API REFERENCE — DOWNLOADABLE EDITION

Every Call Handled — Platform API

The complete API reference · 18 chapters · 122 endpoints
Generated 2026-08-20 from the same source as the interactive reference.
For live, personalised samples and the request console, use the online version.

Contents

01 Getting Started

02 Quickstart — Create, Configure, Go Live

03 Teach Your Assistant — Knowledge

04 Make It Yours — Settings, Voice & Web Chat

05 Go Live — Phone Numbers & Web Chat

06 Bookings

07 Intake Forms

08 See What's Happening

09 Billing & Credits

10 Your Team — Roles & Permissions

11 API Keys

12 Credentials — Connect Your Own Accounts

13 Automations

14 For Partners

15 Chat API & Webhooks

16 Error Handling

17 Best Practices

18 Support

01

Getting Started

Overview

Everything starts with two things: the base URL and an API token. This chapter gets you from zero to your first successful request.

Base URL

What	Host
The API	`https://api.everycallhandled.com`
Your live assistants	`https://{subdomain}.everycallhandled.com` (or `https://{subdomain}.app.everycallhandled.com`)

All API endpoints are prefixed with `/api/v1`.

Requirements

- An API authentication token (request from `support@everycallhandled.com`, or mint a scoped key yourself — see API Keys)
- An HTTPS client capable of REST API calls
- JSON request/response handling

Conventions

- All timestamps are ISO 8601 (e.g. `"2026-05-10T12:00:00Z"`).
- Boolean attributes on apps are stored as strings (`"true"` / `"false"`). Where this affects your client, the docs call it out explicitly.
- Standard HTTP status codes (200/400/401/403/404/409/429/500); error bodies are nested under `err`: `{ "err": { "message": "<message>", "code": "<symbolic_code>" } }` (`code` defaults to `INTERNAL_ERROR`).
- API tokens are long-lived credentials — store securely, rotate periodically.
- Asynchronous operations (knowledge ingestion, taking an assistant live) return immediately; you poll for completion.

Authentication

Email `support@everycallhandled.com` for a token, or mint a capability-scoped key yourself once you have dashboard access (see API Keys). Keep tokens out of client-side code and version control; store them in environment variables or a secrets manager. Contact support immediately if you suspect a token has been compromised.

Include your token in the `Authorization` header on every request:

```
Authorization: Bearer YOUR_API_TOKEN
```

Your first request

List the apps in your account:

```
curl -X GET https://api.everycallhandled.com/api/v1/apps \
  -H 'Authorization: Bearer YOUR_API_TOKEN'
```

A `200` with a JSON body means you're authenticated and ready to build. A `401` means the token is missing or invalid.

What your token can do

Every token acts as a member of your account, with a **role** and a set of **permissions** (features). The role sets a baseline; per-user overrides fine-tune it. When an endpoint needs a specific permission, its documentation says so — for example `Requires `edit_settings` .` Endpoints with no permission noted are available to any signed-in member of the account that owns the app.

Tokens issued by support come pre-configured with the permissions your integration needs. The full permission catalogue and role baselines are in Your Team — Roles & Permissions.

02

Quickstart — Create, Configure, Go Live

Overview

The fastest way to understand the platform is to build an assistant end-to-end. The script below creates an assistant for a fictional heating company, teaches it from the company website, and takes it live — the same five-step journey from the welcome page, in about a minute of API calls.

```
TOKEN="your_api_token_here"

# 1. Create the app
APP=$(curl -s -X POST https://api.everycallhandled.com/api/v1/apps \
  -H "Authorization: Bearer $TOKEN" \
  -H "Content-Type: application/json" \
  -d '{
    "name": "HVAC Receptionist",
    "attributes": {
      "voiceChatPrompt": "You are the AI receptionist for ABC HVAC. Office hours: Mon-Fri 8am-
5pm. Always confirm callback number.",
      "voice": "shimmer",
      "voiceChatEnabled": "true",
      "primaryColor": "#1E3A5F"
    }
  }')
APP_ID=$(echo $APP | jq -r .id)
echo "Created app: $APP_ID"

# 2. Teach it from the company website
JOB=$(curl -s -X POST https://api.everycallhandled.com/api/v1/apps/$APP_ID/knowledge \
  -H "Authorization: Bearer $TOKEN" \
  -H "Content-Type: application/json" \
  -d '{ "type": "url", "content": "https://abchvac.com/services" }')
JOB_ID=$(echo $JOB | jq -r .jobId)

# Poll until ingestion completes
while true; do
  STATUS=$(curl -s https://api.everycallhandled.com/api/v1/apps/$APP_ID/knowledge/jobs/$JOB_ID
\
    -H "Authorization: Bearer $TOKEN" | jq -r .status)
  echo "Knowledge ingest: $STATUS"
  [ "$STATUS" = "completed" ] && break
  [ "$STATUS" = "failed" ] && exit 1
  sleep 3
done

# 3. Take it live
curl -X POST https://api.everycallhandled.com/api/v1/apps/$APP_ID/publish \
  -H "Authorization: Bearer $TOKEN" \
  -H "Content-Type: application/json" \
  -d '{ "subdomain": "abc-hvac", "userEmail": "owner@abchvac.com" }'

# 4. Verify the live app
```

```
curl https://api.everycallhandled.com/api/v1/apps/$APP_ID \
  -H "Authorization: Bearer $TOKEN" | jq '.publishedAppUrl, .attributes.creditBalance'
```

That's a live assistant with a web address. From here you'd give it a phone number (Go Live), connect a calendar (Bookings), and watch the conversations arrive (See What's Happening).

The one concept to internalise before you write real integration code: **editing an app changes a draft; taking it live makes those changes real to your callers.** Change the prompt, the voice, the branding — nothing your callers experience changes until you take the assistant live again. The rest of this chapter covers the app endpoints in detail.

Create an App

An app is one assistant: its instructions, voice, knowledge, branding and (once live) its phone number and web address. Most businesses start with one; you can run many.

`POST /api/v1/apps`

Requires `create_app`.

Request:

```
curl -X POST https://api.everycallhandled.com/api/v1/apps \
  -H 'Authorization: Bearer YOUR_TOKEN' \
  -H 'Content-Type: application/json' \
  -d '{
    "name": "Customer Support Bot",
    "description": "AI assistant for customer support",
    "attributes": {
      "assistantName": "Support Assistant",
      "welcomeMessage": "Hello! How can I help you today?",
      "primaryColor": "#4F46E5",
      "textColor": "#FFFFFF",
      "voice": "nova",
      "voiceChatPrompt": "You are a helpful customer support assistant."
    }
  }'
```

All attributes are optional at creation — you can set everything later with an update. The full attribute reference is in Make It Yours.

Response (200):

```
{
  "id": "550e8400-e29b-41d4-a716-446655440000",
  "organisationId": "org-abc123",
  "name": "Customer Support Bot",
  "description": "AI assistant for customer support",
  "attributes": { /* echoed back, normalised */ },
  "chatId": "...",
  "createdAt": "2026-05-10T10:30:00Z",
  "updatedAt": "2026-05-10T10:30:00Z"
}
```

Save the `id` — you'll need it for all subsequent operations. (Published state and `createdByUserId` are returned by `GET /apps/{id}`, not by create.)

Errors:

- `409 PHONE_NUMBER_IN_USE` if `attributes.twilioPhoneNumber` is already used by another of your apps.

List Apps

```
GET /api/v1/apps?published=all&limit=50&offset=0
```

Requires `view_settings`.

Query params:

- `published` — `true`, `false`, or `all`. **If omitted, only published apps are returned** (pass `all` to include drafts).
- `limit` (default 50, max 200), `offset` (pagination)

Response (200):

```
{
  "apps": [ /* array of app objects (same shape as GET /apps/{id}) */ ],
  "isAdmin": false,
  "pagination": { "total": 17, "limit": 50, "offset": 0, "hasMore": false, "nextOffset": null }
}
```

The response is filtered to your account: you see your own apps and those of any customer accounts beneath yours (see For Partners). Credit balances shown for live assistants are always current.

Get an App

```
GET /api/v1/apps/{appId}
```

```
curl 'https://api.everycallhandled.com/api/v1/apps/{appId}' \
  -H 'Authorization: Bearer YOUR_TOKEN'
```

Available to any signed-in member of the account that owns the app.

Response (200):

```
{
  "id": "550e8400-...",
  "name": "Customer Support Bot",
  "description": "...",
  "attributes": { /* full attribute set; balances and usage totals are always current live
values */ },
  "createdByUserId": "...",
  "isPublished": true,
  "publishedAppUrl": "https://support.app.everycallhandled.com",
  "subdomain": "support",
  "hasPendingPublish": false,
  "pendingPublishAttrIds": [],
  "createdAt": "...",
  "updatedAt": "..."
}
```

`hasPendingPublish` is `true` when the app has draft changes that aren't live yet; `pendingPublishAttrIds` lists which attributes are waiting. Use these to drive an "Apply your changes" prompt in your own UI.

Errors:

- `404 APP_NOT_FOUND` — the app doesn't exist OR your token can't access it (deliberately the same error to avoid existence leaks).

Update an App

`PATCH /api/v1/apps/{appId}`

Requires `edit_settings`.

```
curl -X PATCH https://api.everycallhandled.com/api/v1/apps/$APP_ID \
  -H 'Authorization: Bearer YOUR_TOKEN' \
  -H 'Content-Type: application/json' \
  -d '{
    "name": "Renamed app",
    "attributes": {
      "voiceChatPrompt": "Updated prompt...",
      "voice": "ballad"
    }
  }'
```

PATCH semantics — only fields you supply are changed. Updates are **draft changes**: take the assistant live again to apply them (see Go Live).

Delete an App

`DELETE /api/v1/apps/{appId}`

```
curl -X DELETE 'https://api.everycallhandled.com/api/v1/apps/{appId}' \
  -H 'Authorization: Bearer YOUR_TOKEN'
```

Requires `edit_settings`. Permanently removes the app together with its knowledge, conversations and billing history. If the app was live, its site is taken down too. This cannot be undone.

Duplicate an App

`POST /api/v1/apps/{appId}/clone`

Requires `clone_app`.

```
curl -X POST https://api.everycallhandled.com/api/v1/apps/$APP_ID/clone \
  -H 'Authorization: Bearer YOUR_TOKEN' \
  -d '{ "name": "Copy of My App" }'
```

The duplicate inherits the prompt, tools, voice configuration and billing mode. It does **not** inherit the phone number, live state or credit balance. Duplicating an app is the safe way to experiment with big changes without touching the assistant your callers are using.

03

Teach Your Assistant — Knowledge

Overview

An assistant is only as good as what it knows. The knowledge base is where you give it your facts — your services, prices, opening hours, policies — so it answers with your information instead of improvising. Upload web pages, plain text or files; the platform ingests them in the background and the assistant draws on them in every conversation.

Each app has its own knowledge base. Ingestion is asynchronous: uploads return a `jobId` you can poll.

Upload Content

`POST /api/v1/apps/{appId}/knowledge`

Requires `edit_knowledge`.

JSON for URLs / text:

```
curl -X POST https://api.everycallhandled.com/api/v1/apps/$APP_ID/knowledge \
  -H 'Authorization: Bearer YOUR_TOKEN' \
  -H 'Content-Type: application/json' \
  -d '{ "type": "url", "content": "https://example.com/help" }'
```

Files (PDF, DOCX, etc.) use a two-step JSON flow — no multipart. Ask for a presigned URL, then PUT the file bytes to it:

```
# 1. Request an upload URL (JSON)
curl -X POST https://api.everycallhandled.com/api/v1/apps/$APP_ID/knowledge \
  -H 'Authorization: Bearer YOUR_TOKEN' \
  -H 'Content-Type: application/json' \
  -d '{ "type": "file", "filename": "manual.pdf", "contentType": "application/pdf" }'
# → { "uploadUrl": "https://...", "finalUrl": "https://...", "jobId": "...", "contentId": "..."
}

# 2. PUT the file bytes to the returned uploadUrl
curl -X PUT "$UPLOAD_URL" \
  -H 'Content-Type: application/pdf' \
  --data-binary @./manual.pdf
```

Response (200):

- URL / text: `{ "jobId": "...", "status": "processing", "estimatedCompletionTime": "...", "contentId": "..." }`
- File: `{ "uploadUrl": "...", "finalUrl": "...", "jobId": "...", "contentId": "..." }`

List Content

```
GET /api/v1/apps/{appId}/knowledge
```

```
curl 'https://api.everycallhandled.com/api/v1/apps/{appId}/knowledge' \
  -H 'Authorization: Bearer YOUR_TOKEN'
```

Requires `view_knowledge`.

Response (200): `{ "items": [ { "contentId": "...", "filename": "...", "fileType": "...", "createdAt": "...", "createdBy": "..." } ] }`

Delete Content

```
DELETE /api/v1/apps/{appId}/knowledge/{contentId}
```

```
curl -X DELETE 'https://api.everycallhandled.com/api/v1/apps/{appId}/knowledge/{contentId}' \
  -H 'Authorization: Bearer YOUR_TOKEN'
```

Requires `edit_knowledge`. Removes the item; the assistant stops using it shortly afterwards (re-indexing runs in the background).

Poll Job Status

```
GET /api/v1/apps/{appId}/knowledge/jobs/{jobId}
```

```
curl 'https://api.everycallhandled.com/api/v1/apps/{appId}/knowledge/jobs/{jobId}' \
  -H 'Authorization: Bearer YOUR_TOKEN'
```

Requires `view_knowledge`.

Response (200):

```
{
  "jobId": "...",
  "status": "queued" | "processing" | "completed" | "failed",
  "progress": 67,
  "documentsProcessed": 12,
  "error": null
}
```

Poll every ~3 seconds while `processing`. Most ingestion jobs complete within 30 seconds; large PDFs may take a few minutes.

Knowledge takes effect as soon as ingestion completes — it applies to your live assistant immediately, so you can refresh its facts at any time.

04

Make It Yours — Settings, Voice & Web Chat

Overview

Out of the box your assistant is capable but generic. This chapter is where it becomes *yours*: its name, its greeting, how it sounds on the phone, how the chat widget looks on your website, which languages it opens in, and what it shows visitors. Everything here is configuration you set once and refine as you learn what your callers need.

All of these are app attributes set via `PATCH /api/v1/apps/{appId}` unless a different endpoint is shown. Remember: attribute changes are drafts until you take the assistant live (web-chat settings written via the dedicated endpoint below are the exception — they take effect immediately).

App attribute reference

Identity & behaviour:

- `assistantName` — display name in the chat UI
- `welcomeMessage` — initial greeting
- `voiceChatPrompt` — system instructions for conversations. Either a string (single prompt) or an object `{"default": "...", "demo": "..."}` for multi-mode prompts (selected via `?v=mode` on the voice link)
- `voiceChatTools` — JSON array of tool definitions the assistant can call
- `voice` — voice option (case-insensitive). Valid: `alloy` (default), `ash`, `ballad`, `coral`, `echo`, `fable`, `nova`, `onyx`, `sage`, `shimmer`, `verse`

Appearance:

- `primaryColor`, `secondaryColor`, `tertiaryColor`, `textColor` — hex codes. **Legacy**: for the web-chat widget these are fallbacks only — style the widget via `webchat.branding` (below), where one brand colour derives the whole palette.
- `font` — font family. **Legacy fallback** for the widget; prefer `webchat.branding.fontFamily`. Supported webchat fonts: `Manrope`, `Inter`, `Roboto`, `Open Sans`, `Lato`, `Montserrat`, `Poppins`, `Nunito`, `Source Sans 3`, `Work Sans`, `DM Sans`, `Karla`, `Merriweather`, `Playfair Display`. Set `webchat.branding.fontUrl` to `https://<widget-host>/widget/fonts/<slug>.woff2` (slug = the name lower-cased, spaces → hyphens) so the widget loads the face on any host page.
- `assistantAvatar`, `thumbnailImage` — image objects `{s3Key, url}`
- `embedAppTitle` — title in iframe-embed mode

Billing:

- `accountId` — if set, the app draws from an account credit pool. (Also returned/accepted as the legacy alias `merchantId` — see the naming-transition note under For Partners.)
- `billingMode` — `"app"` (default) or `"merchant"` (pool-funded; see Billing & Credits)
- `pricePerMinute`, `allowanceMinutes`, `overageRatePence` — per-app billing overrides
- `lowBalanceThreshold` — alert threshold in seconds (default 600)

Voice tuning (see the voice attribute table below):

- `voiceEngine`, `turnDetectionType`, `vadEagernessPhase1`, `vadEagernessPhase2`, `vadThreshold`, `vadPrefixPaddingMs`, `vadSilenceDurationMs`, `noiseReduction`, `transcriptionModel`, `transcriptionLanguage`, `maxOutputTokens`, `pipelineConfig`

Voice attribute reference

These control how the assistant listens, transcribes, thinks and speaks on voice calls. The defaults suit most businesses; tune them when you have a specific problem (an assistant that interrupts, a noisy call environment, a non-English caller base).

Attribute	Type	Notes
`voiceEngine`	string	`"openai"` (default), `"livekit"`
`turnDetectionType`	string	`"semantic_vad"` (default), `"server_vad"`, `"disabled"`
`vadEagernessPhase1`	string	`low` / `medium` (default) / `high` / `auto` — speech-end detection during greeting
`vadEagernessPhase2`	string	`low` / `medium` / `high` (default) / `auto` — during conversation
`vadThreshold`	number	0.0-1.0, server_vad only (default 0.5)
`vadPrefixPaddingMs`	number	0-2000, server_vad only (default 300)
`vadSilenceDurationMs`	number	200-2000, server_vad only (default 500)
`noiseReduction`	string	`near_field` / `far_field` (default) / `disabled`
`transcriptionModel`	string	`gpt-4o-transcribe` / `gpt-4o-mini-transcribe` (default) / `whisper-1`
`transcriptionLanguage`	string	ISO 639-1 (e.g. `en`, `es`, `fr`). Default `en`
`maxOutputTokens`	number	256-4096 or -1 for unlimited (default 4096)
`voiceDisplayName`	string	Spoken voice label, also used in voice greetings
`isPreventKBAccessForVoiceChat`	string ("true"/"false")	Default false

Set Voice Pipeline Config (atomic)

Two ways to set voice configuration: field-by-field via `PATCH /apps/{id}` (useful for small tweaks), or atomically as a single `pipelineConfig` blob — one write, internally consistent. Atomic is preferred when you're setting several values that belong together.

POST `/api/v1/apps/{appId}/voice/pipeline-config`

Requires `edit_settings`.

```
curl -X POST https://api.everycallhandled.com/api/v1/apps/$APP_ID/voice/pipeline-config \
  -H 'Authorization: Bearer YOUR_TOKEN' \
  -H 'Content-Type: application/json' \
  -d '{
    "pipelineConfig": {
      "mode": "modular",
      "stt": { "provider": "openai" },
      "llm": { "provider": "openai", "model": "gpt-4o" },
      "tts": {
        "provider": "cartesia",
        "voice": "794f9389-aac1-45b6-b726-9d9369183238",
        "options": { "locale": "en-GB", "gender": "female", "speed": 1.0 }
      },
      "vad": { "type": "server_vad", "threshold": 0.5, "prefix_padding_ms": 200 },
      "multilingual": true,
      "noise_filter": "noizeless"
    }
  }'
```

This is a draft change — take the assistant live to apply it.

Get Resolved Voice Config

GET `/api/v1/apps/{appId}/voice/config`

```
curl 'https://api.everycallhandled.com/api/v1/apps/{appId}/voice/config' \
  -H 'Authorization: Bearer YOUR_TOKEN'
```

Requires `edit_settings`. Returns the merged voice configuration (the `pipelineConfig` blob + individual fields + platform defaults) — what the assistant will actually use, after all layers resolve.

Web-chat branding

The web-chat widget is styled from a single `webchat.branding` object, written via the web-chat endpoint below. Set `primaryColor` and the platform derives the full widget palette (header, bubbles, chat button, contrast text). Optional hex overrides, each replacing its derived value when set and reverting to derived when empty: `headerBgColor`, `headerTextColor`, `assistantBubbleColor`, `assistantTextColor`, `userBubbleColor`, `userTextColor`, `launcherColor`, `launcherIconColor`. Plus `surface` (`light` | `dark` | `auto`), `radius` (0–24), `position` (`bottom-right` | `bottom-left`), `logoUrl` (https), `launcherIcon` (`chat` | `logo` |https URL), `fontFamily` / `fontUrl` (see the font list above).

Web-chat conversation config

Beyond branding, the widget reads a small set of conversational keys from the same `webchat` config object. All are written with:

```
PUT /api/v1/apps/{appId}/webchat
```

```
curl -X PUT 'https://api.everycallhandled.com/api/v1/apps/{appId}/webchat' \
  -H 'Authorization: Bearer YOUR_TOKEN' \
  -H 'Content-Type: application/json' \
  -d '{"webchat":{"quickActions":[{"label":"Book an appointment","message":"I would like to
book an appointment"}]}}'
```

Requires `edit_settings`. The body is `{ "webchat": { ... } }` (a bare `{ ... }` object is also accepted). The write **validates then merges**: keys you omit are left untouched, and sibling config (branding, workflows, the tools array) is preserved — toggling one key never wipes the others. `GET /api/v1/apps/{appId}/webchat` returns the normalised, stored shape. Changes take effect on the live widget immediately.

`quickActions` — up to **4** opening buttons shown on a fresh chat. Array of `{ "label", "message" }`; tapping one sends `message` as the visitor's first turn. `label` is capped at 32 chars, `message` at 200; entries missing either field are dropped, and the list is capped at 4.

```
{
  "webchat": {
    "quickActions": [
      { "label": "Book an appointment", "message": "I'd like to book an appointment" },
      { "label": "Opening hours", "message": "What are your opening hours?" }
    ]
  }
}
```

`startLanguage` — the language the assistant opens in. Values:

- `""` (empty) — English (the default).
- `"auto"` — follow the visitor's browser / host-page language.
- a language **name** (e.g. `"Spanish"`) or ISO code (e.g. `"es"`) — open in that language.

The 12 supported non-English languages are: **French, Spanish, German, Italian, Portuguese, Polish, Romanian, Arabic, Turkish, Urdu, Hindi, Mandarin Chinese**. An unrecognised value resolves to English. The embed snippet can override the app setting per-page with the **`data-ghost-chat-lang`** attribute (same `auto` / name / code semantics); the embed attribute wins over `webchat.startLanguage`.

Opening with an intent — the embed also accepts **`data-ghost-chat-intent`**: a message auto-sent as the visitor's first turn when the chat opens on a fresh thread (e.g. a "Book now" button whose embed carries `data-ghost-chat-intent="I'd like to book an appointment"` opens the

chat already in the booking flow). Resumed conversations and threads with any prior visitor turn ignore it.

```
<script src="https://<widget-host>/widget/embed-div.js"
        data-config-id="YOUR_APP_ID"
        data-ghost-chat-lang="auto"></script>
```

`welcomeMessages` — an object mapping language to welcome-bubble text (up to 12 languages; values capped at 500 chars). Resolution rule for the static opening bubble:

- **Default language** (English / empty start language) — the app's standard welcome message.
- **Non-default language with a stored translation here** — that translation.
- **Non-default language with no translation** — **no static bubble**; the assistant's first turn greets the visitor in-language instead.

```
{
  "webchat": {
    "startLanguage": "auto",
    "welcomeMessages": {
      "Spanish": "¡Hola! ¿En qué puedo ayudarle hoy?",
      "French": "Bonjour ! Comment puis-je vous aider ?"
    }
  }
}
```

`emergencySignpost` — plain text (up to 200 chars), e.g. `"call 999 or NHS 111"`. Used by urgent escalation: when the office is closed and the visitor raises a genuine emergency, the assistant advises this pointer. Omit it and no emergency advice is appended.

```
{ "webchat": { "emergencySignpost": "For a genuine emergency, call 999." } }
```

Chat lifecycle

- **Assistant-ended chats** — when a conversation has naturally concluded (the visitor said goodbye, confirmed they're done, or asked to end), the assistant ends the chat itself. This **sends the conversation summary to your team immediately** (summary email + PDF transcript to the app's recipients, the same wrap-up voice calls get) and puts the widget into its "chat ended" state, from which the visitor can start a new chat.
- **Visitor-ended chats** — the visitor can end at any time via the widget's **End chat** control (with a confirm step). This runs the same wrap-up immediately.
- **Idle chats** — chats left inactive are wrapped up automatically (a sweep runs every ~10 minutes and closes chats idle past ~15 minutes), so a visitor who simply closes the tab still reaches your team.
- **Localised chrome** — the widget's own fixed strings (End chat, "This chat has ended…", the confirm prompts, Start a new chat) render in the resolved **start language**, matching the assistant's conversational language.

Rich chat components — the render contract

Chat doesn't have to be all text. Two mechanisms turn data into native chat UI (cards, tables, timelines, location cards) — in both, **the platform owns layout and enforces limits**; payloads that don't validate degrade to plain text, never an error.

Declare a render kind on any saved action (tools editor → "Show the reply as", or `configuration.render` in the stored tool):

```
"configuration": { "uri": "https://api.you.com/plans", "method": "GET", "render": "cards" }
```

Kinds: `cards` | `comparison` | `timeline` | `table` | `metrics` | `location` | `payment` | `upload` | `multi_select` | `order_cart` | `status` | `signature` | `rating` | `date_range` | `staff` | `document` | `media` | `locations`. The tool's HTTP response body must **be** the component payload, or carry it under a `_render` key alongside your other data. Payload shapes match the curated-component shapes below; validation caps counts and lengths, images must be https, and the review-card machinery (editable fields / edit actions) is platform-only and stripped. The "Test this action" panel shows a render verdict (which component would appear, or exactly why it fell back to text). The model's copy of the raw response is capped at 16 KB.

`payment` and `upload` are display-only from a saved action. These two kinds refuse to take their destination from the tool payload: a payment payload can state the amount and line items (each line `label` + `amountMinor` + optional `quantity`; the platform recomputes the total and rejects a payload whose stated total disagrees), and an upload payload can ask for a named accept group — but the response body can never supply a checkout URL or an upload destination. A saved action therefore renders the *amount card* or the *request* itself read-only. The live halves — a real **Pay now** button into Stripe's hosted checkout on the account's connected Stripe account, and signed upload destinations — are minted server-side by the platform at render time, and only inside a playbook's payment / attachment step on an app that has opted in (below). Card details are entered on Stripe's payment page, never in the chat, and uploads land as signed single-use destinations (one per accepted file extension, content type and size enforced by storage itself).

The 2026-07-30 kinds, briefly. Each projects the tool's response through the same whitelist discipline (invalid payloads fall back to text, never an error):

- `multi_select` — `{ "prompt", "options": [{"label", "hint"?}], "minSelections"?, "maxSelections"? }` (2–12 options). The visitor ticks and sends one canonical message.
- `order_cart` — `{ "prompt"?, "currency"?, "items": [{"value", "label", "priceMinor", "hint"?}], "maxPerItem"? }`. Prices are **your** integers in minor units — floats and zeros refuse, and a payment step charges exactly the banked order (`{"fromOrder": "<slot>"}` in a playbook payment step).
- `status` — `{ "title"?, "reference"?, "stages": [{"label", "when"?, "note"?}], "currentIndex", "badge"?, "updatedAt"?, "refreshMessage"? }`. Stage states are derived from `currentIndex` alone, so the card can never show a contradictory position; `refreshMessage` gives the visitor a Check-again tap.
- `signature` — `{ "title"?, "statement", "note"? }`. The statement renders verbatim and in full into the record; the drawing pad only appears inside a playbook signature step on an

app with attachments enabled.

- `rating` — `{ "prompt"?, "scale": 5|10, "minLabel"?, "maxLabel"? }`.
- `date_range` — `{ "prompt"?, "maxSpanDays"? }` renders the two-date picker.
- `staff` — `{ "prompt"?, "pickable"?, "people": [{"value", "name", "role"?, "bio"?, "photoUrl"? (https), "badges"?}] }`.
- `document` — `{ "title", "description"?, "url" }`; https and `.pdf` only, shown inline with an always-present open-in-new-tab.
- `media` — `{ "title", "description"?, "url" }`; a YouTube/Vimeo page URL or a direct https `.mp4` / `.webm` / `.mp3` file. The platform derives the embed itself (YouTube via the privacy-enhanced domain) — a supplied embed URL is never trusted, and nothing autoplays.
- `locations` — `{ "prompt"?, "pickable"?, "locations": [{"value", "name", "addressLines", "hours"?, "phone"?, "lat"?, "lon"?}] }`. Directions links are platform-built from the address; a map appears only when you supply coordinates.

Opt-ins live on `voiceChatTools` (same PATCH mechanics as everything else here, or the dashboard's Web chat tab → "Chat capabilities" card):

```
"voiceChatTools": {
  "payments": { "enabled": true },
  "attachments": { "enabled": true },
  "feedback": { "enabled": true, "scale": 10, "prompt": "How likely are you to recommend us?",
"minLabel": "Not likely", "maxLabel": "Very likely" },
  "appointmentBooking": { "waitlistEnabled": true }
}
```

The runtime requires `enabled` to be exactly `true`; absence or any other value means off. `payments` additionally requires a connected Stripe account on the billing account — without one the amount still renders, with an honest "payment is not available" note. Turning a switch off in the dashboard removes the block entirely.

Two conversation-level opt-ins joined them on 30 July: **`feedback`** makes the assistant proactively offer a tappable rating card when a visitor wraps up (once per conversation, on **your** configured scale and question — the assistant's own wording never overrides them, and it is instructed to thank and never argue with the score); **`appointmentBooking.waitlistEnabled`** offers a request-a-callback card, with preferred-time chips, when a visitor asks for a day with no availability — alongside, never instead of, the offer to pick another day. Both switches are on the same "Chat capabilities" card.

Saved-action auth & confirmation

Two per-action fields turn saved actions into a full conversational controller for your own API — available to any app as pure configuration.

`configuration.auth: "visitor-session"` — run as the signed-in visitor. Instead of app credentials, the chat forwards the *caller's own* session token as the request's `Authorization` header, so your API sees the person and applies its own permissions per call. Enforced server-side at call time:

- only real signed-in users — anonymous visitors never even see these actions in chat;
- https URLs only;
- the URL's **origin must be in the app's trusted list** (`webchat.trustedSessionOrigins` , tools editor → "Trusted session origins", up to 8) — the visitor's token is never forwarded anywhere else, and an unlisted origin returns a refusal instead of making the call.

```
"configuration": { "uri": "https://api.you.com/v1/me", "method": "GET", "auth": "visitor-session" }
"webchat": { "trustedSessionOrigins": ["https://api.you.com"] }
```

`{args.x}` path parameters — one action per endpoint, not per record. Tokens in the authored URI path substitute from the action's own arguments at call time: `https://api.you.com/v1/orders/{args.orderId}` plus a `parametersSchema` declaring `orderId` lets the assistant fetch any order. Values are URI-encoded (an argument can never add path separators or change the host), tokens are honoured in the path only, an argument used in the path doesn't also join the GET query string or request body, and a missing argument leaves the token in place so the call 404s loudly rather than hitting the wrong resource. For `visitor-session` actions the origin check runs on the resolved URL.

Structured (object) parameters — nested request bodies. Some APIs want a nested body (e.g. `{"fields": {"Name": "…"}}`). Give the action a parameter with type `object` and describe its fields as a JSON Schema fragment — in the tools editor, pick "Structured (JSON fields)" and paste the fields as JSON, e.g. `{ "properties": { "Name": { "type": "string" } }, "required": ["Name"] }` . The assistant then supplies that parameter as one JSON object inside the request body.

`configuration.confirmRequired: true` — ask for an explicit yes. The action's generated schema gains a required `confirm` flag and the assistant is instructed to state exactly what will happen and wait for the user's explicit yes. The hard floor is server-side: without `confirm: true` the platform refuses and **never calls your endpoint**. Use it for anything consequential (payments, cancellations, deletions).

Both compose with everything above: credentials stay available for app-authenticated actions, `render:` shapes the reply, and the `webchat.tools` allowlist still controls which actions reach the chat at all.

Curated component library — pre-approved content, served verbatim

Per-app, human-reviewed component payloads stored in `voiceChatTools.curatedComponents` (up to 8). Each enabled entry becomes a `show_<id>` tool the assistant may *elect*, but the content shown is the stored payload **verbatim** — the model chooses the moment, never the words.

Entry shape (written via the normal app PATCH with `voiceChatToolsReplace: true`, or managed in the dashboard's Web chat tab):

```
{ "id": "our_services", "kind": "cards", "name": "Our services",
  "election": "when the visitor asks what services you offer",
  "enabled": true,
  "payload": { "intro": "…", "items": [{ "title": "…", "badge": "£95", "facts": [{"label": "…",
"value": "…"}] }] } }
```

Kinds and payloads: `cards` (≤10 items: title, subtitle, badge, facts, sections, actions), `table` (≤6 columns × 20 rows + footnote), `location` (name, addressLines, phone, hours, note — the get-directions URL is always server-built from the address), `timeline` (≤8 steps). These four are the kinds available to *curated* components; the full render contract for tool results supports eight (`comparison` and `metrics` are render-directive-only today, and `payment` / `upload` carry the display-only rule above). Every payload is validated on write AND re-validated at serve time.

Drafting and freshness — the dashboard's Web chat tab can draft a best-guess component set from the app's own knowledge base, and later re-read the knowledge to suggest per-entry updates with reasons. Applying a suggestion is always a human's call; nothing changes without you saving it.

Related webchat gates

- `webchat.tools` (array of saved-action ids) is the **allowlist** for operator tools in the anonymous chat — default empty; a tool not listed never reaches visitors.
- `webchat.workflows` allowlists chat-callable automations the same way.
- Curated playbook toggles: `voiceChatTools.playbooks` may contain reference stubs `{ "id": "curated-booking", "curatedKey": "booking" }` (also `enquiry`, `my-account`) that expand at runtime from the platform's proven flow library.
- `voiceChatTools.payments.enabled` / `voiceChatTools.attachments.enabled` gate whether a playbook's payment or attachment step can go live (mint a checkout session / sign upload destinations) — see the render contract above.
- Playbook slots now cover the full component library: alongside `text` / `date` / `time` / `email` / `phone` / `enum`, a slot may be `multi_enum` (tick-all-that-apply, `"collect": "multi_select"`), `order` (a priced catalogue on the slot, `"collect": "order_cart"` — the payment step's `{"fromOrder": "<slot>"}` line charges exactly what was ordered at your authored prices), `consent` (your exact `statement`, `"collect": "signature"`), `rating` (`"scale": 5|10`), `address` (a validated UK address card), or `date_range` (`"maxSpanDays"` bounded). Enum slots can also collect via `staff_picker` (a `people` array on the slot) or `location_picker` (a `locations` array). Every one banks deterministically from a tappable card; Teach knows the whole vocabulary when you describe a flow.

05

Go Live — Phone Numbers & Web Chat

Overview

Your assistant exists as a draft until you take it live. Going live is the moment it becomes real: it gets a web address, can take a phone number, and starts answering. This chapter covers the full go-live surface — taking an assistant live and offline, buying and wiring a phone number, and putting the chat widget on your own website.

Take Your Assistant Live

`POST /api/v1/apps/{appId}/publish`

Requires `publish`.

```
curl -X POST https://api.everycallhandled.com/api/v1/apps/$APP_ID/publish \
  -H 'Authorization: Bearer YOUR_TOKEN' \
  -H 'Content-Type: application/json' \
  -d '{
    "subdomain": "harbour-hotel",
    "userEmail": "owner@harbourhotel.co.uk",
    "userPhone": "+447700900000"
  }'
```

To **apply later changes**, the app already has a subdomain; reuse it (only `subdomain` is required):

```
curl -X POST https://api.everycallhandled.com/api/v1/apps/$APP_ID/publish \
  -H 'Authorization: Bearer YOUR_TOKEN' \
  -d '{ "subdomain": "harbour-hotel" }'
```

Response (200):

```
{
  "appId": "...",
  "subdomain": "harbour-hotel",
  "deploymentUrl": "https://harbour-hotel.app.everycallhandled.com",
  "embedCode": "<script ...></script>",
  "status": "published",
  "publishedAt": "2026-05-10T10:30:00Z"
}
```

What going live does:

- **First time** — takes the app live: gives it its web address, applies your configuration (prompt, tools, voice, billing, calling setup), wires phone routing, and sends a welcome email if `userEmail` is provided.
- **Afterwards** — applies your draft changes to the live assistant. The subdomain and web address stay the same, and running balances and usage history are never touched — this changes behaviour, not billing state.

Going live completes within a few seconds. Until you take the assistant live, edits made via `PATCH /apps/{appId}` have no effect on what callers experience; `GET /apps/{appId}` tells you whether changes are waiting via `hasPendingPublish`.

Take Your Assistant Offline

`POST` /api/v1/apps/{appId}/unpublish

```
curl -X POST 'https://api.everycallhandled.com/api/v1/apps/{appId}/unpublish' \
  -H 'Authorization: Bearer YOUR_TOKEN'
```

Requires `publish`. Takes the live site offline immediately without deleting anything: the app, its knowledge, its conversation history and any phone number are all kept, and the subdomain is reserved so it can go live again on the same address. Conversations captured while live are hidden from listings until the app goes live again.

Response (200):

```
{ "success": true, "unpublished": true, "siteRemoved": true, "subdomainKept": "harbour-hotel" }
```

Calling it on an app that is not live returns `{ "success": true, "alreadyUnpublished": true }`.

Phone Number (buy + wire / release)

The fastest route to a phone-answering assistant: one call buys a number and wires it to the app.

`POST` /api/v1/apps/{appId}/phone-number `DELETE` /api/v1/apps/{appId}/phone-number

```
curl -X POST 'https://api.everycallhandled.com/api/v1/apps/{appId}/phone-number' \
  -H 'Authorization: Bearer YOUR_TOKEN' \
  -H 'Content-Type: application/json' \
  -d '{"country":"GB"}'
```

```
curl -X DELETE 'https://api.everycallhandled.com/api/v1/apps/{appId}/phone-number' \
  -H 'Authorization: Bearer YOUR_TOKEN'
```

Requires `edit_settings`.

POST searches and purchases a phone number for the requested country (GB regulatory requirements handled for you) and wires inbound calling in one call — one app, one number. A wiring failure automatically releases the just-purchased number and clears the app, so a paid number can never leak. DELETE unwires calling, releases the number (released numbers cannot be recovered), and clears it off the app.

Get Calling Status

`GET /api/v1/apps/{appId}/sip`

```
curl 'https://api.everycallhandled.com/api/v1/apps/{appId}/sip' \
  -H 'Authorization: Bearer YOUR_TOKEN'
```

Available to any signed-in member of the account that owns the app. Returns the app's current inbound-calling setup — whether calling is enabled, which number is wired, and the state of the underlying telephony resources.

Enable Calling (SIP)

If you manage your own numbers (for example, bringing an existing number or your own Twilio account — see Credentials), the SIP endpoints give you direct control of the wiring that the phone-number endpoint above automates.

`POST /api/v1/apps/{appId}/sip/enable`

```
curl -X POST 'https://api.everycallhandled.com/api/v1/apps/{appId}/sip/enable' \
  -H 'Authorization: Bearer YOUR_TOKEN' \
  -H 'Content-Type: application/json' \
  -d '{"twilioManaged":"true"}'
```

Requires `edit_settings`. Provisions the app's inbound-calling route; if `twilioManaged="true"`, also provisions a platform-managed number. Idempotent.

Disable Calling (SIP)

`POST /api/v1/apps/{appId}/sip/disable`

```
curl -X POST 'https://api.everycallhandled.com/api/v1/apps/{appId}/sip/disable' \
  -H 'Authorization: Bearer YOUR_TOKEN'
```

Requires `edit_settings`. Tears down the app's inbound-calling route; optionally releases the platform-managed number. Idempotent.

Set the Twilio Voice URL

`POST` /api/v1/apps/{appId}/voice/twilio-url

```
curl -X POST 'https://api.everycallhandled.com/api/v1/apps/{appId}/voice/twilio-url' \
  -H 'Authorization: Bearer YOUR_TOKEN' \
  -H 'Content-Type: application/json' \
  -d '{"voiceUrl":"https://example.com/your-twilio-webhook"}'
```

Requires `edit_settings`. Sets (or clears) the Voice URL on the app's **existing** Twilio number, so an incoming call routes to this assistant — for when you keep the number in your own Twilio account. The app must already have a `twilioPhoneNumber` configured (otherwise `400`).

Body: `{ "voiceUrl": "https://..." }` — pass `null` to clear.

Response (200): `{ "success": true, "phoneNumber": "+44...", "voiceUrl": "https://..." }`

Embed Code

`GET /api/v1/apps/{appId}/embed-code`

```
curl 'https://api.everycallhandled.com/api/v1/apps/{appId}/embed-code' \
  -H 'Authorization: Bearer YOUR_TOKEN'
```

Available to any signed-in member of the account that owns the app. Returns ready-to-paste iframe + `<script>` snippets for embedding the assistant on your own website. The app must be **published** first (otherwise `400`).

Response (200):

```
{
  "appId": "...",
  "appName": "...",
  "isPublished": true,
  "publishedUrl": "https://...",
  "embedCode": {
    "scriptTag": "<script src=\"https://...\"></script>",
    "iframe": "<iframe src=\"https://...\" ...></iframe>"
  },
  "urls": { "embedScript": "...", "widget": "...", "documentation": "..." },
  "configuration": {
    "position": ["bottom-right", "bottom-left"],
    "theme": ["auto", "light", "dark"],
    "mode": ["fab", "embedded"]
  }
}
```

The snippets themselves are intentionally shareable. Per-page embed options (start language, opening intent) are covered under Web-chat conversation config.

Ask Page — a QR-Scannable Page That IS Your Assistant

Some businesses have no website to embed into — or want a chat entry that lives in the physical world: a card at reception, a sticker on a product, a page in a holiday-let welcome book. The ask page is a hosted page that is nothing but your assistant, ready for a QR.

`POST /api/v1/apps/{appId}/ask-page`

```
curl -X POST 'https://api.everycallhandled.com/api/v1/apps/{appId}/ask-page' \
  -H 'Authorization: Bearer YOUR_TOKEN' \
  -H 'Content-Type: application/json' -d '{}'
```

Requires `edit_settings`. Returns `{ "url": "…", "key": "…" }`. The call is get-or-create: an app has one ask page, and repeated calls return the same one. In the dashboard, the web-chat settings section has a **Create an ask page** button that does this and shows the URL with a printable QR.

06

Bookings

Overview

Answering questions is good; taking bookings is better. Give your assistant a diary and it can offer real availability and book appointments during the conversation — on the phone, in web chat, and on a hosted booking page customers use directly.

There are three ways to run bookings, and this chapter covers all of them:

1. **The built-in diary** — the platform hosts the whole thing: your services, your weekly hours, one or more diary lanes, a public booking page with its own link and QR, and an operator diary in the dashboard. No external calendar needed. This is the richest option and the rest of this section documents it first.
2. **A connected calendar** (Google today, Microsoft 365 supported) — the assistant books straight into a calendar you already run.
3. **A supported third-party diary system** — see the Semble integration at the end of this chapter for the first of these.

The Built-in Diary

Set the booking provider to the built-in diary and configure your offerings in one place. Each service (a *type*) carries its display name, duration, price, optional variants, and its own weekly hours: `dayWindows` is both the day whitelist and the per-day hours map, so "Mon/Tue 9–8, Wed evenings only, Sun off" is one small object. Types can share one diary lane or declare their own `resources` (multiple lanes render side by side in the diary and are booked independently).

`PUT /api/v1/apps/{appId}/booking`

```
curl -X PUT 'https://api.everycallhandled.com/api/v1/apps/{appId}/booking' \
  -H 'Authorization: Bearer YOUR_TOKEN' \
  -H 'Content-Type: application/json' \
  -d '{
    "booking": {
      "enabled": true,
      "provider": "ech-native",
      "timezone": "Europe/London",
      "types": {
        "consultation": {
          "displayName": "New client consultation",
          "durationMins": 30,
          "priceLabel": "£50",
          "dayWindows": { "1": {"startHour": 9, "endHour": 20}, "2": {"startHour": 9,
"endHour": 20}, "6": {"startHour": 9, "endHour": 13} }
        }
      }
    }
  }'
```

Requires `manage_bookings`. The same endpoint reads back with `GET`. Everything here is also editable in the dashboard's Bookings tab, which for a built-in-diary app opens as a **workspace**: Overview, Diary, Customers, Messages, Reviews and Settings in a left rail. The diary supports drag-to-create (a booking or a private block), drag-to-reschedule, and a **Close dates** action for holidays and closures (documented below).

The Hosted Booking Page

Every built-in-diary app gets a public booking page at `/book/{appId}` on the platform's public site — brandable (accent colour, logo, welcome text, photos), with service search, category grouping, and a persistent web-chat bubble so a visitor can ask questions mid-booking. The Bookings settings section gives you the **share link, a printable QR, and two embed snippets** (fixed and auto-resizing iframe) for your own website. Share the link anywhere customers are: texts, email signatures, your Google Business profile.

Public Booking API

The hosted page runs on a small unauthenticated API you can also drive directly — for a custom front end, a kiosk, or your own site. All routes live under `/api/v1/public/book/{appId}/…` and need no token; they only ever expose what the public page itself shows.

GET `/api/v1/public/book/{appId}/config` **GET** `/api/v1/public/book/{appId}/slots`

POST `/api/v1/public/book/{appId}/hold` **POST** `/api/v1/public/book/{appId}/confirm`

POST `/api/v1/public/book/{appId}/release` **POST** `/api/v1/public/book/{appId}/manage`

POST `/api/v1/public/book/{appId}/manage-cancel`

POST `/api/v1/public/book/{appId}/manage-reschedule`

POST `/api/v1/public/book/{appId}/review` **GET** `/api/v1/public/book/{appId}/reviews`

POST `/api/v1/public/book/{appId}/voucher-checkout`

POST `/api/v1/public/book/{appId}/voucher-check`

POST `/api/v1/public/book/{appId}/promo-check`

```
curl 'https://api.everycallhandled.com/api/v1/public/book/{appId}/slots?
type=consultation&days=7'
```

Holds are short-lived by design: confirm promptly or the slot returns to the pool. Manage links are signed and expiring — customers can move or cancel their own booking within the policy window you set; the operator endpoints below are never policy-gated, because the business is the business. The manage view also carries your configured `cancellationPolicy` wording, shown to the customer before and after they cancel.

`config` returns everything the page renders: services (with per-service `depositDescription` when your deposit wording is a customer-facing sentence), branding (including a choosable `backgroundColor`), terms, `detailFields`, `newClientGate`, `giftVouchers`, `promoCodesEnabled`, and `voiceDefaults` — the vertical-correct default wording used wherever you haven't set your own (see **Your voice** below).

`confirm` accepts more than name/phone/email. Depending on your configuration it also takes the detail fields (`dob`, `gender`, `addr1`, `addr2`, `county`, `postcode`, `practitionerNote`, `marketingConsent`), the new-client answer (`newClient`), a gift-voucher code (`voucherCode`) and a promo code (`promoCode`). Server-side validation always matches what the page enforces.

Instant Confirmations & Booking Messages

Every booking made through the page, the assistant, or the API sends an instant confirmation — email whenever an address is known, SMS whenever a mobile is — and cancellations and reschedules send notices from any direction. The wording is honest by construction: while a deposit link is outstanding the message says the booking is **reserved** and the deposit secures it; "confirmed" is only ever said once nothing is owed. Your per-service deposit wording (see **Your voice**) rides along wherever the deposit is mentioned. Switch the whole thing off with `"confirmationComms": false` in the booking settings if you send your own.

What the Page Collects

Two settings shape the details step, both editable in the dashboard and via the booking settings `PUT`:

- `detailFields` — per-app because a barber should never ask for a date of birth: `dob`, `gender` and `address` are each `"optional"` or `"required"` (absent = not asked); `note` (`"optional"`) adds a free-text note for the practitioner; `marketingConsent: true` adds a consent tick-box whose answer is stored on the customer record with source and timestamp. `marketingConsentText` lets you phrase the consent ask in your own legal voice.
- `newClientGate` — the "have you visited us before?" rule: `{"enabled": true, "consultationKey": "consultation", "message": "…"}` steers first-timers to a consultation before any treatment, and the server enforces it — the page asks, but the API can't be talked around it either.

Gift Vouchers & Promo Codes

Enable `giftVouchers` and the hosted page grows a voucher storefront: brand-toned gift cards for each option, paid through your own connected Stripe, the code emailed to the buyer the moment payment completes. Codes are redeemable at booking: a voucher that covers the deposit pays it outright (no payment link at all); a smaller one is noted on the booking for the till, with your configured `redemptionNote` telling the customer exactly what happens to it.

```
{
  "giftVouchers": {
    "enabled": true,
    "validityMonths": 12,
    "options": [ {"amountGBP": 25, "title": "A special treat for you"}, {"amountGBP": 50,
"title": "Happy Christmas"} ],
    "redemptionNote": "It will be deducted from your treatment cost at your appointment."
  },
  "promoCodes": [ {"code": "WINTER10", "kind": "percent", "value": 10, "description": "Winter
offer"} ]
}
```

`promoCodes` are validated at checkout (`kind` is `percent` or `fixed` , with optional expiry and an `active` toggle) and recorded on the booking for the till; `promoNote` phrases what the customer is told when a code applies. The public `voucher-check` and `promo-check` routes power the page's Apply buttons and answer honestly — an exhausted voucher says it has been used in full, an expired one says it has expired.

Your Voice — Customer-Facing Wording

Everything your customers read is your wording, never ours — each line is configuration with a sensible neutral default (the dashboard gathers them all in one **Your voice** panel):

Field	What it phrases
`greetingStyle`	`"friendly"` ("Hi Jason,") or `"formal"` ("Dear Mr Edge,") in every booking email — formal keeps the title the customer gave
`types.{key}.paymentDescription`	Your deposit policy, spoken wherever that service's deposit is mentioned: the catalogue, the confirmed screen, the email, the SMS. Write a sentence ("This deposit is deducted from your treatment cost.") and it speaks everywhere; a label form ("Deposit: …") only names the Stripe payment line
`giftVouchers.redemptionNote`	What happens to a voucher that doesn't cover the deposit, and the line in the voucher email
`promoNote`	What customers are told when a promo code applies
`detailFields.marketingConsentText`	The consent ask beside the tick-box
`cancellationPolicy`	Shown before and after a customer cancels
`terms.text`	The acceptance statement, recorded verbatim on every booking
`newClientGate.message`	What first-timers see when steered to a consultation
`vocabulary`	`"clinic"`, `"salon"`, `"trades"` or `"hospitality"` — relabels the workspace (Patients, Clients, Guests, jobs) AND picks the vertical-correct default wording for every line you leave blank

All of it rides the same booking settings `PUT`; leave any field blank and the customer sees the neutral default for your vocabulary.

Operator Diary API

Everything the dashboard diary does is plain API, all requiring `manage_bookings`:

`GET /api/v1/apps/{appId}/bookings/diary?start={unix}&end={unix}`

`POST /api/v1/apps/{appId}/bookings/create`

`POST /api/v1/apps/{appId}/bookings/{bookingId}/cancel`

`POST /api/v1/apps/{appId}/bookings/{bookingId}/reschedule`

`POST /api/v1/apps/{appId}/bookings/{bookingId}/notes`

`POST /api/v1/apps/{appId}/bookings/close-dates` `GET /api/v1/apps/{appId}/booking-reports`

Create takes `{"kind": "booking", "startUnix": …, "endUnix": …, "offeringKey": "consultation", "partyName": "Jane Doe"}` for a manual booking, or `{"kind": "block", "startUnix": …, "endUnix": …, "label": "Lunch"}` for a private block — blocks make the time unbookable but are never shown to customers or callers.

Close dates writes a full-day private block per day per lane across an inclusive range — one call for "closed 20–28 August":

```
curl -X POST 'https://api.everycallhandled.com/api/v1/apps/{appId}/bookings/close-dates' \
  -H 'Authorization: Bearer YOUR_TOKEN' \
  -H 'Content-Type: application/json' \
  -d '{"fromDate": "2026-08-20", "toDate": "2026-08-28", "resourceId": "all", "label": "Summer closure"}'
```

Days already in the past are skipped, today is clamped to now, and existing bookings on closed days are deliberately left in place — the response lists what was created, skipped and (rarely) failed, so you can deal with clashing bookings personally rather than have them silently cancelled.

Customers

The diary quietly builds a customer directory: every booking's party (name, phone, and their booking history) is queryable and editable, with duplicate-merge built in.

`GET /api/v1/apps/{appId}/parties` `POST /api/v1/apps/{appId}/parties`

`GET /api/v1/apps/{appId}/parties/{partyId}` `POST /api/v1/apps/{appId}/parties/{partyId}`

`POST /api/v1/apps/{appId}/parties/{partyId}/merge`

Requires `manage_bookings`. In the dashboard this is the workspace's Customers section, with segmentation filters and export.

The Booking Record

Every booking carries a **record** — the professional evidence trail behind the appointment: what was done and used, permanent notes, and a completeness checklist. In the dashboard it opens from the diary or a customer's history ("Open record"); over the API it is four routes, all requiring `manage_bookings`:

```
GET /api/v1/apps/{appId}/bookings/{bookingId}/record

POST /api/v1/apps/{appId}/bookings/{bookingId}/record/delivery

POST /api/v1/apps/{appId}/bookings/{bookingId}/record/note

POST /api/v1/apps/{appId}/bookings/{bookingId}/record/artefact
```

GET record returns the profile in force (today the built-in `generic-v1` delivery-record profile), the booking's headline facts, the `checklist` (each item's `kind`, `label` and `status` of `completed` / `waived` / `missing`), the ordered `notes`, and the `delivery` work-details block with who recorded it and when.

Work details (`record/delivery`) takes `{"lines": [...]}` — up to 20 lines, each `{"item": "Annual boiler service", "product"?: "Worcester Greenstar 30i", "batch"?: "…", "expiry"?: "…", "quantity"?: 1, "priceGBP"?: 95, "comment"?: "…"}`. `item` is required per line; the batch/expiry fields exist so a clinic can record exactly what was administered — product, batch and expiry per treatment, the pharmacovigilance shape. Saving again replaces the lines (it is a statement of what happened, not a log), and every save is stamped with the practitioner and time.

Notes (`record/note`) takes `{"content": "…", "correctsNoteId"?: "…"}` and is **append-only by construction**: a saved note is stamped with author name and time and can never be edited or deleted through any API. A correction is a new note carrying `correctsNoteId`, so the full history always survives. Content is capped at 2,000 characters; the route returns the new `noteId`.

Checklist (`record/artefact`) takes `{"kind": "delivery-details" | "notes", "status": "completed" | "waived", "waiveReason"?: "…"}`. Waiving **requires a written reason** (422 without one), and the reason is kept on the record with who waived and when.

Nothing in the record is ever shown to customers — it is the operator's own evidence trail, attached to the booking permanently. The current generic profile deliberately excludes special-category clinical data (photos, consent capture, sign-off locks); richer regulated profiles arrive as separate reviewed documents.

Reviews

Post-visit review requests feed a moderated review store: list, publish/unpublish and remove via `GET /api/v1/apps/{appId}/reviews`, `PATCH /api/v1/apps/{appId}/reviews/{reviewId}` and `DELETE`. Published reviews render on the hosted booking page.

Booking Settings & Connection State

`GET /api/v1/apps/{appId}/booking`

```
curl 'https://api.everycallhandled.com/api/v1/apps/{appId}/booking' \
  -H 'Authorization: Bearer YOUR_TOKEN'
```

Requires `manage_bookings`.

Response (200): the app's booking settings plus the connection state:

```
{
  "booking": { /* schedule, appointment types, ... */ },
  "connection": { "status": "connected", "connectedEmail": "owner@example.com" }
}
```

`status` is `connected` or `disconnected`. Apps using the Semble integration always report `connected` — their diary lives in Semble itself.

Connect a Calendar (Google / Microsoft 365)

`GET /api/v1/apps/{appId}/booking/authorize?provider=google`

```
curl 'https://api.everycallhandled.com/api/v1/apps/{appId}/booking/authorize' \
  -H 'Authorization: Bearer YOUR_TOKEN'
```

Requires `manage_bookings`. Returns a signed hosted-consent URL the calendar owner opens in their browser to grant access. `provider` is `google` (default) or `microsoft`. Nothing connects until they complete the consent screen; they land back on the app's Bookings tab afterwards.

Response (200):

```
{ "configured": true, "provider": "google", "url":
"https://api.eu.nylas.com/v3/connect/auth?..." }
```

`{ "configured": false, "url": null }` means calendar booking is not enabled for your account — contact support.

List Upcoming Bookings

```
GET /api/v1/apps/{appId}/bookings?days=7
```

```
curl 'https://api.everycallhandled.com/api/v1/apps/{appId}/bookings' \
  -H 'Authorization: Bearer YOUR_TOKEN'
```

Requires `manage_bookings`. Lists every upcoming appointment across the app's connected calendar(s) for the next `days` days (1-31, default 7) — the business-owner view of what the assistant has booked. Currently supported for calendar-connected apps; apps on the Semble integration return `supported: false` with a message pointing at the Semble diary, since their appointments live there.

Response (200):

```
{
  "provider": "nylas",
  "supported": true,
  "windowDays": 7,
  "count": 2,
  "bookings": [
    { "eventId": "...", "date": "2026-07-18", "time": "10:30", "label": "Check-up", "title":
"Mrs Hughes — Check-up" }
  ]
}
```

Integration: Semble (a practice management system used by healthcare clinics)

What is Semble? Semble is a practice management system widely used by private healthcare clinics in the UK — it holds the clinic's diary, its appointment types and its records. If your business doesn't use Semble, skip this whole section: the generic calendar booking above is the path for you. Semble is the first of a growing set of third-party system integrations; others will follow the same pattern — a special-case configuration surface layered on top of the generic booking experience.

For a Semble-connected app, appointment booking is driven by a declarative **booking config** — a set of named appointment *types*, each binding a Semble product/location to the rules the assistant follows when it offers and books slots (practitioner binding, new-patient flow, scan horizon, per-day hours). The config is applied to the live assistant immediately on write.

Every route in this section Requires `manage_booking_config` — an **opt-in only** permission: it is *not* part of any role baseline, not even Owner. It's granted per user with an explicit `+manage_booking_config` override from the Team page in your dashboard. Don't assume an Owner token can hit these endpoints — verify the grant.

Read the current config

```
GET /api/v1/apps/{appId}/booking-config
```

```
curl https://api.everycallhandled.com/api/v1/apps/$APP_ID/booking-config \
  -H 'Authorization: Bearer YOUR_TOKEN'
```

Response (200):

```
{
  "appointmentBooking": {
    "types": {
      "initial_consult": {
        "displayName": "New Patient Consultation",
        "sembleProductId": "prod_8812",
        "locationId": "loc_04",
        "practitionerId": "usr_231",
        "practitionerName": "Dr Amara Okafor",
        "duration": 30,
        "slotAlignment": 15,
        "leadTimeMins": 120,
        "nextAvailableScanDays": 21,
        "allowedDays": [1, 3, 5],
        "startHour": 9,
        "endHour": 17,
        "position": 1,
        "newPatientFlow": true,
        "newPatientFormUrl": "https://forms.gle/zTEZEZv9example"
      },
      "follow_up": {
        "displayName": "Follow-up",
        "sembleProductId": "prod_8815",
        "locationId": "loc_04",
        "duration": 15,
        "slotAlignment": 15,
        "leadTimeMins": 60,
        "nextAvailableScanDays": 14,
        "allowedDays": [1, 2, 3, 4, 5],
        "position": 2
      }
    }
  },
  "hasSemble": true,
  "editable": true
}
```


`hasSemble` tells an editor whether the Semble integration is wired up for this app (and therefore whether the catalog picker below will return anything).

Write the config

`PUT /api/v1/apps/{appId}/booking-config`

The body must carry a top-level `appointmentBooking` object with a `types` map. The write **replaces only the `appointmentBooking` sub-key** and preserves every sibling verbatim; live

calls pick the change up immediately.

```
curl -X PUT https://api.everycallhandled.com/api/v1/apps/$APP_ID/booking-config \
  -H 'Authorization: Bearer YOUR_TOKEN' \
  -H 'Content-Type: application/json' \
  -d '{
    "appointmentBooking": {
      "types": {
        "initial_consult": {
          "displayName": "New Patient Consultation",
          "sembleProductId": "prod_8812",
          "locationId": "loc_04",
          "duration": 30,
          "slotAlignment": 15,
          "leadTimeMins": 120,
          "nextAvailableScanDays": 21,
          "allowedDays": [1, 3, 5],
          "position": 1
        }
      }
    }
  }'
```

A successful `PUT` returns the same shape as the `GET` (re-read after write). Omit `appointmentBooking` and you get `400 appointmentBooking is required`.

The per-type schema

Each entry under `types` is keyed by a **type key** — the machine identifier, which must match `^[a-z0-9_]+$` (e.g. `initial_consult`). Fields on the type object:

Field	Type	Notes
`displayName`	string	The label the assistant speaks to the caller. Trimmed on write.
`sembleProductId`	string	The Semble appointment-type/product this maps to.
`locationId`	string	The Semble location the booking is created against.
`practitionerId` / `practitionerName`	string	Optional Semble practitioner binding.
`duration`	number	Appointment length in minutes. Must be **positive**.
`slotAlignment`	number	Slot grid granularity in minutes (e.g. 15). Must be **positive**.
`allowedDays`	int[]	Bookable weekdays, `0` =Sun … `6` =Sat. Deduped and sorted on write; each entry must be a whole number 0–6. Empty/absent = no day restriction.
`leadTimeMins`	number	Minimum notice before a slot can be booked. Must be **>= 0**.

Field	Type	Notes
`nextAvailableScanDays`	int	How many days ahead the assistant scans for the next free slot. Whole number **1–70**.
`position`	int	Optional presentation order (see below). Must be **>= 1**.
`startHour / endHour`	number	Optional default business hours for the type. When either is set, both must satisfy `0 <= startHour < endHour <= 23`.
`dayWindows`	object	Optional per-day hours override, `{ "1": { "startHour": 9, "endHour": 15 } }`. Keys 0–6; same hour rule. Takes precedence over the flat `startHour` / `endHour`, and doubles as the day whitelist where hours differ by day.
`newPatientFlow`	bool	Marks this as a Semble new-patient type (triggers the intake/questionnaire flow).
`newPatientFormUrl`	string	External questionnaire URL. **Required when the Semble new-patient flow is on** — unless the app's hosted intake form is active (see Intake Forms), which replaces it.

Unknown keys on a type are preserved (forward-compat), so a newer field survives a round-trip through an older editor.

Position ordering. `position` is **1-based** (`1` = first). It is *the* order the assistant offers types in — it feeds the booking tool's options and the assistant's coaching, and drives card lists in chat/voice. So "always offer the consultation type first" is configuration, not model judgement. Types **without** a `position` fall *after* all positioned types, in stored-object order. Positioned types sort ascending by `position`, ties broken by stored order.

Semble catalog (live pickers)

```
GET /api/v1/apps/{appId}/booking-config/semble-catalog
```

```
curl 'https://api.everycallhandled.com/api/v1/apps/{appId}/booking-config/semble-catalog' \
  -H 'Authorization: Bearer YOUR_TOKEN'
```

Returns live Semble products, practitioners, and locations so an editor can render real pickers instead of asking people to hand-type IDs.

```
{
  "products": [
    { "id": "prod_8812", "name": "New Patient Consultation", "duration": 30, "locationId": null
}
  ],
  "practitioners": [
    { "id": "usr_231", "name": "Dr Amara Okafor" }
  ],
  "locations": [
    { "id": "loc_04", "name": "Harley Street — Main" }
  ]
}
```

This call reaches out to Semble live. If the app has no Semble credential configured, or Semble can't be reached, you get empty arrays plus an `error` string explaining why — the endpoint never hard-fails:

```
{ "products": [], "practitioners": [], "locations": [], "error": "Semble integration is not
configured for this app." }
```

Validation rules (enforced on PUT)

- `types` must be an object; each type key must match `^[a-z0-9_]+$`, and duplicate keys are rejected.
- `duration` and `slotAlignment`, if present, must be positive numbers.
- `leadTimeMins` >= 0; `position` >= 1; `nextAvailableScanDays` a whole number 1–70.
- `allowedDays` must be an array of whole numbers 0–6.
- `startHour` / `endHour` (and every `dayWindows` entry) must satisfy `0 <= startHour < endHour <= 23`.
- The Semble new-patient flow requires a `newPatientFormUrl` unless the hosted intake form is active on the app.

All errors are collected and returned together as a `400` with a `; `-joined message.

How availability is actually decided

Availability for a Semble-connected app is the product of three independent layers, and the config is only one of them:

1. **Semble's own schedule** — Semble returns the raw set of open slots for the product/practitioner/location. If Semble says a slot doesn't exist, no config can conjure it.
2. **The type's config rules** — the assistant then filters those slots by `allowedDays` (or per-day `dayWindows`), business hours (`startHour` / `endHour`), `leadTimeMins`, and the `nextAvailableScanDays` horizon. This is the layer you control here.
3. **The assistant's prompt** — how types are offered, sequenced (via `position`) and spoken about on the call.

A slot is only offered when all three agree. So if a Wednesday slot never comes up, check in order: does Semble show it open, do `allowedDays` / `dayWindows` include Wednesday within hours, and does the prompt surface that type at all.

07

Intake Forms

Overview

Some conversations need more than a chat can comfortably collect — full contact details, structured questionnaires, sign-up information. Intake forms solve this: during a booking, the platform can text the person a link to a hosted form; their answers come back as intake records your team can read, and are then handed on to your connected system and purged on a schedule. No third-party form builder, no copy-paste.

The two endpoints here are the read surface over those records — the list your team works from, and the detail view they read the answers from. Both are gated by `view_intakes`, which is its own permission (rather than riding along with booking config) because submissions can carry sensitive personal details. The baseline is **Owner + Manager**; it's revocable per user with `-view_intakes` and grantable to staff with `+view_intakes`.

List Intake Submissions

`GET /api/v1/apps/{appId}/intakes`

Requires `view_intakes`. Returns answer-free **summary rows** — never the questionnaire answers. Supports a status filter and offset/limit pagination.

Query params:

- `status` — filter to one lifecycle status (`pending` | `received` | `transferred` | `purged`).
- `limit` — page size, default `50`, capped at `200`.
- `offset` — default `0`.

```
curl "https://api.everycallhandled.com/api/v1/apps/$APP_ID/intakes?status=received&limit=50" \
  -H 'Authorization: Bearer YOUR_TOKEN'
```

Response (200):

```
{
  "intakes": [
    {
      "id": "el_intake_9f2a",
      "status": "received",
      "firstName": "Priya",
      "phone": "+447700900123",
      "typeKey": "initial_consult",
      "bookingId": "bk_5521",
      "submittedAt": "2026-07-10T14:32:07.000Z",
      "transferredAt": "",
      "created": "2026-07-10T13:05:00.000Z"
    }
  ],
  "total": 1,
  "limit": 50,
  "offset": 0
}
```

`total` is the full count after the status filter; `intakes` is the current page. Rows are sorted newest-first by creation time, and include every submission for the app, whether it arrived before or after the app went live.

Get One Submission (including answers)

```
GET /api/v1/apps/{appId}/intakes/{intakeId}
```

```
curl 'https://api.everycallhandled.com/api/v1/apps/{appId}/intakes/{intakeId}' \
  -H 'Authorization: Bearer YOUR_TOKEN'
```

Requires `view_intakes`. Returns the full record **including the `answers` blob** — the read your team uses for a confirming call, or for manual entry into a connected system. The record must belong to this app; a mismatched or unknown id returns `404 Intake not found`. The form link's security token is never returned.

Response (200):

```
{
  "id": "el_intake_9f2a",
  "status": "received",
  "appId": "app_9f31",
  "bookingId": "bk_5521",
  "patientId": "",  /* the matched record id in a connected system such as Semble */
  "firstName": "Priya",
  "phone": "+447700900123",
  "typeKey": "initial_consult",
  "submittedAt": "2026-07-10T14:32:07.000Z",
  "transferredAt": "",
  "purgeAfter": "",
  "formVersion": 1,
  "answers": {
    "firstName": "Priya",
    "lastName": "Sharma",
    "dob": "1990-04-12",
    "email": "priya@example.com",
    "postcode": "W1G 6BW",
    "payer": "Self-Pay"
  }
}
```

Status Lifecycle

An intake moves through four statuses:

- **`pending`** — the booking created the record and texted the signed link, but the person hasn't submitted yet. No answers on file.
- **`received`** — the person submitted the form. The `answers` blob is populated and `submittedAt` is stamped. Resubmitting on an open link replaces the answers in place.
- **`transferred`** — the platform has written the details into your connected system (for example Semble) and confirmed it. `transferredAt` is stamped, and `purgeAfter` is set to the retention deadline. Answers are still readable during the grace window.
- **`purged`** — the grace window elapsed and the answers blob has been wiped. The row still exists (status, identity, timestamps) but `answers` is gone.

Transfer-then-purge retention

Intake answers can carry sensitive personal data, so they're held only as long as they're operationally needed. Retention runs in two phases:

1. **Transfer** — a `received` intake with a matched record is written into your connected system, then marked `transferred` and stamped `purgeAfter = now + purgeGraceDays` (the app's configured grace window, **default 7 days**, range 1–60). A `received` record with no matched record or a corrupted/empty answers blob is left alone for a human to sort out — it is never silently dropped.
2. **Purge** — once a `transferred` intake passes its `purgeAfter` deadline, the answers are blanked and the status becomes `purged`. The wipe is verified before the record is reported as purged.

Treat `answers` as **transient**: available while `pending` → `received` → `transferred`, and gone once `purged`. Build any downstream capture of intake answers to happen before the grace window closes; after that, only the summary metadata survives.

08

See What's Happening

Overview

Once your assistant is live, this chapter is where you'll spend your time: every call and chat with its transcript and summary, recordings, an AI quality reviewer, a persistent memory of who's called before, a log of every third-party call your integrations made, and a full history of every configuration change with one-call restore.

List Conversations

Each completed call or chat is stored with its full transcript, caller metadata, sentiment scores and analysis.

```
GET /api/v1/apps/{appId}/conversations?limit=50&offset=0
```

```
curl 'https://api.everycallhandled.com/api/v1/apps/{appId}/conversations' \
  -H 'Authorization: Bearer YOUR_TOKEN'
```

Available to any signed-in member of the account that owns the app.

Query params: `limit` (default 50), `offset` (pagination), `mode` . Pagination is offset-based.

Response (200):

```
{
  "conversations": [ /* conversation rows — see Get a Single Conversation for field names */ ],
  "pagination": { "total": 248, "limit": 50, "offset": 0, "hasMore": true, "nextOffset": 50 },
  "isAdmin": false,
  "mode": "all"
}
```

Get a Single Conversation

`GET /api/v1/apps/{appId}/conversations/{chatId}`

```
curl 'https://api.everycallhandled.com/api/v1/apps/{appId}/conversations/{chatId}' \
  -H 'Authorization: Bearer YOUR_TOKEN'
```

Available to any signed-in member of the account that owns the app. Returns the full transcript + metadata:

```
{
  "chatId": "call-_+447941190142_abc123",
  "source": "voice_phone",
  "contactId": "...",
  "transcript": [
    { "id": "...", "role": "ai", "message": "Hello, thank you for calling...", "createdAt":
"..." },
    { "id": "...", "role": "user", "message": "Hi, I'd like to book...", "createdAt": "..." }
  ],
  "chatSummary": "...",
  "chatSentiment": "positive",
  "sentimentScore": 8,
  "durationSeconds": 145,
  "createdAt": "..."
}
```

Get a Recording

`GET /api/v1/apps/{appId}/conversations/{chatId}/recording`

```
curl 'https://api.everycallhandled.com/api/v1/apps/{appId}/conversations/{chatId}/recording' \
  -H 'Authorization: Bearer YOUR_TOKEN'
```

Available to any signed-in member of the account that owns the app. Returns a secure, time-limited download link (15-minute expiry), or 404 if no recording exists.

Response (200): `{ "url": "https://...", "expiresIn": 900, "contentLength": 123456, "format": "mp3" }` (plus `filteredUrl` when a noise-filtered version exists)

Append a Message (continue session)

`POST /api/v1/apps/{appId}/conversations/{chatId}/messages`

```
curl -X POST
'https://api.everycallhandled.com/api/v1/apps/{appId}/conversations/{chatId}/messages' \
  -H 'Authorization: Bearer YOUR_TOKEN' \
  -H 'Content-Type: application/json' \
  -d '{"role":"user","content":"One more thing, can we move it to 3pm?"}'
```

Used for web-chat continue-session flows. Rare in API integrations.

Import Conversations

`POST /api/v1/apps/{appId}/conversations/import`

```
curl -X POST 'https://api.everycallhandled.com/api/v1/apps/{appId}/conversations/import' \
  -H 'Authorization: Bearer YOUR_TOKEN' \
  -H 'Content-Type: application/json' \
  -d '{"chatId":"legacy-001","userIdentifier":"+447700900123","source":"import","messages":
[{"role":"user","content":"Hi"},{"role":"ai","content":"Hello, how can I help?"}]}'
```

Imports historical conversations — useful when migrating from another system so your history lives in one place.

Body: `{ "chatId": "...", "userIdentifier": "...", "messages": [ { "role": "user"|"ai", "content": "..." } ], "source": "..." }` — `userIdentifier` and a non-empty `messages[]` are required.

Dashboard Summary

`GET /api/v1/apps/{appId}/dashboard-summary`

```
curl 'https://api.everycallhandled.com/api/v1/apps/{appId}/dashboard-summary' \
  -H 'Authorization: Bearer YOUR_TOKEN'
```

Available to any signed-in member of the account that owns the app. A per-app aggregate snapshot — daily call volume, conversion rate, dropped-call rate, abandoned/mid-drop breakdown, recent-issues count. One call instead of N queries for rendering an app overview.

Conversation Reviews

The platform can run an AI reviewer over each completed call to assess assistant quality — flagging issues with severity tags (info / warn / critical) and suggesting prompt improvements. Review cadence and model are configured per app; the review data itself is what you see in the dashboard's quality surfaces.

Review Settings

`PATCH /api/v1/apps/{appId}/review-settings`

Requires `enable_reviews`.

```
curl -X PATCH https://api.everycallhandled.com/api/v1/apps/$APP_ID/review-settings \
  -H 'Authorization: Bearer YOUR_TOKEN' \
  -H 'Content-Type: application/json' \
  -d '{
    "reviewCadence": "per-call",
    "reviewModel": "sonnet"
  }'
```

Body:

- `reviewCadence` — `off` (default), `per-call`, `daily`, `weekly`. Default `off` means no reviews until explicitly enabled per app.
- `reviewModel` — `sonnet` (default), `opus`, `haiku`

Caller Profiles

Your assistant can remember the people who contact it. When enabled, the platform builds a persistent **caller profile** per phone number per app: identity (name, organisation), a rolling AI-written summary refreshed each call, structured traits with confidence scores, and **open issues** — thread-style records that persist across calls until resolved. A returning caller gets an assistant that already knows the context.

Scope: profiles are per `(appId, phoneNumber)`. Two callers with the same phone but different apps are distinct records. Do not assume cross-app continuity.

List Caller Profiles

`GET /api/v1/apps/{appId}/caller-profiles`

```
curl 'https://api.everycallhandled.com/api/v1/apps/{appId}/caller-profiles' \
  -H 'Authorization: Bearer YOUR_TOKEN'
```

Requires `view_caller_profiles`. Query params: `?q=<substring>&limit=200&offset=0&includeArchived=false` (`q` is a substring match across the profile; there is no dedicated `phone` / `since` filter). Returns the list of contacts (phone, name, last-called-at, callCount).

Get a Caller Profile

```
GET /api/v1/apps/{appId}/caller-profiles/{contactId}
```

```
curl 'https://api.everycallhandled.com/api/v1/apps/{appId}/caller-profiles/{contactId}' \
  -H 'Authorization: Bearer YOUR_TOKEN'
```

Requires `view_caller_profiles`. Returns the full profile — identity, rolling summary, traits with confidence, open issues, complete call history references, operator notes.

Update a Caller Profile

```
PUT /api/v1/apps/{appId}/caller-profiles/{contactId}
```

```
curl -X PUT 'https://api.everycallhandled.com/api/v1/apps/{appId}/caller-profiles/{contactId}'
\
  -H 'Authorization: Bearer YOUR_TOKEN' \
  -H 'Content-Type: application/json' \
  -d '{"name":"Jane Smith"}'
```

Requires `edit_caller_profiles`. Human-side updates — correct an extracted name, adjust a trait, add a note. Body shape mirrors the GET response with optional fields. AI-extracted fields can be overridden; future profile updates respect high-confidence human edits.

Delete a Caller Profile (right to be forgotten)

```
DELETE /api/v1/apps/{appId}/caller-profiles/{contactId}
```

```
curl -X DELETE 'https://api.everycallhandled.com/api/v1/apps/{appId}/caller-
profiles/{contactId}' \
  -H 'Authorization: Bearer YOUR_TOKEN'
```

Requires `delete_caller_profile` (owner default). Supports GDPR Article 17: removes the profile, the caller's issue records and analytics keyed to them. Transcripts of past calls remain in your conversation records (needed for service-delivery and accounting) but their linkage to the deleted profile is severed. An audit entry records who, when, why. Irreversible.

Export Caller Profiles (CSV)

```
GET /api/v1/apps/{appId}/caller-profiles.csv
```

```
curl 'https://api.everycallhandled.com/api/v1/apps/{appId}/caller-profiles.csv' \
  -H 'Authorization: Bearer YOUR_TOKEN'
```

Requires `view_caller_profiles`. Bulk export across all profiles — supports GDPR Article 20 (data portability). Returns standard CSV with one row per contact + their analytics columns. Issue detail comes via the issues CSV (below). (There is no single-profile `.csv` endpoint — use the per-contact JSON detail route for one caller.)

Per-Caller Analytics

```
GET /api/v1/apps/{appId}/caller-profiles/{contactId}/analytics
```

```
curl 'https://api.everycallhandled.com/api/v1/apps/{appId}/caller-
profiles/{contactId}/analytics' \
  -H 'Authorization: Bearer YOUR_TOKEN'
```

Requires `view_caller_profiles`. Call volume, common topics, conversion rates, average call length, etc. for one caller.

Find Duplicates

```
GET /api/v1/apps/{appId}/caller-profiles/{contactId}/duplicates
```

```
curl 'https://api.everycallhandled.com/api/v1/apps/{appId}/caller-
profiles/{contactId}/duplicates' \
  -H 'Authorization: Bearer YOUR_TOKEN'
```

Requires `view_caller_profiles`. Returns only contacts matching the target's phone **within the same `{appId}`** — duplicates are strictly per-app.

Merge Duplicates

`POST /api/v1/apps/{appId}/caller-profiles/{contactId}/merge`

```
curl -X POST 'https://api.everycallhandled.com/api/v1/apps/{appId}/caller-
profiles/{contactId}/merge' \
  -H 'Authorization: Bearer YOUR_TOKEN' \
  -H 'Content-Type: application/json' \
  -d '{"sources":["contact-id-to-merge"]}'
```

Requires `edit_caller_profiles`. Strictly per-app: any `sources[]` entry whose contact belongs to a different app than the URL `{appId}` is refused with a per-source error in the response `errors[]` array (in-scope sources in the same payload still process normally). The target's own app is validated up front; a target-app mismatch returns **400** for the whole request. Cross-app merging is not performed regardless of caller role.

Caller Issues

Issues are persistent records tied to a caller; they have a `status` (open / monitoring / completed), `kind`, `summary`, `severity`, `lastObservedAt`, and notes. The assistant sees open issues on the next call from that number (when profile-apply is on), so "the engineer never called back" doesn't get forgotten between calls.

`GET /api/v1/apps/{appId}/caller-profiles/{contactId}/issues`

`GET /api/v1/apps/{appId}/issues` `GET /api/v1/apps/{appId}/issues.csv?since=...&until=...`

```
curl 'https://api.everycallhandled.com/api/v1/apps/{appId}/caller-profiles/{contactId}/issues'
\
  -H 'Authorization: Bearer YOUR_TOKEN'
```

```
curl 'https://api.everycallhandled.com/api/v1/apps/{appId}/issues' \
  -H 'Authorization: Bearer YOUR_TOKEN'
```

```
curl 'https://api.everycallhandled.com/api/v1/apps/{appId}/issues.csv' \
  -H 'Authorization: Bearer YOUR_TOKEN'
```

Requires `view_caller_profiles`. The app-wide `/issues` form returns the consolidated issues view — typically used for daily triage. Query params: `?status=open&severity=critical&since=...`.

Update an Issue

`PUT /api/v1/apps/{appId}/issues/{issueId}`

```
curl -X PUT 'https://api.everycallhandled.com/api/v1/apps/{appId}/issues/{issueId}' \
  -H 'Authorization: Bearer YOUR_TOKEN' \
  -H 'Content-Type: application/json' \
  -d '{"status":"resolved","notes":"Resolved on callback."}'
```

Requires `edit_caller_profiles`. Update an issue's status and notes — close it out, mark it monitoring, or add context for the next person.

Third-Party Call Log

Every outbound HTTP call your app makes to a third party during a conversation is recorded as one row in the call log — a flight recorder for the integrations wired into an app, so you can see what the app called, when, and whether it succeeded.

```
GET /api/v1/apps/{appId}/external-calls?limit=50&offset=0&outcome=http_error
```

```
curl 'https://api.everycallhandled.com/api/v1/apps/{appId}/external-calls' \
  -H 'Authorization: Bearer YOUR_TOKEN'
```

Requires `view_call_logs`.

Query params:

- `limit` (default 50, max 200)
- `offset` (default 0)
- `outcome` — optional exact-match filter, one of `ok`, `http_error`, `network_error`, `timeout`

Rows are returned newest-first (descending `timestamp`).

Response (200):

```
{
  "calls": [
    {
      "id": "0b6f2f0e-4c1d-4a52-9a3f-9a1b2c3d4e5f",
      "timestamp": "2026-07-11T10:32:04.512Z",
      "appId": "app-abc123",
      "conversationId": "conv_1783798947228_x1y2z3",
      "source": "tool:check_availability",
      "url": "https://api.partner.example/availability",
      "method": "POST",
      "status": 200,
      "latencyMs": 431,
      "outcome": "ok"
    }
  ],
  "total": 248,
  "limit": 50,
  "offset": 0
}
```

Row fields:

Field	Notes
`timestamp`	ISO-8601, when the call was made
`conversationId`	The conversation/call the request belonged to (may be empty for flow-level calls)

Field	Notes
`source`	What made the call — see prefixes below
`url`	The target URL, **query string and fragment stripped** (query strings routinely carry API keys — secrets are never stored)
`method`	Uppercased HTTP method
`status`	HTTP status code; `0` means no response was received
`latencyMs`	Round-trip time in milliseconds
`outcome`	`ok` (2xx/3xx) · `http_error` (4xx/5xx) · `network_error` · `timeout`

What gets logged. The `source` field is prefixed by the kind of caller:

Source prefix	What it is
`tool:<toolName>`	A webhook tool the assistant invoked mid-conversation
`pms:semble`	A call to the Semble integration (record lookup, booking)
`booking:nylas`	A calendar booking call
`post-call:custom-destination`	A post-call callback POST to a destination you configured
`flow:onComplete`	A conversation-flow `onComplete` HTTP step
`workflow:httpRequest`	An automation HTTP-request step, for chat-triggered automation runs

Reliability & retention:

- **Fire-and-forget.** Log writes never block or fail a conversation. If a write is slow or errors, the call path proceeds unaffected — a missing log row never means the underlying call failed.
- **Query-stripped URLs.** Only the path portion of each URL is stored (truncated at 500 chars); anything after `?` or `#` is discarded before write, so credentials passed as query params are not retained.
- **30-day retention.** Rows are permanently deleted 30 days after their timestamp. There is no grace window — past 30 days the row is gone.

Configuration History (Versions)

Every configuration change to an app is recorded, so you can always answer "what changed, and when?" — and undo it.

`GET /api/v1/apps/{appId}/versions` `GET /api/v1/apps/{appId}/versions/{auditTimestamp}`

```
curl 'https://api.everycallhandled.com/api/v1/apps/{appId}/versions' \
  -H 'Authorization: Bearer YOUR_TOKEN'
```

```
curl 'https://api.everycallhandled.com/api/v1/apps/{appId}/versions/{auditTimestamp}' \
  -H 'Authorization: Bearer YOUR_TOKEN'
```

Available to any signed-in member of the account that owns the app.

List response (200):

```
{
  "appId": "...",
  "versions": [
    {
      "auditId": "...",
      "auditTimestamp": "2026-05-10T11:30:00.000Z",
      "userId": "...",
      "operation": "UPDATE",
      "changedConfigAttrs": ["voiceChatPrompt", "voice"]
    }
  ],
  "nextCursor": "..."
}
```

The single-version form returns the full attribute set as it was at that point.

Restore a Version

`POST /api/v1/apps/{appId}/versions/{auditTimestamp}/restore`

```
curl -X POST
'https://api.everycallhandled.com/api/v1/apps/{appId}/versions/{auditTimestamp}/restore' \
  -H 'Authorization: Bearer YOUR_TOKEN'
```

Requires `edit_settings`. Restores the app's configuration to that snapshot. Like any configuration change, the restore is a draft until you take the assistant live.

09

Billing & Credits

Overview

Billing on the platform comes down to one model. Hold onto it and every endpoint in this chapter is just a window onto it:

> **Your account has a single pool of credits. 1 credit = 1 second of assistant time. Your plan adds credits to the pool every month. Everything your assistants do — calls, chats, automations — draws from that pool. You can optionally ring-fence part of the pool for one app (a cap), so that app can never spend more than you've set aside. Auto top-up keeps the pool from running dry. The ledger shows every movement, in and out.**

That's the whole system. In detail:

- **The pool.** One balance per account, shared by all your apps. Buy credits or receive your monthly plan allowance and they land here; usage drains from here.
- **Caps (optional).** Give one app its own ring-fenced allowance, moved out of the pool. A capped app spends only its cap — useful for bounding spend per app or per customer. Uncapped apps draw straight from the pool.
- **Auto top-up (optional).** When a balance falls below your threshold, the platform charges your saved card automatically, within limits you set.
- **The ledger.** Every debit and credit — each call, each top-up, each cap movement — is one row, queryable per app and per account.

A small number of apps are billed individually rather than from a pool (`billingMode: "app"`); for those, read "the app's own balance" wherever this chapter says "the pool".

Your Balance at a Glance

```
GET /api/v1/billing/merchant
```

```
curl 'https://api.everycallhandled.com/api/v1/billing/merchant' \
  -H 'Authorization: Bearer YOUR_TOKEN'
```

Requires `view_billing`. Returns your account's pool snapshot (balance, ownerEmail, billingMode) plus a per-app summary. Apps are scoped to your account — you never see another account's apps here.

```
GET /api/v1/billing/bots
```

```
curl 'https://api.everycallhandled.com/api/v1/billing/bots' \
  -H 'Authorization: Bearer YOUR_TOKEN'
```

Requires `view_billing`. Returns every visible app with its billing snapshot (balance, tier, cap state, ownerEmail) — the data behind the dashboard's apps grid.

App Ledger

`GET /api/v1/apps/{appId}/ledger?month=2026-05&limit=50`

`GET /api/v1/apps/{appId}/ledger/summary?month=2026-05`

```
curl 'https://api.everycallhandled.com/api/v1/apps/{appId}/ledger' \
  -H 'Authorization: Bearer YOUR_TOKEN'


curl 'https://api.everycallhandled.com/api/v1/apps/{appId}/ledger/summary' \
  -H 'Authorization: Bearer YOUR_TOKEN'
```

Requires `view_billing`.

Query params: `month=YYYY-MM` OR `from` / `to`, `category`, `limit` (default 50, max 200)

Response (200):

```
{
  "items": [
    {
      "id": "...",
      "occurredAt": "2026-05-10T11:32:30Z",
      "category": "voice-phone",
      "direction": "debit",
      "amountCredits": 145,
      "balanceAfter": 13157,
      "note": "Call from +44...",
      "ref": "call-_+447941190142_abc123",
      "billingSource": "merchant"
    }
  ],
  "nextCursor": "..."
}
```

The summary form returns `{ "totals": { byCategory, byDirection }, "balanceStart", "balanceEnd", "rowCount" }`.

Account Ledger

`GET` /api/v1/merchants/{merchantId}/ledger

`GET` /api/v1/merchants/{merchantId}/ledger/summary

```
curl 'https://api.everycallhandled.com/api/v1/merchants/{merchantId}/ledger' \
  -H 'Authorization: Bearer YOUR_TOKEN'
```

```
curl 'https://api.everycallhandled.com/api/v1/merchants/{merchantId}/ledger/summary' \
  -H 'Authorization: Bearer YOUR_TOKEN'
```

Requires `view_billing`. Same shape as the app variants, aggregated across all apps under an account. The caller's account must match or be a parent of the queried account. Use the summary form for dashboards showing customer-by-customer usage; use the full form for line-by-line investigation.

Buy Credits (Checkout Link)

`POST` /api/v1/billing/topup-session

```
curl -X POST 'https://api.everycallhandled.com/api/v1/billing/topup-session' \
  -H 'Authorization: Bearer YOUR_TOKEN'
```

Requires `edit_billing`. Creates a Stripe Checkout link for a 100-minute credit top-up landing in your account pool, priced at your plan's tier rate. The price and pool target are resolved server-side from your account — the request body is empty. Nothing is charged until the checkout is completed at the link.

Response:

```
{
  "checkoutUrl": "https://checkout.stripe.com/c/pay/cs_...",
  "sessionId": "cs_...",
  "amountPence": 6500,
  "minutes": 100,
  "tier": "pro"
}
```

See Available Plans

Your plan sets your monthly credit allowance and your per-minute rate. Plan management is fully self-serve.

```
GET /api/v1/dashboard/billing/plans
```

```
curl 'https://api.everycallhandled.com/api/v1/dashboard/billing/plans' \
  -H 'Authorization: Bearer YOUR_TOKEN'
```

Requires `view_billing`.

Response (200):

```
{
  "eligible": true,
  "currency": "gbp",
  "legacyPlan": false,
  "current": { "priceId": "price_...", "tier": "pro", "interval": "month", "label": "Pro
(monthly)", "amountPence": 19900 },
  "cancelAtPeriodEnd": false,
  "cancelAt": null,
  "currentPeriodEnd": 1753968000,
  "plans": [
    { "priceId": "price_...", "tier": "starter", "isCurrent": false, "direction": "downgrade"
},
    { "priceId": "price_...", "tier": "pro", "isCurrent": true, "direction": "same" }
  ]
}
```

Accounts without a self-serve subscription get `{ "eligible": false, "plans": [], "current": null }` — plan changes for those accounts go through support. Accounts on a **legacy (grandfathered) price** get `legacyPlan: true` and an empty `plans` array: switching would irreversibly forfeit the legacy deal, so no targets are offered.

Change, Cancel or Resume Your Plan

`POST` /api/v1/dashboard/billing/change-plan `POST` /api/v1/dashboard/billing/cancel

`POST` /api/v1/dashboard/billing/resume

```
curl -X POST 'https://api.everycallhandled.com/api/v1/dashboard/billing/change-plan' \
  -H 'Authorization: Bearer YOUR_TOKEN' \
  -H 'Content-Type: application/json' \
  -d '{"tier":"pro"}'

curl -X POST 'https://api.everycallhandled.com/api/v1/dashboard/billing/cancel' \
  -H 'Authorization: Bearer YOUR_TOKEN'

curl -X POST 'https://api.everycallhandled.com/api/v1/dashboard/billing/resume' \
  -H 'Authorization: Bearer YOUR_TOKEN'
```

Requires `edit_billing` (the one grantable "manage billing" permission covers all three).

Change plan — body `{ "tier": "pro" }` (or `"priceId"`, optionally `"interval": "month"`). The new price is applied with proration; the platform then rewrites the account tier and grants any upgrade credits shortly after — the response confirms the switch was submitted, not that credits have landed yet.

Response (200): `{ "changed": true, "targetPriceId": "price_...", "proration_behavior": "create_prorations", "direction": "upgrade" }` — or `{ "changed": false, "reason": "already-on-plan" }`.

Errors: `422` no active subscription; `400` unknown/non-switchable plan; `409 legacy-plan` when the current price is grandfathered — resend with `"confirmForfeitLegacy": true` only after the account holder explicitly accepts losing the legacy deal (it cannot be restored).

Cancel — end-of-period cancel with an empty body: renewals stop but everything keeps working until the period ends, and prepaid credits are **kept** (no clawback).

Response (200): `{ "scheduledCancel": true, "cancelAt": 1753968000, "currentPeriodEnd": 1753968000 }`

Resume — undoes a scheduled cancel any time before the period end. Empty body.

Response (200): `{ "resumed": true, "currentPeriodEnd": 1753968000 }`

Ring-Fencing an App: Caps

For apps funded by the pool (`billingMode="merchant"`), you can ring-fence part of the pool as a **per-app cap**. A capped app spends only its cap; the pool only moves when you allocate more credits or when the monthly refill runs. Use caps to bound spend per app — or, as a partner, per customer.

How cap state reads:

- `GET /api/v1/apps/{appId}` returns the cap attributes inline: `merchantCapCredits` (current cap balance), `merchantCapAllocated` (lifetime allocated), `merchantCapMonthlyAllowance` , `merchantCapCarryForward` , `merchantCapLastReset` .
- If `merchantCapAllocated > 0` , the app is capped — usage drains the cap, not the pool.
- `merchantCapMonthlyAllowance > 0` turns on automatic monthly refill (cycle anchored to first allocation).

Monthly refill semantics. The refill runs on each app's monthly anchor date. Important — **expired credits evaporate; they do not return to the pool**:

```
Let A = merchantCapMonthlyAllowance
    B = current merchantCapCredits (cap balance)
    C = current merchantCapCarryForward (last cycle's leftover)
    P = current account pool balance

On refill:
    expiring = min(C, B)        // last cycle's leftover, still unspent → evaporates
    newCap   = (B - expiring) + A
    newCarry = B - expiring     // this cycle's contribution becomes next cycle's carry-forward
    poolDelta = -A              // pool ALWAYS funds A; expired credits VANISH (not returned)

If P < A:
    block + alert; no write
```

Behavioural notes:

- **Unused allowance carries forward exactly one month.** Anything still unused a month later evaporates. There is no second grace period.
- **The pool funds the full allowance on every refill** — including when expired credits would have "covered" the cost. If you build reconciliation logic that assumes expired credits return to the pool, your numbers will diverge from ours.
- **Pool-insufficient blocks the entire refill.** If the pool can't cover the allowance, the cap stays as-is for that cycle (no partial refill) and the account is alerted.
- **Top-up bundles bought separately** stay on the pool until either consumed by direct usage or pulled into a per-app cap by the next refill.

Cap Top-Up (allocate from pool to cap)

`POST /api/v1/apps/{appId}/cap-topup`

Requires `edit_billing`.

```
curl -X POST https://api.everycallhandled.com/api/v1/apps/$APP_ID/cap-topup \
  -H 'Authorization: Bearer YOUR_TOKEN' \
  -H 'Content-Type: application/json' \
  -d '{
    "credits": 6000,
    "idempotencyKey": "'$(uuidgen)'",
    "setLastReset": false
  }'
```

Body:

- `credits` (int, required) — credits to move from pool to app cap (1 credit = 1 second).
- `idempotencyKey` (string, recommended) — UUID to safely retry. Without it, an accidental double-click double-allocates.
- `setLastReset` (bool, default false) — when toggling `merchantCapMonthlyAllowance` on for the first time, set true so the scheduler doesn't immediately refill again.

Response (200):

```
{
  "success": true,
  "appId": "...",
  "creditsAllocated": 6000,
  "capBalance": 9600,
  "capAllocatedTotal": 24000,
  "poolBalance": 53000
}
```

Errors:

- `400 INSUFFICIENT_POOL_BALANCE` — the pool can't cover the request.

Remove a Cap (return credits to the pool)

The inverse of cap top-up: returns the app's remaining cap balance to the pool and zeroes every cap attribute (balance, lifetime allocated, monthly allowance, carry-forward, last reset), so the app goes back to **uncapped** and draws directly from the pool again.

`POST /api/v1/apps/{appId}/cap-remove`

Requires `edit_billing`. Only valid for apps with `billingMode=merchant`.

```
curl -X POST https://api.everycallhandled.com/api/v1/apps/$APP_ID/cap-remove \
  -H 'Authorization: Bearer YOUR_TOKEN' \
  -H 'Content-Type: application/json' \
  -d '{ "idempotencyKey": "'$(uuidgen)'" }'
```

Body:

- `idempotencyKey` (string, recommended) — UUID to safely retry. A replayed request returns the same result with `idempotent: true` instead of moving credits twice.

Response (200):

```
{
  "removed": true,
  "appId": "...",
  "returnedToPool": 3600,
  "poolBalance": 56600,
  "merchantId": "acme"
}
```

If the app is already uncapped this is an honest no-op: `{ "removed": false, "reason": "..." }` with a 200.

Notes:

- The transfer is ordered pool-first (credit the pool, then zero the cap), so a mid-flight failure can only ever leave extra credits, never lost ones, and is rolled back automatically.
- Two ledger rows record the move (`cap-remove-debit` on the app, `cap-remove-credit` on the pool) — you'll see them in the ledger endpoints above.
- If the app had a per-app auto top-up configured while capped, removing the cap means future auto top-up reads follow the pool again (see below).

Auto Top-Up (never run out)

Auto top-up charges the saved card automatically when a credit balance falls below a threshold, so a busy line never goes dark mid-month. Nothing is charged when you call these endpoints — they only read and write the configuration; the platform performs the actual off-session charge when the threshold is crossed.

There are two scopes, matching the pooled billing model:

- **Account (pool) level** — the natural home for most customers: one setting that keeps the shared pool topped up, funding every uncapped app.
- **App level** — for individually billed apps and apps with a per-app cap, where the app has its own balance to protect.

Both scopes share the same configuration shape:

```
{
  "enabled": true,
  "thresholdSeconds": 600,
  "amountPence": 2000,
  "monthlyCapPence": 10000
}
```

- `enabled` (bool) — master switch.
- `thresholdSeconds` (int) — top up when the balance drops below this many credit-seconds. Maximum 36,000 (600 minutes) for customer saves.
- `amountPence` (int) — how much to charge per top-up: minimum 500 (£5), maximum 100,000 (£1,000).
- `monthlyCapPence` (int) — spending ceiling per calendar month; `0` means no cap. If set, it must be at least one top-up amount.

Reads also return `hasSavedCard` (whether a card is on file from a plan purchase) and `source` (`"merchant"` when the config lives on the pool, `"app"` when it lives on the app).

Safety rails on execution, regardless of configuration: at least 1 hour between charges, at most 3 charges per day, and the monthly cap is always honoured. Credits are purchased at your plan's per-minute rate (the same rate as a manual top-up).

Read Auto Top-Up Settings

`GET` /api/v1/accounts/{merchantId}/auto-topup `GET` /api/v1/apps/{appId}/auto-topup

```
curl 'https://api.everycallhandled.com/api/v1/accounts/{merchantId}/auto-topup' \
  -H 'Authorization: Bearer YOUR_TOKEN'
```

```
curl 'https://api.everycallhandled.com/api/v1/apps/{appId}/auto-topup' \
  -H 'Authorization: Bearer YOUR_TOKEN'
```

Requires `view_billing`. You can read your own account's pool setting; a parent account can read a child's.

Change Auto Top-Up Settings

`PATCH` /api/v1/accounts/{merchantId}/auto-topup `PATCH` /api/v1/apps/{appId}/auto-topup

```
curl -X PATCH 'https://api.everycallhandled.com/api/v1/apps/{appId}/auto-topup' \
  -H 'Authorization: Bearer YOUR_TOKEN' \
  -H 'Content-Type: application/json' \
  -d '{"enabled":true,"thresholdSeconds":1200,"amountPence":2000,"monthlyCapPence":10000}'
```

Requires `edit_billing`.

```
curl -X PATCH https://api.everycallhandled.com/api/v1/accounts/$ACCOUNT_ID/auto-topup \
  -H 'Authorization: Bearer YOUR_TOKEN' \
  -H 'Content-Type: application/json' \
  -d '{ "enabled": true, "thresholdSeconds": 1200, "amountPence": 2000, "monthlyCapPence":
10000 }'
```

The app-level endpoint is **scope-aware**: it follows the app's billing source. For an individually billed app or a capped app, the config protects the app's own balance. For an uncapped pool-funded app, reads and writes fall through to the shared pool config — the same setting the account-level endpoint manages — and the response says so with `"source": "merchant"`. If two uncapped apps both "have" auto top-up configured, it's one pool setting seen from two places, not two settings.

Errors:

- `409` — enabling without a saved card. Purchase a plan first (that saves the card), then enable.
- `403` — the account is outside your scope.

10

Your Team — Roles & Permissions

Overview

You won't be the only person running your assistant. Receptionists need to read conversations, managers need to change settings, bookkeepers need billing visibility — and nobody should have more access than their job needs. The platform handles this with four roles plus per-user permission overrides.

The four roles

Every team member (and every API token) has a **role** that grants a baseline set of permissions, tunable per user with overrides.

Role	Use for	Baseline
`owner`	Account holders, senior people	**All** permissions. Cannot be revoked. At least one owner per account required.
`manager`	Day-to-day operators, partner team members	Settings, knowledge, apps (create/duplicate/take live), credentials (view + edit), conversations, recordings, caller profiles. **No** user management, billing, branding, or profile deletion (all owner-only).
`staff`	Receptionists, front-of-house	Conversations, recordings, read-only settings and credentials, caller profiles. No editing of settings/knowledge/credentials, no billing.
`viewer`	Bookkeepers, auditors, oversight	Read-only conversation visibility.

Permission catalogue

Permission	What it gates	Default roles
`view_settings`	Listing apps via the API; settings visibility in the dashboard	owner, manager, staff
`edit_settings`	Update/delete an app, restore versions, web-chat config, voice and phone/calling configuration	owner, manager
`create_app`	Create apps	owner, manager
`publish`	Take assistants live and offline	owner, manager
`clone_app`	Duplicate an app	owner, manager
`view_billing`	All ledger and billing reads, plan listing, auto top-up reads (account-level too)	owner
`edit_billing`	Buy credits, cap top-up/remove, plan change/cancel/resume, auto top-up writes	owner
`view_credentials`	List credentials (app + account; never returns secret values), run credential tests	owner, manager, staff
`edit_credentials`	Add, replace or delete credentials (app + account)	owner, manager
`view_conversations`	Conversation visibility in the dashboard	owner, manager, staff, viewer
`edit_conversations`	Transcript edits in the dashboard	owner, manager, staff
`view_recordings`	Recording playback in the dashboard	owner, manager, staff
`download_recordings`	Recording download in the dashboard	owner, manager, staff
`manage_users`	Invite, remove, change roles and overrides; create and list customer accounts	owner
`view_caller_profiles`	List/read caller profiles, issues, analytics, CSV exports	owner, manager, staff
`edit_caller_profiles`	Update profiles, merge duplicates, update issues	owner, manager
`delete_caller_profile`	Right-to-be-forgotten deletion	owner
`enable_reviews`	Per-app review settings	owner
`manage_branding`	Logo, colours, business name; per-account branding	owner
`set_wholesale_rate`	Per-customer rate when a partner creates customer accounts	owner (partner only)
`partner_principal`	Marker — this user is the partner-side principal; bundled by the become-a-	(set by become-a-partner)

Permission	What it gates	Default roles
	partner flow; used for routing + display logic	
`view_knowledge`	Knowledge reads	owner, manager
`edit_knowledge`	Knowledge writes (upload / delete content)	owner, manager
`manage_bookings`	Bookings tab, calendar connect, upcoming-bookings list	owner, manager
`manage_booking_config`	Semble booking config	*none* — opt-in `+manage_booking_config` only
`view_intakes`	Intake submissions (list + detail)	owner, manager
`view_workflow_runs`	Automation run history in the dashboard	owner, manager
`view_call_logs`	Third-party call log	owner, manager
`manage_workflows`	Automation authoring in the dashboard	owner
`manage_api_keys`	Mint / list / revoke scoped API keys	owner
`show_partner_cta`	"Become a partner" visibility + endpoint	*none* — opt-in `+show_partner_cta` only

A note on the conversation permissions: conversation, recording and import **API endpoints** are available to any signed-in member of the owning account; the `view_conversations` / `edit_conversations` / recording permissions govern what each member sees and can do **in the dashboard**.

Scope: permissions take effect within the token's account. A `manager` in one account can edit that account's apps (and its sub-accounts') but never another account's. See For Partners for how account hierarchy works.

Override syntax: when a user is invited or edited, overrides go in as an array — e.g. `featureOverrides: ['+view_billing', '-edit_conversations']`. `+` grants on top of the role's baseline; `-` revokes from it. Precedence is `-` over `+` over baseline.

Manage Team Members

`GET` /api/v1/users?email=… `POST` /api/v1/users `PATCH` /api/v1/users/{userId}

`POST` /api/v1/users/{userId}/pause `DELETE` /api/v1/users/{userId}

```
curl 'https://api.everycallhandled.com/api/v1/users' \
  -H 'Authorization: Bearer YOUR_TOKEN'
```

```
curl -X POST 'https://api.everycallhandled.com/api/v1/users' \
  -H 'Authorization: Bearer YOUR_TOKEN' \
  -H 'Content-Type: application/json' \
  -d '{"email":"teammate@business.com","role":"staff","sendInvite":true}'
```

```
curl -X PATCH 'https://api.everycallhandled.com/api/v1/users/{userId}' \
  -H 'Authorization: Bearer YOUR_TOKEN' \
  -H 'Content-Type: application/json' \
  -d '{"role":"manager"}'
```

```
curl -X POST 'https://api.everycallhandled.com/api/v1/users/{userId}/pause' \
  -H 'Authorization: Bearer YOUR_TOKEN' \
  -H 'Content-Type: application/json' \
  -d '{"paused":true}'
```

```
curl -X DELETE 'https://api.everycallhandled.com/api/v1/users/{userId}' \
  -H 'Authorization: Bearer YOUR_TOKEN'
```

Requires `manage_users`. All endpoints are account-scoped: you only see and affect users inside your own account (and its sub-accounts).

Invite creates the sign-in and the user record and emails the invitation. Role is one of `owner | manager | staff | viewer`; you cannot assign a role that outranks your own, nor grant permissions you don't hold.

Role changes are held to a two-way ceiling: the new role must not outrank yours, and you cannot act on a user whose current role outranks yours (so a manager can never demote an owner). Clearing a role requires owner rank (a missing role defaults to owner). The last owner of an account cannot be demoted.

Pause is a reversible suspend: `{ "paused": true }` blocks the sign-in immediately (existing sessions lapse within the hour), `{ "paused": false }` restores it; nothing is deleted. Email lookups (`?email=`) include the current `paused` state. Pausing is held to the same guards as removal.

Remove deletes the sign-in and the user record — access ends immediately. Guards: you cannot remove yourself; you cannot remove a user whose role outranks yours; the last owner

of an account cannot be removed (assign another owner first). `userId` accepts the id returned by the list endpoint.

Changing permission overrides (the `featureOverrides` array) goes through the same `PATCH /api/v1/users/{userId}` — see the override syntax above. Setting a `wholesaleRate` on a user additionally needs `set_wholesale_rate`.

11

API Keys

Overview

Support-issued tokens are broad. When you want a key for one job — a Zapier zap that posts into chat, a script that syncs knowledge, a form-collection integration — mint a **capability-scoped key** yourself: long-lived, pinned to one app, limited to exactly the capabilities you grant, revocable at any time. No dashboard sign-in needed for the integration that uses it.

> **The `/data/` prefix.** Everything under `/api/v1/apps/{appId}/data/…` is the app's **Data API** — the surface for records an app captures or owns (today: `data/api-keys` for scoped keys, with form records following the same pattern). New captured-record kinds will appear under this prefix as they ship; sub-routes carry their own permission gates.

Mint a Key

`POST /api/v1/apps/{appId}/data/api-keys`

Requires `manage_api_keys` (owner by default). Keys can also be minted from the dashboard's Credentials tab.

```
curl -X POST https://api.everycallhandled.com/api/v1/apps/$APP_ID/data/api-keys \
  -H 'Authorization: Bearer YOUR_TOKEN' \
  -H 'Content-Type: application/json' \
  -d '{ "label": "Zapier integration", "capabilities": ["chat", "knowledge"] }'
```

Returns the raw token **once** — it is never stored or shown again; store it like a password.

Capabilities: `chat` (POST `/apps/{appId}/chat` on the key's own app), `knowledge` (knowledge-base routes), `forms` (the app's form-record routes under `/apps/{appId}/data/`). Anything outside the granted classes returns `403 API_KEY_SCOPE`.

Use a Key

```
Authorization: Bearer <your-api-key>
```

Exactly like a session token. A key is pinned to the app it was minted on and acts with a `viewer` role plus its capability-derived permissions — it can never widen itself. Usage joins the same credit ledger as every other channel.

List and Revoke Keys

`GET` /api/v1/apps/{appId}/data/api-keys `DELETE` /api/v1/apps/{appId}/data/api-keys/{id}

```
curl 'https://api.everycallhandled.com/api/v1/apps/{appId}/data/api-keys' \
  -H 'Authorization: Bearer YOUR_TOKEN'
```

```
curl -X DELETE 'https://api.everycallhandled.com/api/v1/apps/{appId}/data/api-keys/{id}' \
  -H 'Authorization: Bearer YOUR_TOKEN'
```

Requires `manage_api_keys`. Listing returns metadata only, never tokens. Revoking kills the key at the next auth lookup (subsequent requests get 401).

12

Credentials — Connect Your Own Accounts

Overview

Some of the best things an assistant can do require *your* accounts: booking into your Semble diary, sending calls through your own Twilio account, using your own OpenAI key. Credentials are how you hand those over safely — values are stored encrypted in the platform's secret store, never returned by any read endpoint, and only used server-side when an integration needs them.

Credentials live at two levels:

- **Account-level (recommended)** — set once, used by every app in the account. This is the right home for partner-grade setups: your Twilio account, payment keys, model-provider keys you don't want to repeat per customer.
- **App-level** — for special cases where one app needs its own credential (a different Semble diary, a one-off key).

Resolution: when a call or message needs a credential, the platform resolves account-level credential first, then falls back to the platform default. Apps don't carry credentials in this model; the account does — use app-level only when an app genuinely differs.

Credential Catalog

`GET /api/v1/credentials/catalog`

```
curl 'https://api.everycallhandled.com/api/v1/credentials/catalog' \
  -H 'Authorization: Bearer YOUR_TOKEN'
```

Available to any signed-in caller. Returns the static catalog of supported credential types (e.g. `semble_api_key`, `twilio_byot`, `openai_api_key`) with the body schema each type expects (Twilio = `accountSid` + `authToken`; OpenAI = `apiKey`; etc.).

List Account Credentials

`GET /api/v1/merchants/{merchantId}/credentials`

```
curl 'https://api.everycallhandled.com/api/v1/merchants/{merchantId}/credentials' \
  -H 'Authorization: Bearer YOUR_TOKEN'
```

Requires `view_credentials` (and the account must be yours or beneath yours). Returns metadata only — never the secret value.

Add, Replace or Delete an Account Credential

`PUT /api/v1/merchants/{merchantId}/credentials/{name}`

`DELETE /api/v1/merchants/{merchantId}/credentials/{name}`

```
curl -X PUT 'https://api.everycallhandled.com/api/v1/merchants/{merchantId}/credentials/{name}'
\
  -H 'Authorization: Bearer YOUR_TOKEN' \
  -H 'Content-Type: application/json' \
  -d '{"value":"sk_live_..."}'
```

```
curl -X DELETE
'https://api.everycallhandled.com/api/v1/merchants/{merchantId}/credentials/{name}' \
  -H 'Authorization: Bearer YOUR_TOKEN'
```

Requires `edit_credentials`. The PUT body shape depends on the credential type — see the catalog above. DELETE is a hard delete of the stored secret; it cannot be undone, and voice/SMS resolution falls back to platform credentials immediately.

List App Credentials

`GET /api/v1/apps/{appId}/credentials`

```
curl 'https://api.everycallhandled.com/api/v1/apps/{appId}/credentials' \
  -H 'Authorization: Bearer YOUR_TOKEN'
```

Requires `view_credentials`. Metadata only — never the secret value.

Add or Update an App Credential

`POST` /api/v1/apps/{appId}/credentials `PUT` /api/v1/apps/{appId}/credentials/{name}

```
curl -X PUT 'https://api.everycallhandled.com/api/v1/apps/{appId}/credentials/{name}' \
  -H 'Authorization: Bearer YOUR_TOKEN' \
  -H 'Content-Type: application/json' \
  -d '{"value":"sk_live_..."}'
```

Requires `edit_credentials`. POST creates (or replaces by name); PUT updates an existing credential's value.

```
curl -X POST https://api.everycallhandled.com/api/v1/apps/$APP_ID/credentials \
  -H 'Authorization: Bearer YOUR_TOKEN' \
  -H 'Content-Type: application/json' \
  -d '{ "name": "semble-main", "type": "semble_api_key", "value": "sk_live_..." }'
```

The value goes into the encrypted secret store; only the credential's name and type are kept on the app.

Test a Credential

`POST` /api/v1/apps/{appId}/credentials/{name}/test

```
curl -X POST 'https://api.everycallhandled.com/api/v1/apps/{appId}/credentials/{name}/test' \
  -H 'Authorization: Bearer YOUR_TOKEN'
```

Requires `view_credentials`. Pings the upstream API with the stored credential — the quickest way to confirm a key works before a caller finds out it doesn't.

Response (200): `{ "ok": true|false, "detail": "..." }`

Delete an App Credential

`DELETE` /api/v1/apps/{appId}/credentials/{name}

```
curl -X DELETE 'https://api.everycallhandled.com/api/v1/apps/{appId}/credentials/{name}' \
  -H 'Authorization: Bearer YOUR_TOKEN'
```

Requires `edit_credentials`. Hard delete; cannot be undone.

13

Automations

Overview

Beyond conversations, the platform can run **automations** (workflows) — multi-step processes like "fetch listings, score them, email the matches" that run on a schedule or on demand. Automation definitions are platform-authored today (there is no self-serve definition-authoring endpoint); this chapter exists so the shapes you see in run output make sense, and so operator tooling is documented.

Run history for your apps' automations is visible in the dashboard (permission `view_workflow_runs`).

Workflow tasks — the `compute` task

A definition's schedule is an ordered list of named steps, each with a `task` type. Established task types include `httpRequest`, `sendEmail`, `sendSMS`, `workflow_bedrock` (LLM step), data-shaping tasks (`mapData`, `filterData`, `aggregateData`, …) and control tasks (`conditionalBranch`, `parallel`, `iterateArray`, `wait`, `invokeWorkflow`).

`compute` runs a **curated deterministic operation** from a platform-owned registry — pure functions with no ambient authority, used where a pipeline needs exact, auditable logic rather than an LLM step (parsing, geometric evaluation, dedupe, delivery routing). Step shape:

```
{ "name": "geo", "task": "compute",
  "input": { "op": "geoEvaluate", "args": { "listings": "{{$.parse.taskResult.listings}}" } } }
```

`args` values support `{{workflowInput.*}}` and `{{$.step.taskResult.*}}` references, resolved against the run's state. Unknown `op` names fail the step loudly with the known-op list. Compute steps bill at the standard integration step rate (default 5 credits per step; automation step rates are operator-adjustable, so treat the numbers quoted here as defaults rather than fixed prices).

Taught compute steps (approved calculations)

One registry op, `taughtStep`, runs an **operator-taught calculation** stored on an app — small pure functions (for example a custom result-scoring rule) taught in plain language through the platform's teaching assistant, which drafts the code.

The trust model is deliberate:

- A taught step always lands as a **draft** and a draft **never executes** — pipelines treat it as absent.
- Approval is an explicit act: in the dashboard (Settings → Skills → Taught calculations, where the exact code is displayed for review) or via the teaching assistant's confirm-gated approve step. Re-teaching an approved step demotes it back to draft.
- Code is validated against a strict deny-list (no imports, no process/network/filesystem access, no async, no prototype access) **both when drafted and again at execution** — a step that fails validation will not run regardless of how it entered the configuration.
- Execution is fully isolated: a locked-down context exposing only `Math` and `JSON`, with a copied input, a hard 1-second timeout and a bounded output size.
- In a pipeline, `taughtStep` **fails open**: if the referenced step is missing, unapproved or errors, the step's input passes through untouched and the run continues. A broken taught calculation degrades a pipeline; it never breaks one.

Taught steps live in the app's configuration (visible under `voiceChatTools.computeSteps` in `GET /apps/{appId}` responses) with `name`, `description`, `code`, `status` (`draft` / `approved`) and approval timestamps.

14

For Partners

Overview

Everything in this chapter is **partner territory**: reselling the platform to your own customers under your own brand, with your own pricing, your own payment rails and a customer tree you manage. If you run one business with its own assistants, you can skip it entirely.

As a partner you get: **customer accounts** (sub-accounts you create and manage), **wholesale rates** (your per-customer pricing), **payment routing** (your customers pay you directly via Stripe Connect), and **branding** (your logo and colours on customer-facing surfaces, via the `manage_branding` permission in the dashboard).

> **Naming transition — `merchantId` → `accountId`.** We are renaming "merchant" to **"account"** across the product. The API is fully backward-compatible during the transition:
>
> - **Requests:** send either `accountId` (preferred) or `merchantId` (legacy) — they are interchangeable. If you send both, `merchantId` wins.
> - **Responses:** requests may send either field (input aliasing is applied everywhere), and some billing responses already echo an `accountId` alongside `merchantId`. Full response-side aliasing is still rolling out, so for now read `merchantId` and treat `accountId` as an additive alias where present. `merchantId` is **not** being removed yet, so existing integrations keep working unchanged.
> - **Routes and the hierarchy semantics below are unchanged** (`/api/v1/merchants/...`). `accountId` is the same colon-separated value, just a new name for the field.
>
> **Action:** new integrations should read/write `accountId`. `merchantId` is deprecated and will be retired in a future major version (we'll give notice first).

Every app belongs to an account; pool balances, account-level credentials and Stripe Connect linkage all live on the account; access tokens carry account scope to gate cross-customer visibility.

Account IDs and hierarchy

Account IDs are colon-separated strings forming a hierarchy:

```
acme                        ← your partner account
acme:harbour-hotel          ← a customer account you created
acme:harbour-hotel:bath     ← optional deeper nesting
```

The **parent** is everything before the last colon. An account with id `acme:harbour-hotel` has parent `acme`.

Scope inheritance: a token scoped to `acme` can see `acme` and everything beneath (`acme:*`). It cannot see siblings or ancestors. This is enforced server-side on every endpoint; you don't have to filter — the platform does.

Become a Partner (self-serve)

`POST` `/api/v1/dashboard/become-partner`

Requires `show_partner_cta` (an opt-in permission — the same one that surfaces the "Become a partner" button in the dashboard). Idempotent. Grants the caller the partner bundle (`+manage_branding` , `+manage_users` , `+set_wholesale_rate` , `+partner_principal`) and ensures a partner account exists for them. After this fires, the caller can begin creating customer accounts.

```
curl -X POST https://api.everycallhandled.com/api/v1/dashboard/become-partner \
  -H 'Authorization: Bearer YOUR_TOKEN' \
  -H 'Content-Type: application/json' \
  -d '{
    "displayName": "Acme Answering Co"
  }'
```

No body fields required beyond `displayName` for branding purposes.

Create a Customer Account

`POST /api/v1/merchants`

Requires `manage_users` on a token scoped at or above the new account's parent.

```
curl -X POST https://api.everycallhandled.com/api/v1/merchants \
  -H 'Authorization: Bearer YOUR_TOKEN' \
  -H 'Content-Type: application/json' \
  -d '{
    "merchantId": "acme:harbour-hotel",
    "displayName": "Harbour Hotel"
  }'
```

This call creates the **account shell** only. The first owner (via `POST /api/v1/users`), the plan/tier, and any per-customer wholesale rate are set with their own calls afterwards — no financial fields are seeded from this request.

Body fields:

Field	Required	Notes
`merchantId`	yes	Colon-separated hierarchy (lowercase alphanumeric segments, hyphens allowed inside a segment). `accountId` accepted as an alias. Any missing ancestor is created automatically.
`displayName`	yes	Human-readable name shown in branding contexts.

Response (200):

```
{
  "success": true,
  "id": "...",
  "merchantId": "acme:harbour-hotel",
  "displayName": "Harbour Hotel",
  "parentMerchantId": "acme"
}
```

Errors (messages, not symbolic codes — see Error Handling):

- `400` — `merchantId required`, or `Invalid merchantId format...` (spaces, leading/trailing colons, etc.).
- `403` — `Forbidden: target merchantId is outside your scope`.
- `409` — `Merchant {id} already exists`.
- `409 ALREADY_EXISTS` — the id is taken (this is global — even across partners).

List Accounts

`GET /api/v1/merchants`

Requires `manage_users`. Returns accounts visible to your scope: your own account and all descendants.

```
curl https://api.everycallhandled.com/api/v1/merchants \
  -H 'Authorization: Bearer YOUR_TOKEN'
```

Response (200):

```
{
  "merchants": [
    {
      "merchantId": "acme",
      "displayName": "Acme Answering Co",
      "creditBalance": 125000,
      "tier": "enterprise",
      "stripeConnectStatus": "active",
      "subMerchantCount": 14
    },
    {
      "merchantId": "acme:harbour-hotel",
      "displayName": "Harbour Hotel",
      "parentMerchantId": "acme",
      "creditBalance": 3200,
      "tier": "starter",
      "stripeConnectStatus": "none",
      "subMerchantCount": 0
    }
  ]
}
```

`creditBalance` is included per row, so rendering a partner's customer tree needs no extra lookups.

Running your customers day-to-day

Everything else a partner needs is the same API, scoped per customer:

- **Team access** — invite your customer's staff with `POST /api/v1/users`, scoped to their account (Your Team).
- **Usage** — `GET /api/v1/merchants/{merchantId}/ledger` and `/ledger/summary` per customer (Billing & Credits).
- **Credentials** — account-level credentials per customer (Credentials).
- **Spend control** — per-app caps to bound each customer's spend (Billing & Credits).

Stripe Connect — your customers pay you

If your customers should pay *you* directly (not us), Stripe Connect is the topology you want: your customers see a checkout branded as your business; Stripe pays you directly; the platform takes an application fee per transaction. If you prepay us and bill your customers through your own rails, you don't need this section.

Two account types

Express *(recommended for most partners)* — we create a Stripe-managed account in your name. Stripe handles identity checks, payouts, dispute flow and customer support. You provide branding (logo, primary colour, business details) once at onboarding and Stripe uses it on every customer-facing surface (checkout, receipt emails, billing portal). You can't customise the Stripe-side checkout further — it's a managed product.

Standard — you connect an existing Stripe account you already own. You retain full control of branding, checkout customisation, dispute handling and payout schedule. Suitable for partners with significant existing Stripe usage who want continuity with their other Stripe-billed products.

Express is the default we recommend; Standard is the escape hatch for sophisticated existing Stripe users.

Onboarding

Onboarding starts from the Billing page of your dashboard. For Express, the platform creates your Stripe account and hands your browser to Stripe's hosted identity-and-payout setup; for Standard, you're sent through Stripe's connect-account authorisation for the account you already own. Either way you land back on your dashboard with a success or failure status, and your account then carries:

```
{
  "stripeConnectAccountId": "acct_1Abc...",
  "stripeConnectAccountType": "express",
  "stripeConnectStatus": "onboarding"
}
```

Read these via `GET /api/v1/billing/merchant` to know your onboarding state.

Onboarding state machine

```
none → onboarding        (you click Connect)
onboarding → active      (Stripe completes checks + payout setup)
onboarding → restricted  (Stripe needs more info; tasks waiting in your Stripe dashboard)
onboarding → rejected    (checks failed; Connect unavailable on this account)
active → restricted      (compliance event; Stripe pauses payouts)
```

`active` is the only state in which Connect-routed checkouts can be created. `restricted` blocks new charges (existing charges still settle). `rejected` is terminal for that account. Poll `GET /api/v1/billing/merchant` to read the current state, or watch for state transitions on your own Stripe webhook.

Checkout routing

Once your status is `active`, your customers' payments route through your Stripe account automatically. Credit checkouts (Buy Credits, and cap top-ups initiated for an end customer via `POST /api/v1/apps/{appId}/cap-topup`) are created on your Connect account with an application fee: Stripe charges the customer, settles to you, and transfers the fee to the platform. We never touch your customer's card details or settle their funds.

The routing decision is made server-side from the customer's account chain: the platform walks up to the nearest ancestor account with `stripeConnectStatus === 'active'`; if one is found, the checkout routes through that ancestor's Connect account; otherwise it routes to the platform. The application fee is configured per-account at onboarding (default partner rate; bespoke agreements override) and is visible in your dashboard's Billing → Connect page.

Subscriptions

When a customer of yours pays a monthly subscription through Connect, the platform detects each renewal and allocates the monthly credit bundle to that customer's pool automatically — you don't wire any invoice handling. You DO need to listen for failed-renewal events on your own side (so you can chase your customer for payment): Stripe sends `invoice.payment_failed` to your registered Connect webhook endpoint when that happens.

Disputes

When a Stripe dispute opens against a Connect-routed charge, the platform records it and surfaces it read-only in your dashboard's Billing page. There is no dispute-management API today — disputes are handled in your Stripe dashboard (Express or Standard). The platform's role is awareness and audit, not adjudication.

15

Chat API & Webhooks

Overview

Two ways to wire conversations into your own software: **talk to the assistant directly** over HTTP (drive it from your own UI, test prompts from CI, let another system converse with it), and **be told when conversations finish** via a webhook (post transcripts to your CRM, trigger follow-ups, feed analytics).

Send a Chat Message

`POST /api/v1/apps/{appId}/chat`

Available to any signed-in member of the account that owns the app, and to scoped API keys with the `chat` capability. Runs a synchronous chat exchange against the app's prompt. Three fields are required: `conversationId` (your identifier for the conversation — reuse it to continue the same conversation), `userIdentifier` (a stable identifier for the end user, e.g. their email or your user id — this is what links the conversation to a Contact record), and `message`.

```
curl -X POST https://api.everycallhandled.com/api/v1/apps/$APP_ID/chat \
  -H 'Authorization: Bearer YOUR_TOKEN' \
  -H 'Content-Type: application/json' \
  -d '{
    "conversationId": "conv-12345",
    "userIdentifier": "customer@example.com",
    "message": "Hello, can I book an appointment?"
  }'
```

Response (200):

```
{
  "conversationId": "conv-12345",
  "contactId": "...",
  "userMessage": "Hello, can I book an appointment?",
  "aiResponse": "Of course! What date and time works for you?"
}
```

Send the same `conversationId` on subsequent calls to maintain conversation context.

This endpoint waits for the complete answer before responding, and its gateway allows up to 60 seconds. For conversations that use tools (which can take 30 seconds or more per turn), we recommend the streaming variant below — you see the answer as it is generated, exactly like the dashboard preview.

Stream a Chat Message (Server-Sent Events)

Streaming chat uses a dedicated endpoint (not the `api.everycallhandled.com` host) with the SAME `Authorization` header you already use — your API token or scoped key (`chat` capability, pinned to its own app):

```
curl -N -X POST https://vixx7dopyco6rkjo2h4dqq7exq0byvxo.lambda-url.eu-west-1.on.aws/ \
  -H 'Authorization: Bearer YOUR_TOKEN' \
  -H 'Content-Type: application/json' \
  -d '{
    "appId": "'$APP_ID'",
    "targetPath": "/'$APP_ID'",
    "conversationMode": true,
    "conversationId": "conv-12345",
    "userIdentifier": "customer@example.com",
    "isStreamingFormat": true,
    "query": "{\"question\": \"Hello, can I book an appointment?\"}"
  }'
```

The response is a standard Server-Sent Events stream. Text arrives as it is generated:

```
event: message
data: Of course!

event: message
data:  What date and time

event: message
data:  works for you?
```

Notes:

- `query` is a JSON document held as a string, with the user's message under `question` .
- Reuse `conversationId` to continue a conversation; messages are persisted and the conversation appears in the dashboard exactly like one from the buffered endpoint, linked to the Contact identified by `userIdentifier` .
- Errors arrive as `event: error` frames with a JSON body (`401` invalid token, `403` wrong app or account, `429` rate limited — the streaming endpoint allows up to 30 conversation starts per minute per credential).
- Comment frames (lines starting `:`) are heartbeats during long tool runs; SSE parsers ignore them automatically.
- In JavaScript, read the response body with `fetch` and a `ReadableStream` reader (the `EventSource` API only supports GET); in Python, use `requests` with `stream=True` and iterate lines.

Post-Call Webhook

Apps can configure a `postChatCallback` attribute that fires after each completed conversation. Useful for:

- Posting transcripts to your CRM
- Triggering follow-up automation
- Custom analytics

Set the callback by `PATCH`ing the app:

```
{
  "attributes": {
    "postChatCallback": {
      "uri": "https://your-server.com/webhook",
      "method": "POST",
      "headers": { "Authorization": "Bearer YOUR_INTERNAL_TOKEN" }
    }
  }
}
```

Payload (POSTed to your URL after each call):

```
{
  "appId": "...",
  "chatId": "...",
  "callerIdPhone": "+44...",
  "callbackPhone": "+44...",
  "callerName": "...",
  "callerEmail": "...",
  "summary": "Caller booked an appointment for...",
  "urgency": "normal",
  "sentiment": 8,
  "callDuration": 145,
  "transcript": [ { "speaker": "...", "text": "...", "timestamp": "..." } ],
  "patientContext": { /* populated when the Semble integration is connected */ },
  "bookingsMade": [ /* if a booking integration is connected */ ]
}
```

Webhook Reliability & Failure Handling

Timeout: each delivery attempt times out after **5 seconds**. Your endpoint should acknowledge with a 2xx response inside that window — even a bare `200 OK` is fine; we don't need a body. Slow processing (writing to your CRM, triggering downstream automations) should happen asynchronously after you've already responded; treating webhook delivery as an inbox-write pattern is the safest shape.

Retry policy: failed deliveries (any non-2xx response, connection refused, or timeout) are retried with exponential backoff:

Attempt	Delay after previous
1	immediate (first try)
2	~30 seconds
3	~2 minutes
4	~10 minutes
5	~1 hour
6 (final)	~6 hours

After 6 failed attempts, the platform stops trying for that conversation and records the failure internally. We do not notify your account by default; if you need failure notifications, talk to support.

Idempotency: every delivery carries the same `(appId, chatId)` tuple — if you receive the same payload twice (e.g. a retry fired even though your endpoint succeeded but the connection dropped before we read the response), de-duplicate on `chatId`. Payload content never changes between retries, so a second delivery for a known `chatId` is safe to drop.

Signature verification: the platform signs each request with HMAC-SHA256 over the JSON body, included as an `X-ILQ-Signature` header. The signing key is generated when you configure the callback and returned as the `webhookSigningSecret` field in the `PATCH /apps/{id}` response that sets `postChatCallback` (it starts with `whsec_`); store it server-side. If you set the callback up before signing was available, re-save it once (any `PATCH` that sets `postChatCallback`) to generate the key — deliveries stay unsigned, exactly as before, until you do. Verify on receipt:

```
import hmac, hashlib
expected = hmac.new(signing_key.encode(), request.body, hashlib.sha256).hexdigest()
if not hmac.compare_digest(expected, request.headers.get('X-ILQ-Signature', '')):
    return 401
```

Without verification you can't tell a real platform delivery from a third party who's discovered your callback URL. We strongly recommend verifying every payload.

Failure-mode summary:

Scenario	What happens
Your endpoint returns 2xx	Delivery succeeded; no retry.
Your endpoint returns 4xx	Treated as permanent failure; retries cease after attempt 1. (Reason: 4xx usually means malformed payload from our side; retrying won't help.)
Your endpoint returns 5xx	Retried per the backoff table above.
Connection refused / DNS failure	Retried per the backoff table above.
Timeout (>5s, no response)	Retried per the backoff table above.
All 6 retries exhausted	Recorded on our side; not delivered to your account.

The call itself succeeds regardless of webhook outcome. Webhook delivery is asynchronous to the call; a failed webhook never affects the caller's experience or your billing. The transcript + summary + analytics are still stored in your account and queryable via the `/conversations` endpoints — the webhook is a convenience, not the source of truth.

Other events: the post-call webhook is the only customer-subscribable push notification today. For billing and usage events, poll `GET /api/v1/billing/merchant` or the ledger endpoints on your own schedule. A general-purpose subscribable event stream is on the roadmap but not shipped.

16

Error Handling

Overview

All errors return JSON nested under `err`: `{ "err": { "message": "<message>", "code": "<symbolic_code>" } }`. `code` is present on validation and feature-gate errors and defaults to `INTERNAL_ERROR` otherwise. HTTP status codes follow standard semantics.

Status	Meaning	Common causes
400	Bad Request	Validation failure, missing required field
401	Unauthorized	Missing or invalid token
403	Forbidden	Token valid but lacks the required permission or admin gate
404	Not Found	App doesn't exist OR your token can't access it (deliberately the same to avoid existence leaks)
409	Conflict	Optimistic-concurrency mismatch, idempotency-key collision, phone-number conflict
429	Too Many Requests	Rate limit hit
500	Internal	Platform error — captured in our monitoring

Common `code` values:

- `APP_NOT_FOUND`, `VALIDATION_ERROR`
- `INSUFFICIENT_BALANCE`, `INSUFFICIENT_POOL_BALANCE`
- `EXPECTED_VALUE_MISMATCH`
- `PHONE_NUMBER_IN_USE`
- `FEATURE_FORBIDDEN` (RBAC feature gate), `API_KEY_SCOPE`, `INTERNAL_ERROR` (default). Note: some ad-hoc 403s carry no `code` and fall back to `INTERNAL_ERROR`.

Retry recommendations:

- `429`: backoff (start 1s, double up to 60s).
- `500`: retry once after 1s; escalate to `support@everycallhandled.com` if persistent.
- `409` on idempotency-key collision: treat as success (the prior call landed).

17

Best Practices

Idempotency

Any endpoint that mutates billing or persists a one-shot side-effect (`cap-topup` , `cap-remove` , `adjust-credits` , `publish` , knowledge upload) accepts an `idempotencyKey` field. Pass a UUID per logical operation; re-runs become safe no-ops. Critical for retries through unreliable networks.

Draft-vs-live discipline

Always treat `PATCH /apps/{id}` as a **draft change**. Use `GET /apps/{id}` → `hasPendingPublish` to detect changes that aren't live yet and decide whether to surface an "Apply your changes" prompt. Don't assume edits are live until you've taken the assistant live.

Token security

- Store API tokens in env vars or a secrets manager
- Never commit them to version control
- Never expose them to client-side JavaScript
- Rotate periodically — contact `support@everycallhandled.com` for a new token, or revoke and re-mint scoped keys yourself
- Request (or mint) the minimum access you need — a capability-scoped key beats a broad token for single-purpose integrations

Webhook reliability

If you set `postChatCallback` , your endpoint should:

- Return `200` quickly (within 5s); do heavy processing asynchronously
- Be idempotent — the platform may retry
- Verify the signature (and check the `Authorization` header matches the value you supplied)

Rate limits

Soft per-account limits apply (~100 req/min steady-state, burst higher). If you anticipate sustained higher load (e.g. bulk migration), email `support@everycallhandled.com` and we'll raise the limit.

Polling patterns

For asynchronous operations (knowledge upload, going live):

- Initial poll: ~3 seconds after the request
- Subsequent polls: every 3-5 seconds
- Give up after 5 minutes for knowledge jobs, 30 seconds for going live

---

18

Support

Overview

- **Email:** `support@everycallhandled.com`
- **Status:** https://status.everycallhandled.com
- **Documentation issues / suggestions:** email support directly.

When reporting a bug, include:

1. The full curl command (with token redacted)
2. Full response body (including the `code` field)
3. Approximate timestamp (UTC)
4. The app id and your account id