

LPS 3rd Party Integration Guide

Live Play Mobile

Live Play Service Integration Guide

Author: Fred Ring

fred.r@liveplaymobile.com

This document is intended for a technical / business audience looking to integrate with LPS. It contains all the information a developer requires to perform a full integration with LPS.

Revisions

When	Who	What
Jul 30, 2025	FR	Initial document creation
Jul 31, 2025	FR	1st pass internal review
Aug 21, 2025	FR	Updating JSON for set_kpi and adding more details about its parameters
Sep 21, 2025	FR	Adding details for streaming/chat widget. app_id → game_id
Oct 20, 2025	EJ	Adding details for Name generation and stream frame settings
Oct 24, 2025	EJ	Updating Video iFrame messages
Nov 3, 2025	EJ	Add MUTE_AUDIO and UNMUTE_AUDIO iFrame messages
Nov 6, 2025	EJ	Add SET_CHAT_STYLE_SETTINGS

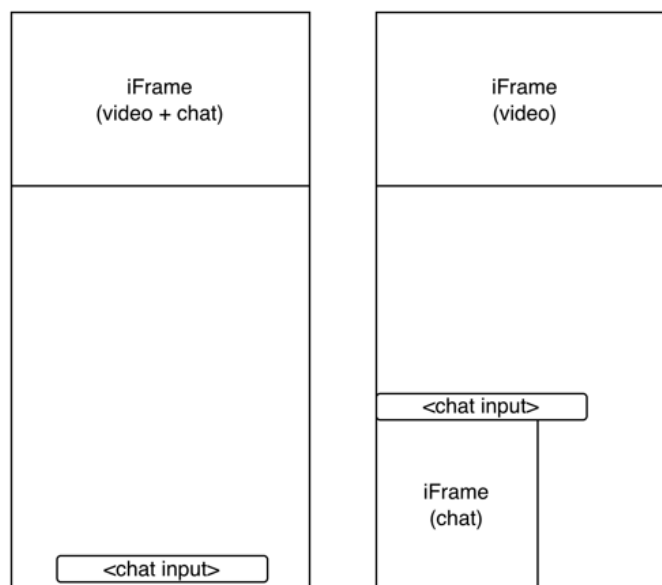
Nov 13, 2025	EJ	Add PING-PONG messages + update in Creating the Video Stream iFrame
Jun 24, 2026	FR	Updated with 006 features including chroma key transparency and chat separation from the streaming frame, including chat styling

Overview

Live Play Mobile's LPS provides a live-hosted video stream, with social chat, that can be integrated into an application. To view the video streams and interact with chat LPS provides its *Streaming Service*. This is how your users will experience LPS hosts and the community. To drive those experiences you must provide a close-to-real-time data feed from your backend with details about game play. This allows LPS to drive both human and AI hosts and create a sense of real time community play.

Your Application

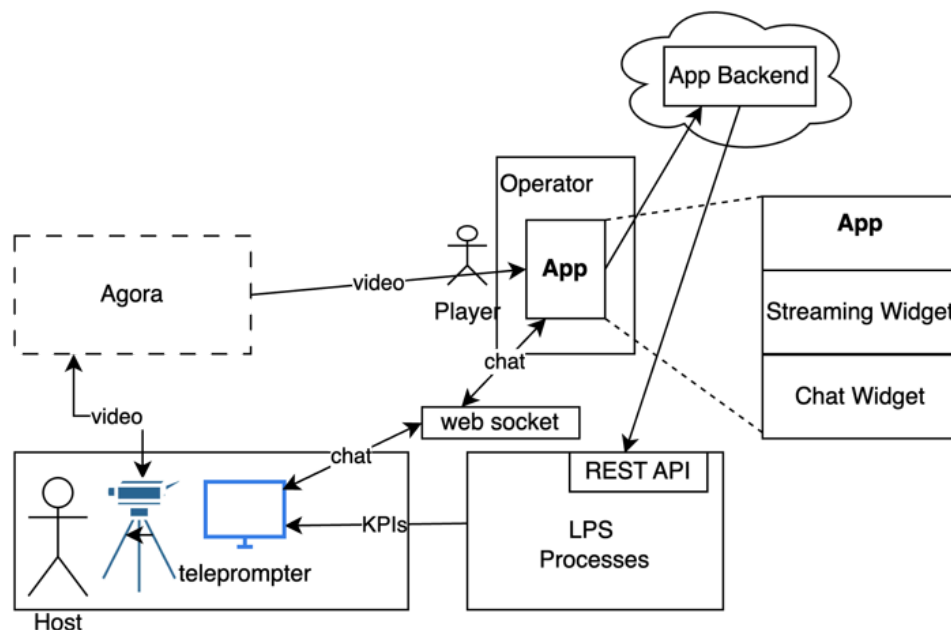
Integration with LPS is easy. It revolves around the use of iframes. You can put the entire LPS into a single frame or split it into two, with the video and chat appearing in separate frames. When loading the iframe you can specify how the iframe will operate. By default the URL will load a combined video and chat UI. To separate them add stream-only or chat-only as a URL parameter. You can configure some styling elements using a message the iframe listens for - details on how to do that are below.



If you want your users to be able to send chat messages then in addition to the iframe(s) you must provide a way to get those messages from the user and send it. Typically this is a simple text input field, but you can take any approach you like to achieve your chat input goals. Sending chat messages involves sending a message to the iframe containing the LPS chat widget

The Bigger Picture

Below shows how your application plugs into the LPS ecosystem. You can see your application and also the backend that supports your application. This is also an integration point - your backend talks to the LPS backend. That communication contains close to real time game play metrics sent for each player. This is used by both human and AI hosts with information to create the shared, communal experience.



• REST API (Backend To Backend)

Your backend will send application-specific metrics, as games are played, to the LPS backend. These metrics should be sent close to real time. As metrics arrive they are processed to create data feeds that either a human or AI host can use to create an engaging experience for players. You will work with LPS to create this metric connection between the backends.

Note that the REST API is intended for backend to backend communication and your deployed applications should not directly communicate with LPS. You will receive an API Key that must be

included in the headers with each request.

• **Streaming Background**

Some streams are delivered with a solid, rectangular, background and some are provided with no background so the host appears to be floating inside your application. Talk to your LPS producer to understand the visual format of the streams your application will receive.

• **Chat Input**

LPS chat displays the chat messages, but does not include an input field. The application must implement the input field and send messages to the iframe to transmit chat messages for the player. To transmit a chat message post a message to the iframe hosting this widget.

Implementation Steps

1. Work with LPS to develop backend-to-backend game data feed for your application
2. Obtain an *operator_id* and *game_id* from LPS
3. Insert an iFrame into your application to view the stream and chat
 - You can also load chat in a separate iFrame, allowing you to place it separately from the video stream, so you would have 2 iFrames in that case.
4. Load the LPS-provided URL into the frame(s)
5. Obtain a token from LPS to access the stream and pass it into the iFrame(s)

There might be one more step to add to that list - support for giving rewards to players. Work with your LPS producer to determine if that also needs to be implemented.

Backend to Backend Game Metrics

There is just a single REST POST endpoint that accepts game play data. We will work with you to develop a parser for the data structures you send to the endpoint. The endpoint accepts JSON as the request body. Received data is queued for processing so the response is indicative only of the reception of the data so errors sending data represent a communication issue and not a data-content issue.

All calls must come from your application's backend. Applications on player devices should not connect to this API. An API key is issued to you and must accompany all POST requests as an HTTPS header. As API keys are sensitive data you cannot make them publicly available and

they need to stay on your backend.

Required Root Fields

Calls are required to include all of the following in their JSON body root. You will receive these from LPS during your integration process.

game_id: *string* Provided by LPS. This is a UUID permanently assigned to a specific game from a game developer. It does not change with game versions or deployments.

operator_id: *string* Provided by LPS. This is a UUID permanently assigned to a specific operator. It does not change.

game_version: *string* version string the app uses internally. Different versions can have slightly different LPS processing. In most cases you can set this to “1”. If there is ever a need for different event processing based on the *game_version* then this is how we would facilitate.

operator_player_id: *string* Obtained by the app from the operator. This is a unique value on the operator’s platform that identifies the individual playing the app. It is expected to be a universally unique value.

event_type: *string* valid values are the result of working with LPS engineering to create this data interface. This tells the LPS processor how to interpret the message.

event: *JSON* object containing all the data needed to process the event.

Optional Root Fields

happened_on: *number optional* Epoch time when the event happened, in seconds. If you guarantee to never send the same message more than once then you can ignore this and not include it in your messages.

Here is a pseudocode JSON sample...

```
1 {  
2   "game_id":"<LPS supplies this>",  
3   "game_version":"1.0.0",  
4   "operator_id":"<LPS supplies this>",  
5   "happened_on":"<epoch in seconds>",  
6   "operator_player_id":"<app gets from operator platform>",  
7   "event_type":"<type>",  
8   "event":{event payload}  
9 }
```

What are valid values for **event_type**? What does an **event** look like? A big part of LPS integration is working with LPS so we understand the incoming events and can parse them into usable data. You will need to contact your LPM producer to start that process. Our technical team builds processors to extract useful data from these events. The building of those processors can only happen with a full meeting of the minds between the application's backend developers and LPS.

Avoiding Event Duplication (Idempotency)

If your backend guarantees that the same message will never be sent more than once then you can ignore this entire section on duplicates.

5 Minute Window

When a message is received, the knowledge of that reception lasts for 5 minutes. So if you send the exact same message again, within that 5-minute window, the repeated message is ignored. But if you send the exact same message again, more than 5 minutes after the original, we no longer detect the duplicate and will record the 2nd message as unique.

Happened On

If you cannot guarantee sending every event only once then you should include **happened_on**. This value should be an approximate epoch timestamp of when the event actually happened and not the time of transmission to LPS. The most important part of avoiding KPI duplication is smart use of the **happened_on** timestamp value.

Because **happened_on** is part of the message, changing its value changes the message. So if we try to use the time of transmission instead of the time when the event happened then duplicates will appear and we do not want that.

You can best avoid this problem by setting **happened_on** to a timestamp when the event is generated and not at the time of transmission. That way, when your transmitter gets the message to send it is in the best position to avoid duplicates because you will not be sending the same event with a different timestamp.

Streaming Video Transparency

If the stream contains a chroma key value then that color will be removed and replaced with transparency so the host will appear on top of whatever background the iframe is over. Streams can also contain a full-frame background, depending on the content in the stream. In those cases the frame will be filled with something instead of transparency.

Postmate

Iframe communication is implemented with [Postmate](#) (available on github). Postmate adds a useful layer of logic to the bare messages that makes them easier to use.

Getting an Access Token & Accessing the Stream

Access tokens are specific to a player/game/operator so all three are required when requesting an access token. Get the *player_id* from the operator platform. LPS provides *operator_id* and *game_id*. You will also have multiple API Keys from LPS. You will need to include your **Access Token API Key** as an *x-api-key* header.

We provide a REST GET endpoint that provides access tokens. It looks like the one below.

`https://<endpoint>/core/widget-token?player_id=<pid>&game_id=<game_id>&operator_id=<op_id>`

Parse the response as JSON and you will find an access token.

```
1 response = await fetch(<url>, {method: 'GET', headers:{'x-api-key':  
  <your API Key>}})  
2 data = await response.json()  
3 accessToken = data.token
```

After you have the token you pass it to the iframe(s) using Postmate. Your web document might look something like the following where we can see both the setting of the access token and a ping/pong we can use to verify that communications are getting through.

```
1 <div id="uniqueIframeId"></div>  
2  
3 const handshake = new Postmate({  
4   container: document.getElementById('uniqueIframeId'),  
5   url: iframeHostedUrl  
6 });  
7  
8 let childRef;  
9 handshake.then(child => {  
10   childRef = child  
11   child.on('PONG', (data) => {  
12     console.log('Parent received message:', data);  
13   });  
14 }  
15  
16 function pingChild() {  
17   childRef.call('sendMessageToChild', {"type": "PING"});
```

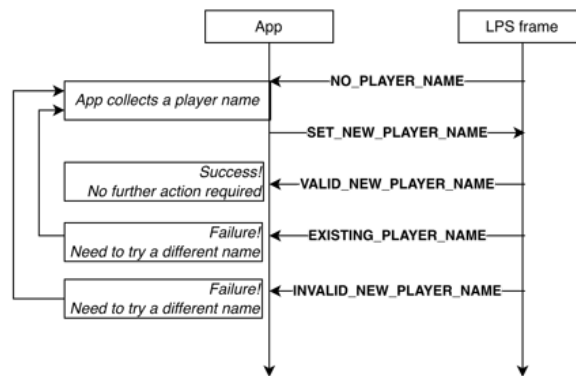
```

18 }
19
20 function setAccessToken(token){
21     childRef.call('sendMessageToChild', {"type": "ACCESS_TOKEN",
22     "payload":token});
22 }

```

Player Names

All LPS players are given a player name. The 1st time a player loads a game, LPS will verify that they have a player name and, if not, will send a message requesting that your app run a player name input process. After your app collects a player name, it submits it to LPS for validation. Not all player names are acceptable. The flow of messages is show below.



The UI to collect a player name is up to your application. LPS will remember the player's name so it has to be set only once. You are free to change the player's name at any time in the future,

Loading the iFrame(s).

URL

The current URL to load LPS into your iframe is

<https://stream.develop.lpsdevelop.com/0.0.6/index.html>

It remains the same for all environments. You can see a version number 0.0.6 in there. As of this writing that is the current version of LPS. Use that to as the base URL to load your iframe(s).

iFrame Modes

- **full** is the default. In this frame both the video and chat will appear.
- **?stream-only** add this to the base URL to see only the stream in the iframe.
- **?chat-only** add this to the base URL to see only chat in the iframe.

So if you want just a single iframe with both...

```
1 <iframe src="https://stream.develop.lpsdevelop.com/0.0.6/index.html">
  </iframe>
```

If you want to split chat and stream ...

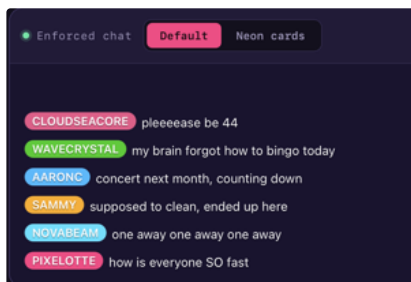
```
1 <iframe src="https://stream.develop.lpsdevelop.com/0.0.6/index.html?
  stream-only"></iframe>
2 <iframe src="https://stream.develop.lpsdevelop.com/0.0.6/index.html?
  chat-only"></iframe>
```

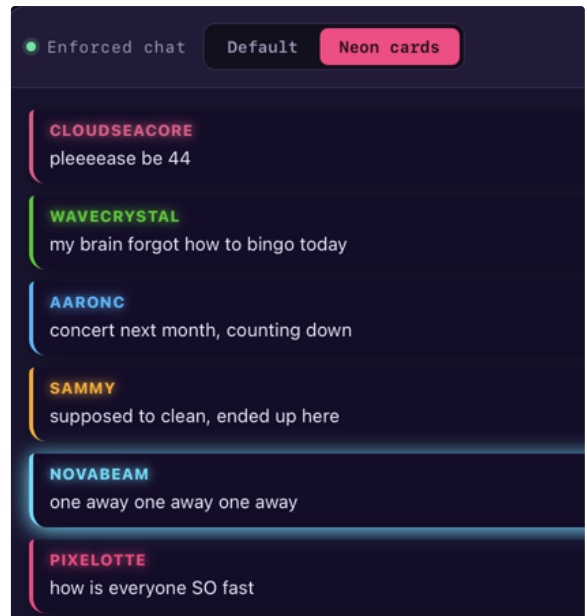
Chat Styling

To control the styling of the chat you can send raw CSS via the SET_CHAT_CSS message. Because of the cascade ordering, your CSS will override any CSS that was loaded from the origin. Here is an example.

```
1 chatChildRef.call('sendMessageToChild', {
2   type: 'SET_CHAT_CSS',
3   payload: `
4     .lps-chat__name { background: hotpink; border-radius: 4px; }
5     .lps-chat__text { font-style: italic; }
6   `,
7 });
```

Here is a chat displayed with the default CSS and then after sending custom CSS via SET_CHAT_CSS.





Target these classes. They are a stable public API.

PUBLIC CLASS	ELEMENT
.lps-chat	chat container
.lps-chat__list	the scrolling message list
.lps-chat__row	a single message row
.lps-chat__name	the player-name element
.lps-chat__text	the message body

User Name Color

Each row exposes the widget's auto-assigned player color as the CSS variable `--lps-name-color`.

Animations

Each new message is a fresh keyed DOM node, as a result an animation on `.lps-chat__row` fires per new message automatically so you can add animations to your chat CSS.

```
1 @keyframes lps-msg-in {
2   from { opacity: 0; transform: translateY(10px) scale(.98); }
3   to   { opacity: 1; transform: none; }
4 }
5 .lps-chat__row { animation: lps-msg-in .28s cubic-bezier(.22,1,.36,1);
6 }
```

Sending Messages

Following are the messages you can send to the iFrame(s) using Postmate.

PING

Simple message without payload that will respond with PONG message.

```
{“type”: “PING”}
```

ACCESS_TOKEN

After obtaining an access token you will need to send it to the iFrame.

```
{“type”: “ACCESS_TOKEN”, “payload”: <accessToken>}
```

SET_NEW_PLAYER_NAME

This is how you can set a human-readable name for the currently connected player.

```
{“type”: “SET_NEW_PLAYER_NAME”, “payload”: <player name>}
```

MUTE_AUDIO

Message with empty payload to mute video or stream audio. This is a global setting from parent app and only sending **UNMUTE_AUDIO** message will unmute audio in iFrame if **MUTE_AUDIO** was sent to iFrame.

```
{“type”: “MUTE_AUDIO”}
```

UNMUTE_AUDIO

Message with empty payload to unmute video or stream audio - works only if **MUTE_AUDIO** was sent already.

```
{“type”: “UNMUTE_AUDIO”}
```

CHAT_MESSAGE

Sends a text message to the chat accompanying the stream.

```
{“type”: “CHAT_MESSAGE”, “payload”: <You message here>}
```

SHOW_CHAT

Show the chat UI inside the iFrame. Empty payload.

```
{"type": "SHOW_CHAT"}
```

HIDE_CHAT

Hide the chat UI inside the iFrame. Chat keeps running in the background; this only affects display. `{"type": "HIDE_CHAT"}`

SET_VOLUME

Set the host audio volume. Payload is a number between 0 (silent) and 1 (full volume); values outside that range are clamped.

```
{"type": "SET_VOLUME", "payload": <0.0 - 1.0>}
```

SET_CHAT_CSS

Override the iFrame's default CSS. Payload is any valid CSS, as a string.

```
{"type": "SET_CHAT_CSS", "payload": <css string>}
```

Receiving Messages

Here are the messages you can expect to receive from the iFrame. Because communication uses Postmate, you register listeners on the child object returned by the handshake — not with raw `window.addEventListener('message', ...)`. Raw listeners will receive Postmate's protocol envelope rather than the typed events listed below.

```
const handshake = new Postmate({
  container: document.getElementById('uniqueIframeId'),
  url: iframeHostedUrl
});

handshake.then(child => {
  child.on('MEDIA_CONNECTED', (data) => {
    // data.type === 'MEDIA_CONNECTED'
    // data.payload === <payload, if any>
    ...
  });
  child.on('CHAT_MESSAGE_RECEIVED', (data) => { ... });
});
```

Each handler receives a single data argument shaped *{type, payload?}*, matching the JSON examples below.

INVALID_ACCESS_TOKEN

Received after an ACCESS_TOKEN message was found to contain an invalid access token in its payload field.

PONG

Simple message without payload that is a response for the **PING** message.

```
{"type": "PONG"}
```

MEDIA_CONNECTING

Received when the video stream is trying to connect to its source to begin streaming

```
{"type": "MEDIA_CONNECTING"}
```

MEDIA_CONNECTED

The next step after MEDIA_CONNECTING, when a stream starts.

```
{"type": "MEDIA_CONNECTED"}
```

WEBSOCKET_CONNECTING

Received when the chat module is trying to connect to LPS websocket.

```
{"type": "WEBSOCKET_CONNECTING"}
```

WEBSOCKET_CONNECTED

The next step after WEBSOCKET_CONNECTING — fired when the chat WebSocket connection has been established and the widget is ready to send and receive chat messages.

The payload contains a *stream_frame_settings* dictionary with the server's per-room limits that the parent app should honor when constructing input fields or chat messages.

```
{
```

```
  "type": "WEBSOCKET_CONNECTED",
```

```
  "payload": {
```

```
    "stream_frame_settings": {
```

```
      "player_name_minimum_characters": <minimum length for SET_NEW_PLAYER_NAME>,
```

```
      "player_name_maximum_characters": <maximum length for SET_NEW_PLAYER_NAME>,
```

"chat_maximum_characters": <maximum length for CHAT_MESSAGE>

}}}

NO_PLAYER_NAME

Received when LPS detects that it has never been told the player's name. LPS wants to know!

{ "type": "NO_PLAYER_NAME", "payload": { "suggested_player_name": <player name> } }

EXISTING_PLAYER_NAME

A response to a SET_NEW_PLAYER_NAME that failed because the name belongs to another player.

{ "type": "EXISTING_PLAYER_NAME", "payload": { "player_name": <player_name> } }

VALID_NEW_PLAYER_NAME

A response to a successful SET_NEW_PLAYER_NAME.

{ "type": "VALID_NEW_PLAYER_NAME", "payload": { "player_name": <player_name> } }

INVALID_NEW_PLAYER_NAME

A response to a failed SET_NEW_PLAYER_NAME. The payload.reason tells you why.

{ "type": "INVALID_NEW_PLAYER_NAME", "payload": { "reason": <message> }, "player_name": <player_name> } }