example implementation of the IPlayerLoyaltyLogicService
    /// </summary>
    public class PlayerLoyaltyLogicService : IPlayerLoyaltyLogicService
    {
        private readonly ILoyaltyRuleEngine LoyaltyRuleEngine;

        private readonly ILoyaltyClient LoyaltyClient;

        private readonly IPlayerAttributeService PlayerAttributeService;

        private readonly IPlayerLoyaltyStateRepository
PlayerLoyaltyStateRepository;

        private readonly IDateTimeProvider DateTimeProvider;

        public PlayerLoyaltyLogicService(ILoyaltyRuleEngine loyaltyRuleEngine,
ILoyaltyClient loyaltyClient, IPlayerAttributeService playerAttributeService,
IPlayerLoyaltyStateRepository playerLoyaltyStateRepository, IDateTimeProvider
dateTimeProvider)
        {
            LoyaltyRuleEngine = loyaltyRuleEngine;
            LoyaltyClient = loyaltyClient;
            PlayerAttributeService = playerAttributeService;
            PlayerLoyaltyStateRepository = playerLoyaltyStateRepository;
            DateTimeProvider = dateTimeProvider;
        }

        /// <summary>
        /// Create a PlayerLoyaltyState for a new player
        /// </summary>
        /// <param name="playerId"></param>
        /// <returns></returns>
        public async Task<PlayerLoyaltyResult> CreateAnonymousPlayerAsync(string
playerId)
        {
            // get rule criteria
```

```
            var myVipTier = await
PlayerAttributeService.GetMyVipTierAsync(playerId).ConfigureAwait(false);

            // get the initial balance rule
            var rule = await
LoyaltyRuleEngine.GetInitialBalanceRuleAsync(LoyaltyIdentityType.Anon, myVipTier);

            // create the player's Loyalty state (should throw if player Loyalty
state already exists)
            var dateTimeNow = DateTimeProvider.GetDateTimeOffsetUTCNow();

            var initialState = new PlayerLoyaltyState(playerId,
rule.InitialBalance, rule.InitialBalance, 0m, new Dictionary<string, int>(),
dateTimeNow, dateTimeNow, new DateTimeOffset?());

            await
PlayerLoyaltyStateRepository.CreatePlayerLoyaltyStateAsync(playerId,
initialState).ConfigureAwait(false);

            // sync initial Loyalty balance
            var syncResult = await
SyncLoyaltyAsync(playerId).ConfigureAwait(false);

            // return end result of initial balance and sync operation
            return new PlayerLoyaltyResult(syncResult.SyncOccurred,
syncResult.SyncAmount, initialState.AnonymousLoyaltyBalance,
syncResult.LoyaltyState);
        }

        /// <summary>
        /// Create a PlayerLoyaltyState for a new player
        /// </summary>
        /// <param name="playerId"></param>
        /// <returns></returns>
        public async Task<PlayerLoyaltyResult>
CreateAuthenticatedPlayerAsync(string playerId)
        {
            // get rule criteria
            var myVipTier = await
PlayerAttributeService.GetMyVipTierAsync(playerId).ConfigureAwait(false);

            // get the initial balance rule
            var rule = await
LoyaltyRuleEngine.GetInitialBalanceRuleAsync(LoyaltyIdentityType.Auth, myVipTier);

            // create the player's Loyalty state (should throw if player Loyalty
state already exists)
            var dateTimeNow = DateTimeProvider.GetDateTimeOffsetUTCNow();

            var initialState = new PlayerLoyaltyState(playerId,
rule.InitialBalance, 0m, rule.InitialBalance, new Dictionary<string, int>(),
dateTimeNow, dateTimeNow, new DateTimeOffset?());

            await
```

```
PlayerLoyaltyStateRepository.CreatePlayerLoyaltyStateAsync(playerId,
initialState).ConfigureAwait(false);

            // sync initial Loyalty balance
            var syncResult = await
SyncLoyaltyAsync(playerId).ConfigureAwait(false);

            // return end result of initial balance and sync operation
            return new PlayerLoyaltyResult(syncResult.SyncOccurred,
syncResult.SyncAmount, initialState.UnsyncedAuthenticatedLoyaltyBalance,
syncResult.LoyaltyState);
        }


        /// <summary>
        /// Earn Loyalty for a game specific event that represents an aggregation
(of chips, of minutes of ads viewed, etc)
        /// </summary>
        /// <param name="playerId"></param>
        /// <param name="earnMethod"></param>
        /// <param name="count"></param>
        /// <returns></returns>
        public async Task<PlayerLoyaltyResult>
EarnLoyaltyFromMultipleEventAsync(string playerId, string earnMethod, long count)
        {
            // reset daily counts if necessary
            await CheckForNewDayAsync(playerId).ConfigureAwait(false);

            // add or increment daily occurrence count of this earn method
            await AddOccurrenceOfEarnMethod(playerId,
earnMethod).ConfigureAwait(false);

            // get the identity type of the player
            var identityType = await
PlayerAttributeService.GetIdentityTypeAsync(playerId).ConfigureAwait(false);

            // use rule engine and player Loyalty state to calculate and add
loyalty
            var earnResult = await EarnLoyaltyMultipleEventAsync(playerId,
identityType, earnMethod, count);

            // sync the Loyalty
            var syncResult = await
SyncLoyaltyAsync(playerId).ConfigureAwait(false);

            // return end result of earn amount and sync operation
            return new PlayerLoyaltyResult(syncResult.SyncOccurred,
syncResult.SyncAmount, earnResult.Item2, syncResult.LoyaltyState);
        }


        /// <summary>
        /// Earn Loyalty for a game specific event
        /// </summary>
```

```
        /// <param name="playerId"></param>
        /// <param name="earnMethod"></param>
        /// <returns></returns>
        public async Task<PlayerLoyaltyResult>
EarnLoyaltyFromSingleEventAsync(string playerId, string earnMethod)
        {
            // reset daily counts if necessary
            await CheckForNewDayAsync(playerId).ConfigureAwait(false);

            // add or increment daily occurrence count of this earn method
            await AddOccurrenceOfEarnMethod(playerId,
earnMethod).ConfigureAwait(false);

            // get the identity type of the player
            var identityType = await
PlayerAttributeService.GetIdentityTypeAsync(playerId).ConfigureAwait(false);

            // use rule engine and player Loyalty state to calculate and add
loyalty
            var earnResult = await EarnLoyaltySingleEventAsync(playerId,
identityType, earnMethod);

            // sync the Loyalty
            var syncResult = await
SyncLoyaltyAsync(playerId).ConfigureAwait(false);

            // return end result of earn amount and sync operation
            return new PlayerLoyaltyResult(syncResult.SyncOccurred,
syncResult.SyncAmount, earnResult.Item2, syncResult.LoyaltyState);
        }

        /// <summary>
        /// Return eligibility of a player to merge anonymous Loyalty to an
authenticated account
        /// </summary>
        /// <param name="playerId"></param>
        /// <returns></returns>
        public async Task<bool> CanMergeLoyaltyAsync(string playerId)
        {
            var identityType = await
PlayerAttributeService.GetIdentityTypeAsync(playerId).ConfigureAwait(false);

            if (identityType == LoyaltyIdentityType.Anon) return false;  // player
must currently be authed in order to merge

            var maxAccountMerges = await
LoyaltyRuleEngine.GetMaxAccountMergesAsync();

            var playerLifetimeAccountMerges = await
LoyaltyClient.GetLifetimeAccountMergesAsync(playerId);

            return (playerLifetimeAccountMerges < maxAccountMerges);
        }
```

```
        /// <summary>
        /// Merge an anonymous player's Loyalty Points into an authenticated
player's Loyalty state in playAWARDS
        /// </summary>
        /// <param name="playerId"></param>
        /// <returns></returns>
        public async Task<PlayerLoyaltyResult> MergeLoyaltyAsync(string playerId)
        {
            var canMergeLoyalty = await
CanMergeLoyaltyAsync(playerId).ConfigureAwait(false);

            if (canMergeLoyalty)
            {
                // update the player Loyalty state now that all previous steps
have succeeded
                await PlayerLoyaltyStateRepository.UpdatePlayerLoyaltyStateAsync(
                    playerId,
                    existing =>
                    {
                        // move the AnonymousLoyaltyBalance to
UnsyncedAuthenticatedLoyaltyBalance
                        return new PlayerLoyaltyState(
                                playerId,
                                existing.CurrentDayAmount,
                                0m,
                                existing.AnonymousLoyaltyBalance,
                                existing.DailyOccurrenceCounts,
                                existing.CreatedAt,
                                existing.LastResetAt,
                                existing.LastSyncedAt);
                    }).ConfigureAwait(false);

                // need to incorporate merge protocol with playAWARDS Account API
(TBD)
                throw new NotImplementedException();
            }
            else
            {
                // attempt to sync Loyalty
                return await SyncLoyaltyAsync(playerId).ConfigureAwait(false);
            }
        }

        /// <summary>
        /// Start a new play session for a player
        /// </summary>
        /// <param name="playerId"></param>
        /// <returns></returns>
        public async Task<PlayerLoyaltyResult> StartPlaySessionAsync(string
playerId)
        {
            // check for new day to reset daily trackers
            await CheckForNewDayAsync(playerId).ConfigureAwait(false);
```

```
            // sync Loyalty balance
            return await SyncLoyaltyAsync(playerId).ConfigureAwait(false);
        }


        /// <summary>
        /// Send any unsynched local Loyalty Points to playAWARDS
        /// </summary>
        /// <param name="playerId"></param>
        /// <returns></returns>
        public async Task<PlayerLoyaltyResult> SyncLoyaltyAsync(string playerId)
        {
            // read the player Loyalty state
            var state = await
PlayerLoyaltyStateRepository.ReadPlayerLoyaltyStateAsync(playerId).ConfigureAwait(
false);

            // get identity criteria for syncing loyalty
            var identityType = await
PlayerAttributeService.GetIdentityTypeAsync(playerId).ConfigureAwait(false);

            // early return if this is not an authenticated player
            if (identityType == LoyaltyIdentityType.Anon) return new
PlayerLoyaltyResult(false, 0m, 0m, state);

            var minSyncAmount = await LoyaltyRuleEngine.GetMinSyncAmountAsync();

            // specific date handling implementation may need to be customized
here
            var dateTimeNow = DateTimeProvider.GetDateTimeOffsetUTCNow();

            // determine if we have accrued enough Loyalty to send update to
playAWARDS
            var canSyncLoyalty = state.LastSyncedAt == null ||
(state.UnsyncedAuthenticatedLoyaltyBalance >= minSyncAmount);

            // early return if we cannot sync Loyalty yet
            if (!canSyncLoyalty) return new PlayerLoyaltyResult(false, 0m, 0m,
state);

            // send an update to remote Loyalty system (playAWARDS Loyalty API)
            var amountSynced = await LoyaltyClient.AddLoyaltyAsync(playerId,
state.UnsyncedAuthenticatedLoyaltyBalance).ConfigureAwait(false);

            // update the player Loyalty state now that all previous steps have
succeeded
            var updatedStateAndAmountSynced = await
PlayerLoyaltyStateRepository.UpdatePlayerLoyaltyStateAsync(
                playerId,
                existing =>
                {
                    // clear the UnsyncedAuthenticatedLoyaltyBalance and set the
LastResetAt time
                    return Tuple.Create(
```

```
                        new PlayerLoyaltyState(
                            playerId,
                            existing.CurrentDayAmount,
                            existing.AnonymousLoyaltyBalance,
                            existing.UnsyncedAuthenticatedLoyaltyBalance -
amountSynced,  // calculation here in case not all Loyalty is synced
                            existing.DailyOccurrenceCounts,
                            existing.CreatedAt,
                            existing.LastResetAt,
                            dateTimeNow),
                        amountSynced);
                }).ConfigureAwait(false);

            return new PlayerLoyaltyResult(true,
updatedStateAndAmountSynced.Item2, 0m, updatedStateAndAmountSynced.Item1);
        }


        /// <summary>
        /// Factored utility that checks if daily Loyalty and earn method
occurrence counts should be reset
        /// </summary>
        /// <param name="playerId"></param>
        /// <returns></returns>
        private async Task<PlayerLoyaltyState> CheckForNewDayAsync(string
playerId)
        {
            // resets player Loyalty state pertaining to daily cap and earn method
occurrence counts if current datetime is different than updatedAt datetime
            var updateResult = await
PlayerLoyaltyStateRepository.UpdatePlayerLoyaltyStateAsync(
                playerId,
                existing =>
                {
                    var dateTimeNow = DateTimeProvider.GetDateTimeOffsetUTCNow();

                    // specific date rollover implementation may need to be
customized here
                    if (existing.LastResetAt.Date != dateTimeNow)
                    {
                        // clear the CurrentDayAmount and reset the
DailyOccurrenceCounts dictionary
                        return new PlayerLoyaltyState(
                                playerId,
                                0m,
                                existing.AnonymousLoyaltyBalance,
                                existing.UnsyncedAuthenticatedLoyaltyBalance,
                                new Dictionary<string, int>(),
                                existing.CreatedAt,
                                dateTimeNow,
                                existing.LastSyncedAt);
                    }
                    else
                    {
```

```
                        return existing;
                    }
                }).ConfigureAwait(false);

            return updateResult;
        }

        /// <summary>
        /// Factored utility that adds or increments a daily occurrence count in
the player Loyalty state
        /// </summary>
        /// <param name="playerId"></param>
        /// <param name="earnMethod"></param>
        /// <returns></returns>
        private async Task<PlayerLoyaltyState> AddOccurrenceOfEarnMethod(string
playerId, string earnMethod)
        {
            return await
PlayerLoyaltyStateRepository.UpdatePlayerLoyaltyStateAsync(
                playerId,
                existing =>
                {
                    // add or increment the DailyOccurrenceCount of this
earnMethod
                    if (existing.DailyOccurrenceCounts.ContainsKey(earnMethod))
                    {
                        existing.DailyOccurrenceCounts[earnMethod]++;
                    }
                    else
                    {
                        existing.DailyOccurrenceCounts[earnMethod] = 1;
                    }

                    return existing;
                }).ConfigureAwait(false);
        }

        /// <summary>
        /// Earn Loyalty for a game specific single event
        /// </summary>
        /// <param name="playerId"></param>
        /// <param name="earnMethod"></param>
        /// <returns></returns>
        private async Task<Tuple<PlayerLoyaltyState, decimal>>
EarnLoyaltySingleEventAsync(string playerId, LoyaltyIdentityType identityType,
string earnMethod)
        {
            // get player attributes
            var gameLevel = await
PlayerAttributeService.GetGameLevelAsync(playerId).ConfigureAwait(false);
            var myVipTier = await
PlayerAttributeService.GetMyVipTierAsync(playerId).ConfigureAwait(false);
            var platform = await
PlayerAttributeService.GetPlatformAsync(playerId).ConfigureAwait(false);
```

```
            // update the player Loyalty state
            return await
PlayerLoyaltyStateRepository.UpdatePlayerLoyaltyStateAsync(
                playerId,
                async existing =>
                {
                    // authenticated single event case
                    var earnRule = await
LoyaltyRuleEngine.GetEarnRuleAsync(earnMethod, LoyaltyEarnCriteria.Single,
existing.DailyOccurrenceCounts[earnMethod], platform, identityType, myVipTier,
gameLevel);
                    var dailyCapRule = await
LoyaltyRuleEngine.GetDailyCapRuleAsync(identityType, myVipTier);
                    var lifetimeCapRule = await
LoyaltyRuleEngine.GetLifetimeCapRuleAsync(identityType, myVipTier);

                    // the Loyalty amount is exactly what is stated in the rule
                    var amountToAdd = earnRule.Amount;

                    if (identityType == LoyaltyIdentityType.Auth)
                    {
                        var lifetimeLoyaltyEarned = await
LoyaltyClient.GetLifetimeLoyaltyEarnedAsync(playerId);

                        // cap room calculations
                        var dailyCapRoom = Math.Max(0m, dailyCapRule.Cap -
existing.CurrentDayAmount);
                        var lifetimeCapRoom = Math.Max(0m, lifetimeCapRule.Cap -
(lifetimeLoyaltyEarned + existing.UnsyncedAuthenticatedLoyaltyBalance));
                        var eligibleToAdd = new[] { dailyCapRoom, lifetimeCapRoom,
amountToAdd }.Min();

                        return Tuple.Create(
                            new PlayerLoyaltyState(
                                playerId,
                                existing.CurrentDayAmount + eligibleToAdd,
                                existing.AnonymousLoyaltyBalance,  // keep
whatever exists here
                                existing.UnsyncedAuthenticatedLoyaltyBalance +
eligibleToAdd,
                                existing.DailyOccurrenceCounts,
                                existing.CreatedAt,
                                existing.LastResetAt,
                                existing.LastSyncedAt),
                            eligibleToAdd);
                    }
                    else
                    {
                        // cap room calculations
                        var dailyCapRoom = Math.Max(0m, dailyCapRule.Cap -
existing.CurrentDayAmount);
                        var lifetimeCapRoom = Math.Max(0m, lifetimeCapRule.Cap -
existing.AnonymousLoyaltyBalance);
```

```
                        var eligibleToAdd = new[] { dailyCapRoom, lifetimeCapRoom,
amountToAdd }.Min();

                        return Tuple.Create(
                            new PlayerLoyaltyState(
                                playerId,
                                existing.CurrentDayAmount + eligibleToAdd,
                                existing.AnonymousLoyaltyBalance + eligibleToAdd,
                                existing.UnsyncedAuthenticatedLoyaltyBalance,   //
keep whatever exists here
                                existing.DailyOccurrenceCounts,
                                existing.CreatedAt,
                                existing.LastResetAt,
                                existing.LastSyncedAt),
                            eligibleToAdd);
                    }
                }).ConfigureAwait(false);
        }

        /// <summary>
        /// Earn Loyalty for a game specific event that represents an aggregation
(of chips, of minutes of ads viewed, etc)
        /// </summary>
        /// <param name="playerId"></param>
        /// <param name="earnMethod"></param>
        /// <param name="count"></param>
        /// <returns></returns>
        private async Task<Tuple<PlayerLoyaltyState, decimal>>
EarnLoyaltyMultipleEventAsync(string playerId, LoyaltyIdentityType identityType,
string earnMethod, long count)
        {
            // get player attributes
            var gameLevel = await
PlayerAttributeService.GetGameLevelAsync(playerId).ConfigureAwait(false);
            var myVipTier = await
PlayerAttributeService.GetMyVipTierAsync(playerId).ConfigureAwait(false);
            var platform = await
PlayerAttributeService.GetPlatformAsync(playerId).ConfigureAwait(false);

            // update the player Loyalty state
            return await
PlayerLoyaltyStateRepository.UpdatePlayerLoyaltyStateAsync(
                playerId,
                async existing =>
                {
                    // anonymous multiple event case
                    var earnRule = await
LoyaltyRuleEngine.GetEarnRuleAsync(earnMethod, LoyaltyEarnCriteria.Multiple,
existing.DailyOccurrenceCounts[earnMethod], platform, identityType, myVipTier,
gameLevel);
                    var dailyCapRule = await
LoyaltyRuleEngine.GetDailyCapRuleAsync(identityType, myVipTier);
                    var lifetimeCapRule = await
LoyaltyRuleEngine.GetLifetimeCapRuleAsync(identityType, myVipTier);
```

```
                    // apply the rule's Loyalty amount to each item in the count
                    var amountToAdd = earnRule.Amount * count;

                    if (identityType == LoyaltyIdentityType.Auth)
                    {
                        var lifetimeLoyaltyEarned = await
LoyaltyClient.GetLifetimeLoyaltyEarnedAsync(playerId);

                        // cap room calculations
                        var dailyCapRoom = Math.Max(0m, dailyCapRule.Cap -
existing.CurrentDayAmount);
                        var lifetimeCapRoom = Math.Max(0m, lifetimeCapRule.Cap -
(lifetimeLoyaltyEarned + existing.UnsyncedAuthenticatedLoyaltyBalance));
                        var eligibleToAdd = new[] { dailyCapRoom, lifetimeCapRoom,
amountToAdd }.Min();

                        return Tuple.Create(
                            new PlayerLoyaltyState(
                                playerId,
                                existing.CurrentDayAmount + eligibleToAdd,
                                existing.AnonymousLoyaltyBalance,  // keep
whatever exists here
                                existing.UnsyncedAuthenticatedLoyaltyBalance +
eligibleToAdd,
                                existing.DailyOccurrenceCounts,
                                existing.CreatedAt,
                                existing.LastResetAt,
                                existing.LastSyncedAt),
                            eligibleToAdd);
                    }
                    else
                    {
                        // cap room calculations
                        var dailyCapRoom = Math.Max(0m, dailyCapRule.Cap -
existing.CurrentDayAmount);
                        var lifetimeCapRoom = Math.Max(0m, lifetimeCapRule.Cap -
existing.AnonymousLoyaltyBalance);
                        var eligibleToAdd = new[] { dailyCapRoom, lifetimeCapRoom,
amountToAdd }.Min();

                        return Tuple.Create(
                            new PlayerLoyaltyState(
                                playerId,
                                existing.CurrentDayAmount + eligibleToAdd,
                                existing.AnonymousLoyaltyBalance + eligibleToAdd,
                                existing.UnsyncedAuthenticatedLoyaltyBalance,   //
keep whatever exists here
                                existing.DailyOccurrenceCounts,
                                existing.CreatedAt,
                                existing.LastResetAt,
                                existing.LastSyncedAt),
                            eligibleToAdd);
                    }
```

```
                }).ConfigureAwait(false);
        }
    }
```

LoyaltyConfigurationQueryDTO

The **LoyaltyConfigurationQueryDTO** is obtained by calling the playAWARDS Loyalty Economy Configuration Query API and represents the current earn method rules, daily cap rules, lifetime cap rules, and initial balance rules defined by the playAWARDS economist. The **LoyaltyConfigurationQueryDTO** also provides values that dictate the number of times that an anonymous account's Loyalty balance can be merged into an authenticated account (across the set of all playAWARDS applications used by that player) and the sync frequency by which an authenticated account can add points to their Loyalty balance seen across all playAWARDS applications.

```
    /// <summary>
    /// DTO class that represents the Loyalty Economy configuration rulesets for
earn methods, daily caps, lifetime caps, initial balances, and rules for syncing
and merging accounts
    /// </summary>
    public class LoyaltyConfigurationQueryDTO
    {
        /// <summary>
        /// Specify if null values appear in json serialization
        /// </summary>
        public const NullValueHandling NULL_VALUE_HANDLING =
NullValueHandling.Ignore;

        /// <summary>
        /// Constructor
        /// </summary>
        /// <param name="id"></param>
        /// <param name="earnMethods"></param>
        /// <param name="dailyCapRules"></param>
        /// <param name="lifetimeCapRules"></param>
        /// <param name="initialBalanceRules"></param>
        /// <param name="maxAccountMerges"></param>
        /// <param name="minimumSyncAmount"></param>
        public LoyaltyConfigurationQueryDTO(string id, IDictionary<string,
IReadOnlyCollection<LoyaltyEarnRuleQueryDTO>> earnMethods,
IReadOnlyCollection<LoyaltyDailyCapRuleDTO> dailyCapRules,
IReadOnlyCollection<LoyaltyLifetimeCapRuleDTO> lifetimeCapRules,
IReadOnlyCollection<LoyaltyInitialBalanceRuleDTO> initialBalanceRules, int
maxAccountMerges, long minimumSyncAmount)
        {
            Id = id;
            EarnMethods = earnMethods;
            DailyCapRules = dailyCapRules;
            LifetimeCapRules = lifetimeCapRules;
            InitialBalanceRules = initialBalanceRules;
            MaxAccountMerges = maxAccountMerges;
```

```
            MinSyncAmount = minimumSyncAmount;
        }

        /// <summary>
        /// The unique Id of this configuration that can be used for auditing
purposes
        /// </summary>
        [JsonProperty(Required = Required.Always)]
        public string Id { get; set; }

        /// <summary>
        /// A mapping of earn method names to rules that apply to that earn method
determined by player attributes
        /// </summary>
        [JsonProperty(Required = Required.Always)]
        public IDictionary<string, IReadOnlyCollection<LoyaltyEarnRuleQueryDTO>>
EarnMethods { get; }

        /// <summary>
        /// A collection of rules for daily player caps determined by player
attributes
        /// </summary>
        [JsonProperty(Required = Required.Always)]
        public IReadOnlyCollection<LoyaltyDailyCapRuleDTO> DailyCapRules { get;
set; }

        /// <summary>
        /// A collection of rules for lifetime player caps determined by player
attributes
        /// </summary>
        [JsonProperty(Required = Required.Always)]
        public IReadOnlyCollection<LoyaltyLifetimeCapRuleDTO> LifetimeCapRules {
get; set; }

        /// <summary>
        /// A collection of rules for initializing a new player's Loyalty balance
determined by player attributes
        /// </summary>
        [JsonProperty(Required = Required.Always)]
        public IReadOnlyCollection<LoyaltyInitialBalanceRuleDTO>
InitialBalanceRules { get; set; }

        /// <summary>
        /// Specifies how many times anonymous accounts can be merged to an
Authenticated account
        /// </summary>
        [JsonProperty(Required = Required.Always)]
        public int MaxAccountMerges { get; set; }

        /// <summary>
        /// The minimum amount of Loyalty Points that should be accrued before
syncing Loyalty Points
        /// </summary>
        [JsonProperty(Required = Required.Always)]
```

```
        public long MinSyncAmount { get; set; }
    }
}
```

ID

Each Loyalty Economy configuration is given a unique **ID** (GUID) when it is created that should be logged by the `ILoyaltyRuleEngine` implementation for auditing purposes.

Earn Methods

An **earn method** is a uniquely named application event that occurs during the course of a user play session. Some examples of earn methods include:

- Coin-in to spin a slot machine
- Spinning a bonus wheel
- Logging in to the application to initiate a new play session

When an earn method trigger occurs in the server application, the server must consult the Loyalty Economy Configuration and determine the amount of Loyalty to award to the player using the best matched rule in the collection of earn rules for that earn method. Each rule consists of player attributes such as:

- **Game Level Range** (*Optional, can specify a range by minimum and/or maximum values - inclusive*)
- **Tier Range** (*Optional, can specify a range by minimum and/or maximum values - inclusive*)
- **Occurrence Count Range** (*Optional, can specify a range by minimum and/or maximum values - inclusive*). This can be used to limit Loyalty awards to the first occurrence of that event for a player on each day such as logging in to the application. The game server should implement the suggested `IPlayerLoyaltyStateRepository` to track occurrence counts of each earn method per player.
- **Platform** (*Required, one of the following values: **Any**, **iOS**, or **Android***). This can be used to award different Loyalty amounts to iOS players than Android players.
- **IdentityType** (*Required, one of the following values: **Any**, **Anon**, or **Auth***). This can be used to award different Loyalty amounts to **Anonymous** versus **Authenticated** players.

An earn method amount is further defined by the earn criteria (*Required, one of the following values: **Single** or **Multiple***). When a **Multiple** earn method is triggered, an additional parameter **N** (supplied at the point of trigger) is multiplied by the amount. The **Per** property describes what the **N** things are, such as **COININ** to a slot machine, or **GAMEPLAY_SECONDS** of a match or **AD_SECONDS** of advertisements viewed. Amounts of earn methods with **Multiple** earn criteria might be fractional.

If the player is an authenticated player, the resulting unsynced balance should immediately be synced if it exceeds the **MinSyncAmount** so that it is counted toward the Loyalty balance earned across all playAWARDS applications.

In order to return the best matched rule, the rules are ordered by **PriorityScore** and should be searched in descending order. The first rule that matches the specified criteria is used. The earn methods are defined in the `LoyaltyConfigurationQueryDTO` as a Dictionary/Map named **earnMethods** of string **LoyaltyEarnRuleQueryDTO[]**.

The priority scores of the earn method rules are unique for a given earn method, and can be logged by the `ILoyaltyRuleEngine` implementation for auditing purposes.

```
    /// <summary>
    /// DTO class that describes Loyalty earned for a specified method
    /// The player must match the player attributes described in the rule
    /// </summary>
    public class LoyaltyEarnRuleQueryDTO
    {
        /// <summary>
        ///
        /// </summary>
        /// <param name="priorityScore"></param>
        /// <param name="occurrenceMin"></param>
        /// <param name="occurrenceMax"></param>
        /// <param name="myVipTierMin"></param>
        /// <param name="myVipTierMax"></param>
        /// <param name="gameLevelMin"></param>
        /// <param name="gameLevelMax"></param>
        /// <param name="platform"></param>
        /// <param name="identityType"></param>
        /// <param name="earnCriteria"></param>
        /// <param name="per"></param>
        /// <param name="amount"></param>
        public LoyaltyEarnRuleQueryDTO(int priorityScore, int? occurrenceMin, int?
occurrenceMax, int? myVipTierMin, int? myVipTierMax, int? gameLevelMin, int?
gameLevelMax, LoyaltyPlatform platform, LoyaltyIdentityType identityType,
LoyaltyEarnCriteria earnCriteria, string per, decimal amount)
        {
            PriorityScore = priorityScore;
            OccurrenceMin = occurrenceMin;
            OccurrenceMax = occurrenceMax;
            MyVipTierMin = myVipTierMin;
            MyVipTierMax = myVipTierMax;
            GameLevelMin = gameLevelMin;
            GameLevelMax = gameLevelMax;
            Platform = platform;
            IdentityType = identityType;
            EarnCriteria = earnCriteria;
            Per = per;
            Amount = amount;
        }

        /// <summary>
        /// Unique id used to enforce rule ordering and to provide auditing
mechanism
        /// Rules are processed in descending order of PriorityScore, looking for
the first matching rule
        /// </summary>
        [JsonProperty(Required = Required.Always)]
        public int PriorityScore { get; set; }
```

```
        /// <summary>
        /// The minimum (inclusive) instance count of this event per day needed to
match this rule
        /// </summary>
        [JsonProperty(NullValueHandling =
LoyaltyConfigurationQueryDTO.NULL_VALUE_HANDLING)]
        public int? OccurrenceMin { get; set; }

        /// <summary>
        /// The maximum (inclusive) instance count of this event per day needed to
match this rule
        /// </summary>
        [JsonProperty(NullValueHandling =
LoyaltyConfigurationQueryDTO.NULL_VALUE_HANDLING)]
        public int? OccurrenceMax { get; set; }

        /// <summary>
        /// The minimum (inclusive) Tier that the player must be in to match this
rule
        /// </summary>
        [JsonProperty(NullValueHandling =
LoyaltyConfigurationQueryDTO.NULL_VALUE_HANDLING)]
        public int? MyVipTierMin { get; set; }

        /// <summary>
        /// The maximum (inclusive) Tier that the player must be in to match this
rule
        /// </summary>
        [JsonProperty(NullValueHandling =
LoyaltyConfigurationQueryDTO.NULL_VALUE_HANDLING)]
        public int? MyVipTierMax { get; set; }

        /// <summary>
        /// The minimum (inclusive) game level that the player must be in to match
this rule
        /// </summary>
        [JsonProperty(NullValueHandling =
LoyaltyConfigurationQueryDTO.NULL_VALUE_HANDLING)]
        public int? GameLevelMin { get; set; }

        /// <summary>
        /// The maximum (inclusive) game level that the player must be in to match
this rule
        /// </summary>
        [JsonProperty(NullValueHandling =
LoyaltyConfigurationQueryDTO.NULL_VALUE_HANDLING)]
        public int? GameLevelMax { get; set; }

        /// <summary>
        /// The platform that the player must be on to match this rule
        /// </summary>
        [JsonConverter(typeof(StringEnumConverter))]
        [JsonProperty(Required = Required.Always)]
        public LoyaltyPlatform Platform { get; set; }
```

```
        /// <summary>
        /// The identity type of the player (anonymous or authenticated)
        /// </summary>
        [JsonConverter(typeof(StringEnumConverter))]
        [JsonProperty(Required = Required.Always)]
        public LoyaltyIdentityType IdentityType { get; set; }

        /// <summary>
        /// The earn criteria of the earn method
        /// </summary>
        [JsonProperty(Required = Required.Always)]
        public LoyaltyEarnCriteria EarnCriteria { get; set; }

        /// <summary>
        /// The description of the item that EarnCriteria applies to
        /// </summary>
        [JsonProperty(Required = Required.Always)]
        public string Per { get; set; }

        /// <summary>
        /// The amount of Loyalty per event (may be fractional for things like
coin-in)
        /// </summary>
        [JsonProperty(Required = Required.Always)]
        public decimal Amount { get; set; }
    }
```

Daily Cap Rules

The **Daily Cap Rules** are a collection of rules that dictate how many Loyalty Points a player can earn in a single day. The game server should implement the suggested `IPlayerLoyaltyStateRepository` to track Loyalty awarded to each player on the current day. When Loyalty is awarded, daily cap conditions limit the amount that is ultimately awarded for that event. Each rule consists of player attributes:

- **Tier Range** (*Optional, can specify a range by minimum and/or maximum values - inclusive*)
- **IdentityType** (*Required, one of the following values: **Any**, **Anon**, or **Auth***). This can be used to assign different daily caps to **Anonymous** versus **Authenticated** players.

In order to return the best matched rule, the rules are ordered by **PriorityScore** and should be searched in descending order. The first rule that matches the specified criteria is used. The daily cap rules are defined in the `LoyaltyConfigurationQueryDTO` as an array of **LoyaltyDailyCapRuleDTO(s)** named **dailyCapRules**.

The priority scores of the daily cap rules are unique and can be logged by the `ILoyaltyRuleEngine` implementation for auditing purposes.

```
    /// <summary>
    /// DTO class that describes daily Loyalty cap for a particular player by
myTier and identity type
    /// </summary>
```

```
    public class LoyaltyDailyCapRuleDTO
    {
        /// <summary>
        /// Constructor
        /// </summary>
        /// <param name="priorityScore"></param>
        /// <param name="identityType"></param>
        /// <param name="myVipTierMin"></param>
        /// <param name="myVipTierMax"></param>
        /// <param name="cap"></param>
        public LoyaltyDailyCapRuleDTO(int priorityScore, LoyaltyIdentityType
identityType, int? myVipTierMin, int? myVipTierMax, long cap)
        {
            PriorityScore = priorityScore;
            IdentityType = identityType;
            MyVipTierMin = myVipTierMin;
            MyVipTierMax = myVipTierMax;
            Cap = cap;
        }

        /// <summary>
        /// Unique id used to enforce rule ordering and to provide auditing
mechanism
        /// Rules are processed in descending order of PriorityScore, looking for
the first matching rule
        /// </summary>
        [JsonProperty(Required = Required.Always)]
        public int PriorityScore { get; set; }

        /// <summary>
        /// The identity type of the player (anonymous or authenticated)
        /// </summary>
        [JsonProperty(Required = Required.Always)]
        [JsonConverter(typeof(StringEnumConverter))]
        public LoyaltyIdentityType IdentityType { get; set; }

        /// <summary>
        /// The minimum (inclusive) Tier that the player must be in to match this
rule
        /// </summary>
        [JsonProperty(NullValueHandling =
LoyaltyConfigurationQueryDTO.NULL_VALUE_HANDLING)]
        public int? MyVipTierMin { get; set; }

        /// <summary>
        /// The maximum (inclusive) Tier that the player must be in to match this
rule
        /// </summary>
        [JsonProperty(NullValueHandling =
LoyaltyConfigurationQueryDTO.NULL_VALUE_HANDLING)]
        public int? MyVipTierMax { get; set; }

        /// <summary>
        /// The maximum amount of Loyalty that the player can earn per day
```

```
        /// </summary>
        [JsonProperty(Required = Required.Always)]
        public long Cap { get; set; }
    }
```

Lifetime Cap Rules

The **Lifetime Cap Rules** are a collection of rules that dictate how many Loyalty Points a player can earn over their lifetime across all playAWARDS applications. The game server should implement the `ILoyaltyClient` in order to obtain the player's lifetime Loyalty earned. When Loyalty is awarded, lifetime cap conditions limit the amount that is ultimately awarded for that event. Each rule consists of player attributes:

- **Tier Range** (*Optional, can specify a range by minimum and/or maximum values - inclusive*)
- **IdentityType** (*Required, one of the following values: **Any**, **Anon**, or **Auth***). This can be used to assign different lifetime caps to **Anonymous** versus **Authenticated** players.

In order to return the best matched rule, the rules are ordered by **PriorityScore** and should be searched in descending order. The first rule that matches the specified criteria is used. The lifetime cap rules are defined in the `LoyaltyConfigurationQueryDTO` as an array of **LoyaltyLifetimeCapRuleDTO(s)** named **lifetimeCapRules**.

The priority scores of the lifetime cap rules are unique and can be logged by the `ILoyaltyRuleEngine` implementation for auditing purposes.

```
    /// <summary>
    /// DTO class that describes lifetime Loyalty cap for a particular player by
Tier and identity type
    /// </summary>
    public class LoyaltyLifetimeCapRuleDTO
    {
        /// <summary>
        /// Constructor
        /// </summary>
        /// <param name="priorityScore"></param>
        /// <param name="identityType"></param>
        /// <param name="myVipTierMin"></param>
        /// <param name="myVipTierMax"></param>
        /// <param name="cap"></param>
        public LoyaltyLifetimeCapRuleDTO(int priorityScore, LoyaltyIdentityType
identityType, int? myVipTierMin, int? myVipTierMax, long cap)
        {
            PriorityScore = priorityScore;
            IdentityType = identityType;
            MyVipTierMin = myVipTierMin;
            MyVipTierMax = myVipTierMax;
            Cap = cap;
        }

        /// <summary>
        /// Unique id used to enforce rule ordering and to provide auditing
```

```
mechanism
        /// Rules are processed in ascending order of RuleId, looking for the
first matching rule
        /// </summary>
        [JsonProperty(Required = Required.Always)]
        public int PriorityScore { get; set; }

        /// <summary>
        /// The identity type of the player (anonymous or authenticated)
        /// </summary>
        [JsonProperty(Required = Required.Always)]
        [JsonConverter(typeof(StringEnumConverter))]
        public LoyaltyIdentityType IdentityType { get; set; }

        /// <summary>
        /// The minimum (inclusive) Tier that the player must be in to match this
rule
        /// </summary>
        [JsonProperty(NullValueHandling =
LoyaltyConfigurationQueryDTO.NULL_VALUE_HANDLING)]
        public int? MyVipTierMin { get; set; }

        /// <summary>
        /// The maximum (inclusive) Tier that the player must be in to match this
rule
        /// </summary>
        [JsonProperty(NullValueHandling =
LoyaltyConfigurationQueryDTO.NULL_VALUE_HANDLING)]
        public int? MyVipTierMax { get; set; }

        /// <summary>
        /// The maximum amount of Loyalty that the player can earn over their
lifetime
        /// </summary>
        [JsonProperty(Required = Required.Always)]
        public long Cap { get; set; }
    }
```

Initial Balance Rules

The **Initial Balance Rules** are a collection of rules that set the initial Loyalty balance of a new player of the partner's application. The game server should implement the suggested `IPlayerLoyaltyStateRepository` to create the Loyalty state for a new player with the specified balance. If the player is an authenticated player, the initial balance should immediately be synced so that it is counted toward the Loyalty balance earned across all playAWARDS applications. Each rule consists of player attributes:

- **Tier Range** (*Optional, can specify a range by minimum and/or maximum values - inclusive*)
- **IdentityType** (*Required, one of the following values: **Any**, **Anon**, or **Auth***). This can be used to assign different initial balances to **Anonymous** versus **Authenticated** players.

In order to return the best matched rule, the rules are ordered by **PriorityScore** and should be searched in descending order. The first rule that matches the specified criteria is used. The lifetime cap rules are defined in the `LoyaltyConfigurationQueryDTO` as an array of **LoyaltyInitialBalanceRuleDTO(s)** named **initialBalanceRules**.

The priority scores of the initial balance rules are unique and can be logged by the `ILoyaltyRuleEngine` implementation for auditing purposes.

```
    /// <summary>
    /// DTO class that describes daily Loyalty cap for a particular player by Tier
and identity type
    /// </summary>
    public class LoyaltyDailyCapRuleDTO
    {
        /// <summary>
        /// Constructor
        /// </summary>
        /// <param name="priorityScore"></param>
        /// <param name="identityType"></param>
        /// <param name="myVipTierMin"></param>
        /// <param name="myVipTierMax"></param>
        /// <param name="cap"></param>
        public LoyaltyDailyCapRuleDTO(int priorityScore, LoyaltyIdentityType
identityType, int? myVipTierMin, int? myVipTierMax, long cap)
        {
            PriorityScore = priorityScore;
            IdentityType = identityType;
            MyVipTierMin = myVipTierMin;
            MyVipTierMax = myVipTierMax;
            Cap = cap;
        }

        /// <summary>
        /// Unique id used to enforce rule ordering and to provide auditing
mechanism
        /// Rules are processed in descending order of PriorityScore, looking for
the first matching rule
        /// </summary>
        [JsonProperty(Required = Required.Always)]
        public int PriorityScore { get; set; }

        /// <summary>
        /// The identity type of the player (anonymous or authenticated)
        /// </summary>
        [JsonProperty(Required = Required.Always)]
        [JsonConverter(typeof(StringEnumConverter))]
        public LoyaltyIdentityType IdentityType { get; set; }

        /// <summary>
        /// The minimum (inclusive) Tier that the player must be in to match this
rule
        /// </summary>
```

```
        [JsonProperty(NullValueHandling =
LoyaltyConfigurationQueryDTO.NULL_VALUE_HANDLING)]
        public int? MyVipTierMin { get; set; }

        /// <summary>
        /// The maximum (inclusive) Tier that the player must be in to match this
rule
        /// </summary>
        [JsonProperty(NullValueHandling =
LoyaltyConfigurationQueryDTO.NULL_VALUE_HANDLING)]
        public int? MyVipTierMax { get; set; }

        /// <summary>
        /// The maximum amount of Loyalty that the player can earn per day
        /// </summary>
        [JsonProperty(Required = Required.Always)]
        public long Cap { get; set; }
    }
```

MaxAccountMerges

The `MaxAccountMerges` property specifies a limit of the amount of times that an anonymous player can merge their anonymous Loyalty balance (kept strictly in the game server's `IPlayerLoyaltyStateRepository`) with that of an authenticated user. The authenticated user might have an existing Loyalty balance across other playAWARDS applications, and might have already merged other anonymous accounts. The game server should implement the `ILoyaltyClient` in order to obtain the current number of times that an authenticated player has previously merged anonymous accounts.

The game server should implement the suggested `IPlayerLoyaltyStateRepository` to track the anonymous Loyalty balance and when a merge operation occurs, transfer this amount to the unsynced authenticated Loyalty balance and then immediately do a sync operation so that it is counted toward the Loyalty balance earned across all playAWARDS applications.

MinSyncAmount

The `MinSyncAmount` property specifies the minimum amount of Loyalty that must be accumulated by an authenticated player in their `IPlayerLoyaltyStateRepository` before a sync operation can be performed. This is effectively a throttle on how often the game server can update the player's Loyalty balance seen across all playAWARDS applications. The sync operation should check this condition before proceeding with the `ILoyaltyClient` call to add Loyalty Points.

LoyaltyPlatform Enum

```
    //
    // Summary:
    //     Enumeration type for each possible platform
    [JsonConverter(typeof(StringEnumConverter))]
    public enum LoyaltyPlatform
    {
```

```
        //
        // Summary:
        //     Any Platform
        Any = 0,
        //
        // Summary:
        //     Android Platform
        Android = 1,
        //
        // Summary:
        //     iOS Platform
        iOS = 2,
        //
        // Summary:
        //     Web Platform
        Web = 3
    }
```

LoyaltyIdentityType Enum

```
    //
    // Summary:
    //     Enumeration type for a player's identity type. Different rules and
policies may
    //     be specified for Anonymous vs Authenticated players
    [JsonConverter(typeof(StringEnumConverter))]
    public enum LoyaltyIdentityType
    {
        //
        // Summary:
        //     Any Players
        Any = 0,
        //
        // Summary:
        //     Anonymous Players
        Anon = 1,
        //
        // Summary:
        //     Authenticated Players
        Auth = 2
    }
```

LoyaltyEarnCriteria Enum

```
    //
    // Summary:
    //     Enumeration type for an earn method's earn criteria. Earn rules can
apply to
    //     a single amount (EARN_METHOD_LOGIN) or a multiple amount
```

```
(EARN_METHOD_COININ)
    //     Multiple signals that the EARN_METHOD's loyaltyAmount should be
multiplied by
    //     a number N (for example, the total number of coins at the time the
EARN_METHOD_COININ
    //     method is triggered)
    [JsonConverter(typeof(StringEnumConverter))]
    public enum LoyaltyEarnCriteria
    {
        //
        // Summary:
        //     The Loyalty amount applies to a single event
        Single = 0,
        //
        // Summary:
        //     The Loyalty amount should be multiplied by a number N that is
available at the
        //     time the earn method is triggered
        Multiple = 1
    }
```

Sample LoyaltyConfigurationQueryDTO


```
{
  "id": "a3aa43ab-9269-404e-abdf-cabe9166f807",
  "earnMethods": {
    "EARN_METHOD_BONUSWHEEL": [
      {
        "priorityScore": 1,
        "occurrenceMin": 1,
        "occurrenceMax": 1,
        "gameLevelMax": 99,
        "platform": "Any",
        "identityType": "Any",
        "earnCriteria": "Single",
        "per": "SPIN",
        "amount": 10
      },
      {
        "priorityScore": 0,
        "occurrenceMin": 1,
        "occurrenceMax": 1,
        "gameLevelMin": 100,
        "platform": "Any",
        "identityType": "Any",
        "earnCriteria": "Single",
        "per": "SPIN",
        "amount": 20
      }
    ],
    "EARN_METHOD_COININ": [
```

```
        {
          "priorityScore": 4,
          "myVipTierMin": 1,
          "myVipTierMax": 2,
          "platform": "Any",
          "identityType": "Any",
          "earnCriteria": "Multiple",
          "per": "COININ",
          "amount": 0.00001
        },
        {
          "priorityScore": 3,
          "myVipTierMin": 3,
          "myVipTierMax": 4,
          "platform": "Any",
          "identityType": "Any",
          "earnCriteria": "Multiple",
          "per": "COININ",
          "amount": 0.00003
        },
        {
          "priorityScore": 2,
          "myVipTierMin": 5,
          "myVipTierMax": 6,
          "platform": "Any",
          "identityType": "Any",
          "earnCriteria": "Multiple",
          "per": "COININ",
          "amount": 0.00005
        },
        {
          "priorityScore": 1,
          "myVipTierMin": 7,
          "myVipTierMax": 8,
          "platform": "Any",
          "identityType": "Any",
          "earnCriteria": "Multiple",
          "per": "COININ",
          "amount": 0.00007
        },
        {
          "priorityScore": 0,
          "myVipTierMin": 9,
          "platform": "Any",
          "identityType": "Any",
          "earnCriteria": "Multiple",
          "per": "COININ",
          "amount": 0.00009
        }
      ],
      "EARN_METHOD_LOGIN": [
        {
          "priorityScore": 1,
          "occurrenceMin": 1,
```

```
          "occurrenceMax": 1,
          "platform": "Any",
          "identityType": "Auth",
          "earnCriteria": "Single",
          "per": "LOGIN",
          "amount": 50
        },
        {
          "priorityScore": 0,
          "occurrenceMin": 1,
          "occurrenceMax": 1,
          "platform": "Any",
          "identityType": "Any",
          "earnCriteria": "Single",
          "per": "LOGIN",
          "amount": 25
        }
      ]
    },
    "dailyCapRules": [
      {
        "priorityScore": 2,
        "identityType": "Auth",
        "myVipTierMin": 1,
        "myVipTierMax": 6,
        "cap": 5000
      },
      {
        "priorityScore": 1,
        "identityType": "Auth",
        "myVipTierMin": 7,
        "cap": 10000
      },
      {
        "priorityScore": 0,
        "identityType": "Any",
        "cap": 1000
      }
    ],
    "lifetimeCapRules": [
      {
        "priorityScore": 2,
        "identityType": "Auth",
        "cap": 1000000000
      },
      {
        "priorityScore": 1,
        "identityType": "Anon",
        "cap": 1000000
      },
      {
        "priorityScore": 0,
        "identityType": "Any",
        "cap": 1000000
```

```
      }
    ],
    "initialBalanceRules": [
      {
        "priorityScore": 2,
        "identityType": "Auth",
        "initialBalance": 1000
      },
      {
        "priorityScore": 1,
        "identityType": "Anon",
        "initialBalance": 0
      },
      {
        "priorityScore": 0,
        "identityType": "Any",
        "initialBalance": 0
      }
    ],
    "maxAccountMerges": 10,
    "minSyncAmount": 1000
  }
```