VP Spaces API Documentation

Author: Josh S. Sakweli, Backend Lead Team **Last Updated**: 2026-08-20 **Version**: v1.0

Base URL: `http://localhost:8765/api/v1` (local) — `server.port=8765`

Short Description: A Space is a **live audio conversation with an audience**. A small stage of speakers talks over WebRTC; everyone else listens over HLS through the CDN. It is the only feature in the platform that is two-way at the centre and one-way at the edge, and every design decision below falls out of that one fact.

Hints:

- Every endpoint needs `Authorization: Bearer <token>`.
- **Which endpoints return a token *is* the cost model.** Opening the stage and being promoted return one; joining as a listener never does.
- **There is no "go live" call.** SRS decides `LIVE` when the mixed audio actually reaches it. A client never sets state.
- `playbackUrl` is **null until media flows**. Handing it out earlier gives a listener a playlist that 404s, which reads as broken rather than as not-started-yet.
- Refusals are `409` with a code in `data.code` (`404` shape for `NOT_FOUND` is *not* used here — Spaces return 409 for everything, including `NOT_FOUND`).

Standard Response Format

```
{
  "success": true,
  "httpStatus": "OK",
  "message": "Space created",
  "action_time": "2026-08-20T10:30:45",
  "data": { }
}
```

Refusals — note the shape

`409 CONFLICT`. **Spaces put the prose in `message` and the code in `data`** — the opposite of streams and calls, which put the code in `message`:

```
{
  "success": true,
  "httpStatus": "CONFLICT",
  "message": "The stage is full (10)",
  "action_time": "2026-08-20T10:30:45",
  "data": { "code": "STAGE_FULL" }
}
```

Always read `data.code`. That is the only field consistent across all three features. `success` stays `true` on a refusal — the envelope reports transport, not business outcome.

`SpaceDeniedException` is its own type rather than borrowing VP Live's: a Space is no longer a stream, and an exception crossing that line would quietly re-create the coupling the entity split exists to remove.

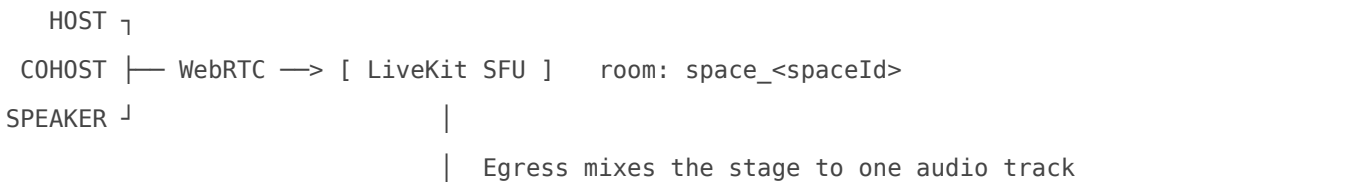
Why a Space is not a stream

A Space started life as VP Live's third `StreamMode`, which made it share a state machine, ingest keys and scheduling with Live and Radio. It now has its own table and its own endpoints, because:

- it is a **two-way conversation**, so it belongs with the other two-way features in `chat_system`;
- sharing the row meant `chat_system` importing `live_mng` — the boundary crossing the split removes.

What it **keeps** from that heritage is the *shape*, and deliberately so: the stage is WebRTC and the audience is HLS, so a Space still needs an ingest key (Egress publishes to SRS with it) and a playback URL (what listeners actually play). Those are not chat concepts; they are **the price of an audience that does not cost per head**.

The cost model — the one diagram to present



```
                        ▼
                RTMP  rtmp://srs:1935/space/<ingestKey>
                        |
                  [  SRS  ]
                        |  HLS
                        ▼
                [   CDN   ]
                        |
LISTENER ×10 000 ———————┘        (no token, no SFU, no per-head cost)
```

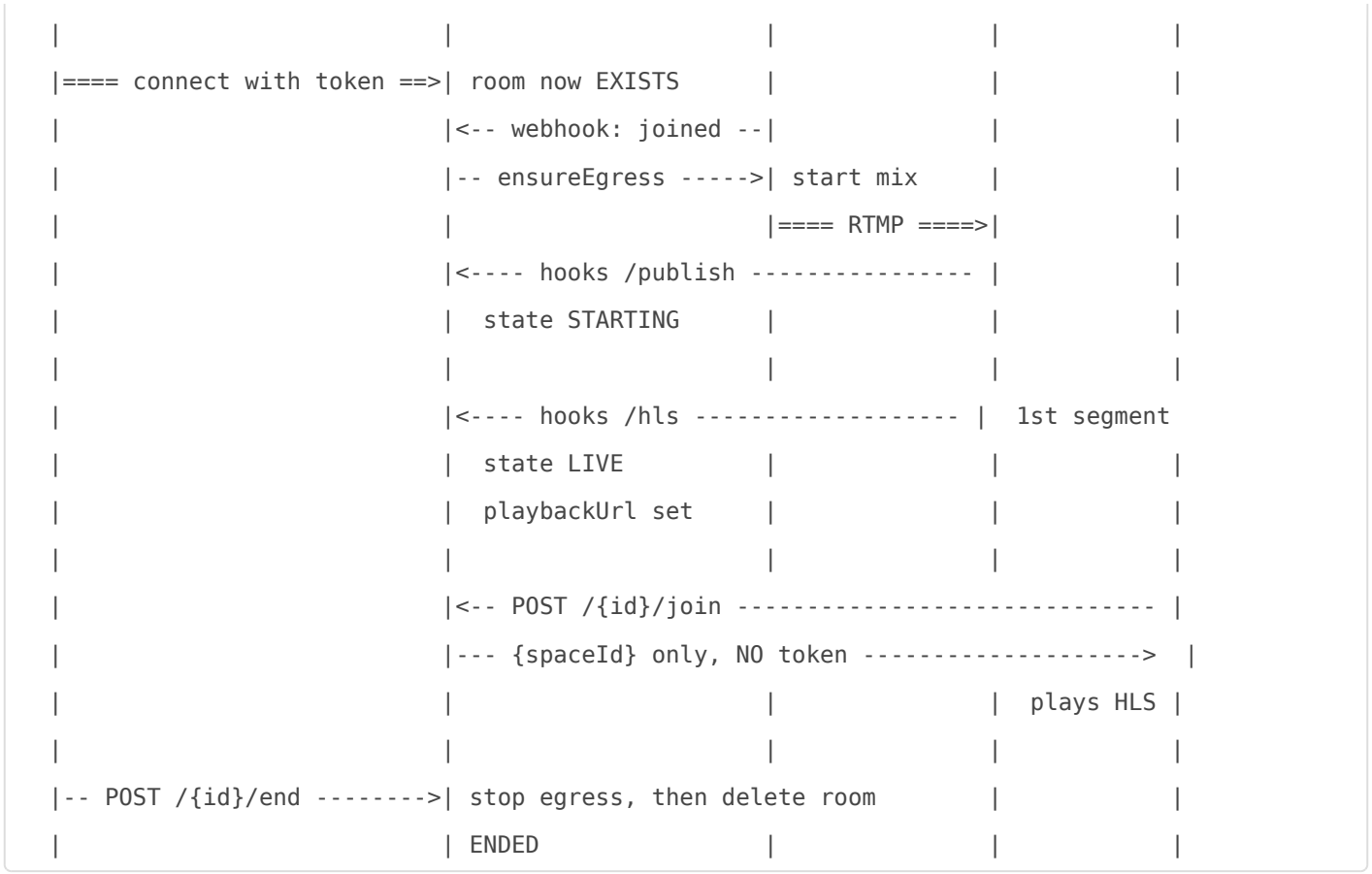
An SFU costs **linearly per subscriber**. An audience of ten thousand on LiveKit would need ten thousand SFU downstreams; on HLS it is one origin read fanned out by the CDN. So listeners must never become LiveKit participants — and the enforcement is simply that **they are never given a token**.

`SpaceRole`	On the SFU?	May change the stage?
`HOST`	yes	yes
`COHOST`	yes	yes
`SPEAKER`	yes	no
`LISTENER`	**no** — HLS via CDN	no

Stage cap: **10** (`app.live.space-stage-limit`). A product limit, not an infrastructure one — Egress was measured carrying 10+ concurrent Spaces. A conversation with twenty people talking is not a conversation.

Lifecycle — and the trap in the middle of it

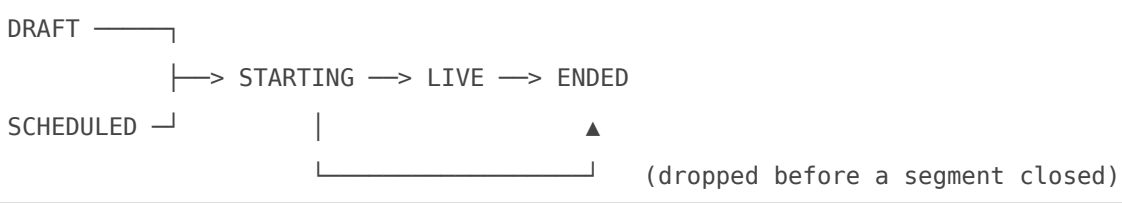
```
HOST                    BACKEND                LIVEKIT          SRS        LISTENER
 |                         |                      |              |            |
 |-- POST /live/spaces ----->| DRAFT                |              |            |
 |                         | mints roomName +     |              |            |
 |                         | ingestKey (12h) NOW  |              |            |
 |<-- {id, state:DRAFT} ----|                      |              |            |
 |                         |                      |              |            |
 |-- POST /{id}/stage ------>| upsert HOST          |              |            |
 |<-- {token, roomName} -----|  *** no egress yet ***               |            |
```



The trap, and it cost a debugging session: the mix is **not** started at `openStage`. Egress cannot mix a room that does not exist, and a LiveKit room does not come into being until the first participant connects — which has not happened yet at `openStage`, because we are still minting the token they will use to connect. Starting egress there returns **404 every time**. So `ensureEgress` runs off the LiveKit *participant-joined webhook* instead, and is idempotent, so every later joiner is a no-op.

If egress refuses, the log says it plainly: "*speakers will hear each other and nobody else will.*" That is the exact failure signature — a stage that works and an audience that gets nothing.

State machine



State	Meaning	Set by
`DRAFT`	Being set up. The room does not exist yet and nothing costs anything	`create`
`SCHEDULED`	Booked for later	`create` with `scheduledFor`

State	Meaning	Set by
`STARTING`	The mix is reaching SRS, but no segment has closed — a listener has nothing to play yet. **Deliberately not `LIVE`**	SRS `/publish`
`LIVE`	A segment exists. The first moment a listener can actually hear it	SRS `/hls`
`ENDED`	Over	host `end`, or SRS `/unpublish`

`onUnpublish` counts `STARTING` as well as `LIVE`: a stage that connected and dropped before any segment closed still ended.

> `SpaceState` also declares `PROCESSING`, `REPLAY_READY` and `FAILED`. **No code path sets them today** — see *Known gaps*.

Endpoints

`api/v1/live/spaces`

Method	Path	Token?	Who
`POST`	`/live/spaces`	no	anyone
`GET`	`/live/spaces/{spaceId}`	no	anyone
`POST`	`/live/spaces/{spaceId}/stage`	**yes**	host
`POST`	`/live/spaces/{spaceId}/join`	**no, on purpose**	anyone
`PUT`	`/live/spaces/{spaceId}/hand`	no	participant
`PUT`	`/live/spaces/{spaceId}/speakers/{userId}`	**yes**	host / co-host
`DELETE`	`/live/spaces/{spaceId}/speakers/{userId}`	no	host / co-host
`DELETE`	`/live/spaces/{spaceId}/me`	no	participant
`GET`	`/live/spaces/{spaceId}/participants`	no	anyone
`POST`	`/live/spaces/{spaceId}/end`	no	host

`POST` /api/v1/live/spaces — Create

Request:

Field	Type	Required	Description
`title`	string	**yes**	
`description`	string	no	
`scheduledFor`	ISO-8601	no	Present → `SCHEDULED`; absent → `DRAFT`

```
curl -s -X POST $BASE/api/v1/live/spaces \
  -H "Authorization: Bearer $TOKEN" -H 'Content-Type: application/json' \
  -d '{"title":"Friday pricing talk"}'
```

The host is **always a person** — a shop cannot hold a conversation.

`roomName` and `ingestKey` are minted **up front**, at creation: Egress needs the key the moment the stage opens, and unlike a phone there is no device-check screen to ask on. The key lives **12 hours** and is single-purpose, so a leaked key is not a standing right to publish into this Space.

`GET` /api/v1/live/spaces/{spaceId} — Read

Response (`data`):

Field	Type	Notes
`id`	UUID	
`state`	enum	
`hostUserId`	UUID	
`title`, `description`	string	
`scheduledFor`	ISO-8601	
`playbackUrl`	string	**null unless** `LIVE`
`startedAt`, `endedAt`	ISO-8601	

Note what is absent: `ingestKey`, `roomName` and `egressId`. They are broadcast internals and never appear on an object listeners read.

`POST` /api/v1/live/spaces/{spaceId}/stage — Open the stage

Host only. Registers the host as `HOST` and returns a LiveKit token.

Response:

```
{ "spaceId": "…", "roomName": "space_<spaceId>", "token": "…" }
```

This does not make the Space `LIVE`. SRS does that when the mixed audio reaches it — a room with a host and no audio is not a broadcast, and marking it live hands listeners a playlist that 404s.

`POST` /api/v1/live/spaces/{spaceId}/join — Join the audience

Returns no connection details, deliberately:

```
{ "spaceId": "…" }
```

The listener plays the `playbackUrl` from `GET /live/spaces/{spaceId}` like any other HLS viewer. A token here is precisely what would put the audience on the SFU.

`PUT` /api/v1/live/spaces/{spaceId}/hand — Raise / lower a hand

Param	In	Required
`raised`	query	**yes** (`true` / `false`)

Refuses with `ALREADY_SPEAKING` if you are already on stage, and `NOT_IN_SPACE` if you never joined.

`PUT` /api/v1/live/spaces/{spaceId}/speakers/{userId} — Promote

Host or co-host. Sets the target to `SPEAKER`, clears their raised hand, and returns **their** token:

```
{ "userId": "…", "token": "…" }
```

The riskiest interaction in the feature: the token is a **permission, not a connection**. Crossing from HLS playback to a LiveKit publish without a gap in audio is the client's problem.

`DELETE` /api/v1/live/spaces/{spaceId}/speakers/{userId} — Demote

Back to `LISTENER`. Refuses `CANNOT_DEMOTE_HOST`.

The token is not revoked, because a LiveKit token cannot be revoked — the same limit calls hit with the camera rule. What ends their turn is the client disconnecting; recording the demotion is what makes the *next* promotion valid rather than a duplicate.

`DELETE` /api/v1/live/spaces/{spaceId}/me — Leave

Sets `leftAt` and lowers the hand. **The row is kept, not deleted** — a Space is a conversation, and who spoke in it is what moderation needs to answer later; deleting the row deletes the answer.

Rejoining after a dropped connection **must not demote a speaker back into the audience**, so the upsert only ever raises a `LISTENER`, never lowers a speaker.

`GET` /api/v1/live/spaces/{spaceId}/participants — Who is here

Everyone who has not left. The envelope `message` is `"<n> in this Space"`.

```
[ { "userId": "…", "role": "SPEAKER", "handRaised": false, "joinedAt": "…" } ]
```

`POST` /api/v1/live/spaces/{spaceId}/end — End it

Host only. Idempotent — ending an `ENDED` Space returns it unchanged.

Order matters: **stop the mix first, then delete the room.** Deleting the room drops the stage, and an egress still running against a room that no longer exists is a job nobody will ever stop.

The ingest key is deliberately **not cleared** on end: a recording callback arrives after the Space ends and is named after the key SRS received. Nothing is weakened, because a usable key requires `acceptsPublish()`, which an `ENDED` Space fails.

Refusal codes

Code	When
`NOT_FOUND`	no such Space
`NOT_HOST`	host-only verb (`stage`, `end`)
`NOT_OPEN`	the Space has ended; it will not accept a publisher
`NOT_IN_SPACE`	you (or the target) never joined
`NOT_STAGE_CONTROL`	only the host or a co-host can change the stage
`STAGE_FULL`	at the stage limit (10)
`ALREADY_SPEAKING`	raising a hand while already on stage
`CANNOT_DEMOTE_HOST`	the host holds the Space

Server-to-server (internal)

Neither of these is for clients. **They are what actually drive Space state.**

LiveKit webhooks ? `SpaceRoomHandler`

Routed by room-name prefix: **calls own `call_*`, Spaces own `space_*`.** Without that split every Space event would reach `CallWebhookService`, which resolves each one against a `CallEntity` and would find nothing — quietly, for every participant of every Space.

Event	Effect
participant joined	**`ensureEgress`** — the earliest moment Egress can be asked to mix. Idempotent
participant left	`leave(user, space)`. Only stage members are LiveKit participants, so **this never fires for a listener**
room finished	Logged only. The Space's state still belongs to SRS
track published	A non-`AUDIO` track is logged as a **client bug** — Spaces are audio only

SRS hooks ? `SpaceSrsHandler` (app `space`)

The publisher here is **never a person**: it is LiveKit Egress pushing the mixed stage in as RTMP. SRS cannot tell the difference and does not need to — the ingest key is the credential either way, which is what lets a Space go `LIVE` by the same rule as every broadcast: **because media arrived, not because an app said so.**

Hook	Effect
`/publish`	Validates the key; `STARTING`. Returns a refusal if the key authorises nothing
`/hls`	First segment closed → **`LIVE`**, `playbackUrl` set
`/unpublish`	The mix stopped → `ENDED`
`/dvr`	**Logged, not published.** Recording a Space is a consent question a broadcast does not raise, and is not built. The file exists on disk for moderation only

Configuration

Property	Default	Meaning
`app.live.space-stage-limit`	`10`	Concurrent speakers
`app.live.rtmp-ingest-url`	`rtmp://srs:1935`	Where Egress publishes the mix
`app.live.playback-base-url`	`http://127.0.0.1:8080`	HLS base; playlist is `/space/<ingestKey>.m3u8`
`app.livekit.url`	`ws://127.0.0.1:7880`	SFU websocket

Ingest key: 24 random bytes, URL-safe Base64, unpadded. Valid 12 hours.

Schema note

Enum columns are **VARCHAR, not Postgres enums**. Hibernate freezes a `CHECK` constraint around an enum column when the table is created and never revisits it, so a value added later is refused at `INSERT` — **a trap that has already cost this project twice.**

Known gaps

Honest list, from reading the package — worth knowing before the demo so nobody asks about them from the floor.

Gap	Detail
`COHOST` cannot be assigned	`requireStageControl` honours it, but no endpoint or code path ever *sets* the role. `promote` grants `SPEAKER`. Today, only the host can control the stage
`SCHEDULED` never opens itself	Nothing schedules or transitions it — no job, no announcer. The host simply opens the stage whenever they like. Unlike streams, which have `StreamScheduleJob`
No announcements or notifications	Nothing outside `space_mng` references Spaces at all. No follower announce, no reminders, no push
`PROCESSING` / `REPLAY_READY` / `FAILED` are unreachable	Declared on `SpaceState`, never set. Replay is not built — `/dvr` deliberately does not publish
No comments on a Space	Live comments are a stream feature (`/live/streams/{id}/comments`); Spaces have no equivalent overlay
Demotion cannot force a disconnect	LiveKit tokens are not revocable; a demoted speaker stops publishing only when their client cooperates

---

Revision #1
Created 20 August 2026 17:02:30 by Admin Qbit
Updated 20 August 2026 17:02:46 by Admin Qbit