Service Overview: Presence API

Introduction

The **Presence API** is a high-availability, low-latency service designed to bridge the gap between static friend lists and active gameplay sessions. While the *Friends API* manages the persistent social graph (the "who"), the Presence API manages the real-time state (the "what" and "where").

By integrating this service, game teams enable players to visualize their social circle's activity, facilitate direct communication, and drive player engagement through seamless session "join-in" functionality.

Core Functionality

The Presence service is built on a publish-subscribe architecture that handles four primary domains:

Feature	Description	Dependency
Status Tracking	Monitors if a user is Online, Offline, Away, or "In-Game."	Identity Core
Rich Presence	Metadata describing the player's specific state (e.g., "In Lobby," "Level 42 Mage").	Game Client
Messaging	Real-time XMPP-based text routing between authenticated users.	Notification Service
Invitations	Secure handshake protocol to pass session IDs between clients.	Matchmaking API

Technical Implementation

1. State Management

Presence is ephemeral. When a client initializes a heartbeat with the gateway, the service updates the distributed cache (Redis) with the user's current `AvailabilityStatus`.

- Heartbeat Interval: 30 seconds.

- **TTL (Time-to-Live):** 90 seconds (automatic "Offline" transition upon heartbeat failure).

2. Rich Presence Data (RPD)

Developers can push custom JSON blobs to the RPD field. This allows other players to see context-specific information in their UI.

```
JSON
{
  "userId": "1002938475",
  "status": "ONLINE",
  "richPresence": {
    "gameId": "Project_Atlas",
    "state": "EXPLORING",
    "map": "Hinterlands",
    "partySize": 3,
    "maxPartySize": 4,
    "joinable": true
  }
}
```

3. Invitation Flow: "The Join-In"

The Presence API facilitates the **Invite-to-Play** flow by acting as the secure transport for Session URIs.

1. **Originator:** Calls POST /v1/invite/send with the target friendId and sessionId.
2. **Service:** Validates the friendship via the *Friends API*.
3. **Recipient:** Receives a push notification containing the inviteToken.
4. **Acceptance:** The recipient client resolves the inviteToken to the sessionId and triggers the game engine's network join logic.

API Endpoints (Internal)

GET /v1/users/{userId}/presence

Retrieves the real-time status and Rich Presence data for a specific user. Use this for populating Social Sidebars.

POST /v1/users/me/status

Updates the authenticated user's status.

- **Payload:** {"status": "BUSY", "message": "Boss Fight - Do Not Disturb"}

POST /v1/messages/send

Routes a peer-to-peer message.

- **Rate Limit:** 10 messages per 5-second window to prevent spamming.

Best Practices for Game Teams

- **Batching:** When rendering a friends list of 100+ people, do not call the Presence API for each user. Use the POST /v1/presence/batch endpoint to retrieve statuses in a single round-trip.
- **Privacy:** Always respect the PrivacySettings flag. If a user is in "Invisible" mode, the API will return OFFLINE regardless of their active socket connection.

Note to Developers: To request a higher throughput quota for seasonal events or beta launches, please submit a ticket via the EAX-Infrastructure portal under the "Scaling Request" category.

Integration Guide: Real-Time Invitations

1. Architectural Overview

To ensure a seamless player experience, the invitation flow must handle three distinct states: **Sending**, **Receiving (Toast)**, and **Executing (Join)**.

2. Implementation: Sending an Invite

When a player selects a friend from the social UI, the game client must package the current session's connection metadata into an invitePayload.

C++ Interface Example:

```
C++
// Define the invitation payload
FPresenceInvitePayload InviteData;
InviteData.SessionId = CurrentGameSession->GetId();
InviteData.MapName = "Hinterlands_PVP";
InviteData.JoinString = "-connect 192.168.1.1:7777"; // Internal routing
string

// Execute the send via Presence Service Wrapper
PresenceService->SendInvite(TargetFriendId, InviteData,
    FOnInviteSent::CreateLambda([](bool bSuccess) {
        if (bSuccess) {
            UI->ShowNotification("Invite Sent!");
        }
    })
);
```

3. Implementation: Handling Incoming Invites

The client must register a listener with the **Presence Gateway** to intercept incoming invite packets. This usually occurs at the GameInstance level to ensure persistence across level loads.

A. The Notification (Toast)

When the OnInviteReceived delegate fires, the game should display a non-intrusive UI element.

- **Key Data:** The payload contains the SenderName and RichPresence metadata (e.g., "In a 3/4 Party").
- **Action:** Provide two buttons: **Accept** and **Decline**.

B. The Join Logic

If the user accepts, the game must transition from the current state (Main Menu or active game) to the new session.

```
C++
void UMyGameInstance::HandleInviteAccepted(const FPresenceInvitePayload&
Payload) {
    // 1. Teardown current session
    SessionManager->DestroyCurrentSession();

    // 2. Parse the JoinString from the Presence Payload
    FString ConnectionURL = Payload.JoinString;

    // 3. Initiate the Engine Travel
    // This triggers the internal 'map travel' to the remote host
    GetWorld()->GetFirstPlayerController()->ClientTravel(ConnectionURL,
TRAVEL_Absolute);
}
```

4. Technical Constraints & Edge Cases

Scenario	Recommended Handling
Incompatible Version	Compare Payload.BuildId with local BuildId. If they mismatch, block the join and prompt the user to update.
Session Full	Query the Matchmaking API using the SessionId before attempting to travel.
Player Busy	If the player is in a cinematic or un-pausable state, queue the invite in a "Notifications" tray rather than showing a toast.