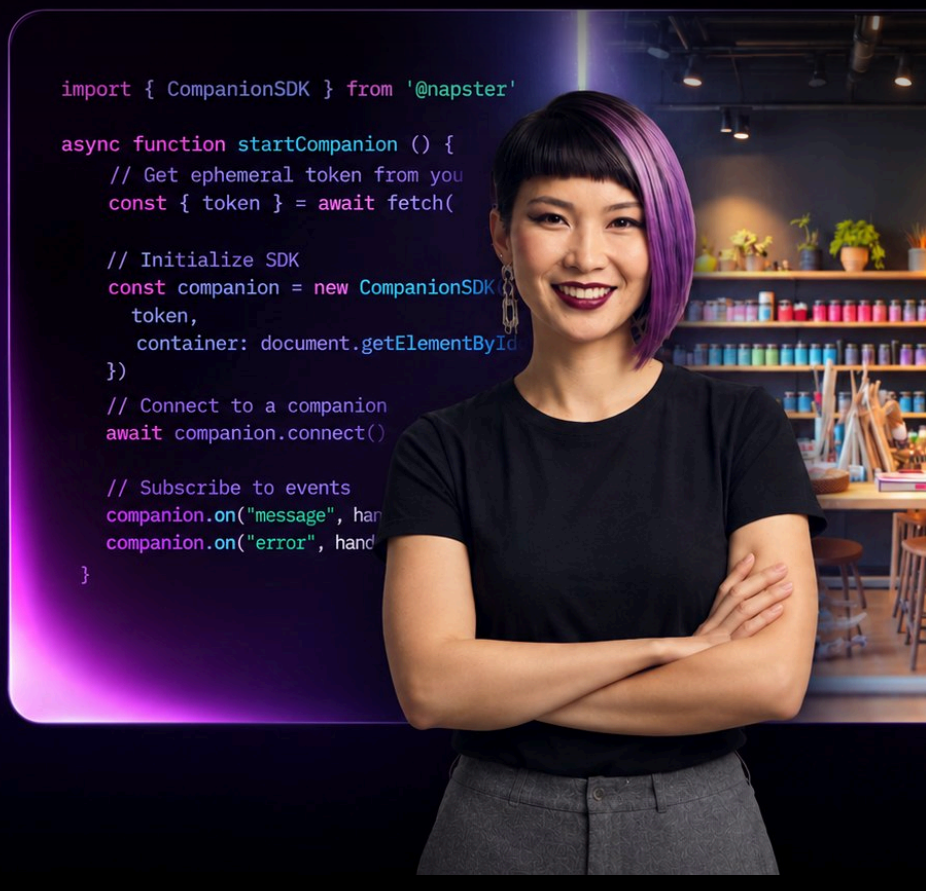
Napster

DEVELOPER DOCUMENTATION

Napster API *Product Manual.*

Build, deploy, and monitor Omniagents — AI agents with a face, a voice, and memory — across web, apps, phone, and Napster Station.



Napster.

CONTENTS

01 Introduction 4

- Meet the Napster API 5
- AI Coding Agents Support 9
- Quickstart 14
- Authentication 18
- Limits 20

02 Build Your Omniagent 23

- Get Started 24
- Agent Configuration 29
- Manage Your Agent 39
- Companion 41
- Digital Twins 45
- Avatar Models 49

TOOLS

- Overview 53
- Creating Tools 55
- Executing Tools 62
- WebSocket Tools 68

MCP

- Servers 74
- Connectors 85

KNOWLEDGE

- Files 89
- FAQs 93

03 Deploy Your Omniagent 96

- Get Started 97
- Add an AI Agent to Your Website 98
- Session Configuration 102

CHANNELS

- Choosing a Channel 106
- WebRTC 109
- WebSockets 114
- VoIP 121
- SIP 125
- Kiosk 138

MESSAGING

- Overview 143
- Server Events 144
- Client Commands 159

04 Monitor Your Omniagent 168

- Sessions 169

Webhooks 181

05 SDKs 190

WEB SDK
Overview 191
Installation 193
Initialization 195
Configuration 198
Features 204
Persistence 211
Events 219
Messaging 222
Methods 225
Examples 230
Troubleshooting 234

EDGE MCP
Overview 238
Installation 241
Tools 244
Resources 247
Testing 252

06 Guides 256

Create a New API Key 257
Create a Resource in Azure Portal 264
Re-enable an Azure Resource 271
Enable Digital Twins 276
Video Recordings 278

07 FAQs 286

FAQs 287

08 What's New 293

PLATFORM & API
September 29, 2026 294
August 19, 2026 300
August 12, 2026 303
July 13, 2026 306
June 26, 2026 309
June 1, 2026 311
May 7, 2026 314

WEB SDK
August 12, 2026 318

EDGE MCP
August 12, 2026 320

Napster

SECTION 01

Introduction

What the Napster API is, and how to make your first authenticated call.

Meet the Napster API	5
AI Coding Agents Support	9
Quickstart	14
Authentication	18
Limits	20



Meet the Napster API

The foundational layer powering multimodal coworkers across every surface your customer touches. Create an **Omniagent** — an AI agent with a persistent identity, voice, and visual appearance that carries its knowledge, memory, and actions across website, apps, phone, and beyond. One API, one agent, every channel.

AGENT ON YOUR WEBSITE

Add an AI agent that operates your website.

Not just a chat bubble — visitors ask in their own words, and the agent searches, navigates, fills, and books through your site's real functions, while the page updates in front of them.

Explore the guide →

Stage 1: Agentify the website

To agentify a website, install Edge MCP and expose the operations and live state you choose from your own code. One prompt in your AI coding tool does the integration.

Edge MCP →

Stage 2: Add the AI agent

Next, add the agent with the Web SDK. It discovers your tools automatically, so it operates the site while it talks to the visitor.

Web SDK →



Quick Start

Your first Omniagent in minutes!

Docs MCP & Agent Skills

Browse docs and get guidance from your coding tool.

Learn more →

Claude Plugin

All-in-one plugin with MCP, skills, and quickstart wizard.

Learn more →

Or build it yourself

Quickstart

Go from zero to a working Omniagent in minutes. Create a companion, configure your agent, and deploy it to a channel.

API Reference

Browse every endpoint with request and response schemas. Agents, companions, connections, tools, knowledge, and sessions.

Discord

Ask questions, share what you're building, and get help from the team and community.

Follow the path. Each step links to the in-depth resource you need.

1 **Get an API key**

Sign up at napster.com/developer — new accounts start with $6 of free credits, roughly 100 minutes of conversation. Prefer billing through your Azure subscription? Create a resource in the Azure Portal instead. Either way, you then generate an API key from the dashboard — see Authentication for details on how to authenticate API calls.

2 **Build**

Pick a companion, add knowledge and tools, configure voice and settings — then create your Omniagent.

3 **Deploy**

Choose a channel — WebRTC, WebSocket, VoIP, or SIP — and go live. Same agent, every surface.

4 **Monitor**

Track sessions, review transcripts, and refine your agent over time.

Capabilities

Capability	Details
Real-time video avatars	2D avatars with lip sync and natural expressions
Persistent identity and memory	Agents maintain a consistent persona and remember users across sessions and channels
Tool integration	Connect agents to real systems through custom functions to trigger workflows, generate content, and more
Multiple architectures	Realtime (Azure OpenAI by default, or OpenAI), Cascade (pick the speech recognition, language, and voice providers separately, including Anthropic), or connect an agent you've built on HUMAIN. Use Napster-managed infrastructure or bring your own credentials
Web SDK	Works with React, Vue, Angular, or vanilla JS; mount a companion in any DOM container
Low latency	~300ms response times
Multilingual	~30 languages supported
2,000+ companions	Pre-built expert agents, or wire your own knowledge base and tools
Built-in safety	Content moderation, crisis detection, topic restrictions, and auditability are always active

Compliance and security

- **SOC 2 Type II** — report available under NDA
- **GDPR & CCPA** — US and EU data protection
- **Encryption** — AES + TLS 1.2+ at rest and in transit
- **Pen testing** — annual third-party security audits
- **Data residency** — regional options for enterprise
- **Tenant isolation** — your data stays in your environment; no cross-customer data mixing, no training on your data

AI Coding Agents Support

Turn your favorite coding agent into a Napster API expert with access to docs, tools, and best practices

Napster provides three layers of support for AI coding agents — an MCP server for documentation access, agent skills for architectural guidance, and a Claude plugin that bundles everything together. Use one or all three depending on your tool.

Quick start prompt

Copy this prompt and paste it into your AI coding tool. It walks you through building and deploying your first Omniagent end to end — API key setup, persona creation, agent configuration, tools, knowledge, and channel deployment.

MCP server

For coding agents that support the Model Context Protocol, Napster provides a free MCP server with tools for browsing and searching the documentation site. This gives your coding agent direct access to API references, guides, and examples without leaving the editor.

Add the Napster docs MCP server to your tool:

> CLAUDE CODE

```
claude mcp add napster-docs --transport http https://developers.napster.com/mcp
```

CURSOR

Add this to your MCP configuration:

```
{"napster-docs": {"url": "https://developers.napster.com/mcp"}}
```

Or install directly: Add to Cursor

VS CODE / COPILOT

```
code --add-mcp '{"name":"napster-
docs","type":"http","url":"https://developers.napster.com/mcp"}'
```

CODEX

```
codex mcp add --url https://developers.napster.com/mcp napster-docs
```

Agent skills

Napster publishes open-source skills for coding agents that provide architectural guidance and best practices for building omniagents — covering workflow design, channel configuration, tool integration, and deployment patterns. Combined with the MCP server, skills give your coding agent deep expertise about building with the Napster API so it can look up the docs, understand the architecture, and generate code that follows best practices.

```
npx skills add napster/omniagent-api-skills
```

Claude plugin

The Napster plugin bundles the MCP server and agent skills into a single package with an interactive quickstart wizard. Setup depends on which Claude environment you use.

Claude Code (CLI)

> ℹ This is the recommended setup. The CLI always pulls the latest skills and MCP server.

Step 1. Add the Napster marketplace:

```
/plugin marketplace add https://github.com/Napster/omniagent-api-skills.git
```

Step 2. Install the plugin:

```
/plugin install napster-omniagent-api@napster
```

To update to the latest version later:

```
/plugin marketplace update napster
```

Claude browser & desktop

For the browser and desktop versions, you need to download the plugin manually first.

Step 1 — Download the plugin

Go to the Napster API Skills repo, click the green **Code** button, and select **Download ZIP**. Save the file to your machine.

> ⚠ Skills installed from a ZIP won't update automatically. If we publish new skills, you'll need to re-download and re-upload. The Claude Code CLI handles updates for you.

Step 2 — Install the plugin

> CLAUDE (BROWSER)

1. Go to claude.ai
2. In the left sidebar, click **Customize**
3. Click the **+** button to add a plugin, then choose **Create plugin → Upload plugin**
4. Select the ZIP file you downloaded in Step 1
5. Once uploaded, click the plugin in the left sidebar — the skills install automatically
6. Open the **Connectors** tab — you'll see **Napster Docs** (the MCP server). Click **Install**

> CLAUDE (DESKTOP)

1. Open Claude Desktop
2. Go to **Settings → Plugins**
3. Click **Add plugin → Upload plugin**
4. Select the ZIP file you downloaded in Step 1
5. Once uploaded, the skills install automatically
6. Go to **Settings → Connectors** — you'll see **Napster Docs** (the MCP server). Click **Install**

What you get

Once installed, the plugin gives Claude:

- Access to the full Napster API documentation via MCP
- Architectural guidance for building omniagents
- A quickstart wizard that walks you through creating your first agent

GitHub repository

Source code for the skills and everything else is available on GitHub:

github.com/Napster/omniagent-api-skills

Quickstart

Get started with the Napster API in minutes

This guide walks you through building and deploying your first Omniagent — a single AI agent with consistent voice, personality, and knowledge, available on any channel.

> Don't have an account yet? Sign up at napster.com/developer — you start with $6 of free credits, roughly 100 minutes of conversation. Or create a resource in the Azure Portal if you prefer billing through your Azure subscription.

Before you start, you'll need an API key. You can generate one from the dashboard — see Authentication for details.

> Your API key is a secret. Keep it private and only use it on the server side — never expose it in client-side code.

1 Choose a companion

Start by choosing a companion — the identity layer that defines your agent's appearance, personality, and behavior. It's a separate resource from the agent itself so one companion can front several agents, and the avatar is generated once no matter how often you reconfigure them — see Companion vs. Omniagent. Browse the catalog or create a custom one.

Browse the companion catalog (API reference):

```
curl -X GET https://companion-api.napster.com/public/companions/napster-stock \
  -H "X-Api-Key: $NAPSTER_API_KEY"
```

Or create a custom one (API reference):

```
curl -X POST https://companion-api.napster.com/public/companions \
  -H "X-Api-Key: $NAPSTER_API_KEY" \
  -H "Content-Type: application/json" \
  -d '{
    "firstName": "Alex",
    "lastName": "Chen",
    "description": "A senior tech support specialist who has been helping customers
for over a decade. Warm, patient, and detail-oriented. Explains technical concepts in
plain language and always walks users through solutions step by step. Keeps a
professional but approachable tone, occasionally uses light humor to put frustrated
users at ease.",
    "pictureUrl": "https://example.com/images/alex-avatar.png"
  }'
```

This uses the default v1 avatar model (image input). For higher quality avatars using a video reference, set `version` to `"v2"` and provide a `videoUrl` instead. See Avatar Models for details.

You can also equip your agent with tools and knowledge to expand what it can do and what it knows.

Learn more about building your Omniagent

2 Create your Omniagent

Bring everything together into a persistent agent. This combines the companion, voice, tools, knowledge, and model settings into a single reusable resource (API reference):

```
curl -X POST https://companion-api.napster.com/public/agents \
  -H "X-Api-Key: $NAPSTER_API_KEY" \
  -H "Content-Type: application/json" \
  -d '{
    "companionId": "comp_abc123",
    "name": "Support Agent",
    "voiceId": "alloy",
    "functions": ["func_abc123", "func_def456"],
    "providerSettings": {
      "temperature": 0.7
    }
  }'
```

The response returns an agent `id` that you use to deploy it to any channel.

3 Deploy to a channel

Create a WebRTC session from your Omniagent with a single call (API reference):

```
curl -X POST https://companion-api.napster.com/public/agents/agent_abc123/connections
\
  -H "X-Api-Key: $NAPSTER_API_KEY" \
  -H "Content-Type: application/json" \
  -d '{
    "channelType": "webrtc"
  }'
```

The response returns a `token` and a `connection` object with an `id`. Use the token to initialize the Web SDK and embed the agent directly in your application. Store the connection `id` on your backend to retrieve session details later, such as transcripts or duration.

Install the Web SDK:

```
npm install @touchcastllc/napster-companion-api @reduxjs/toolkit
```

Then use it in your React app:

```
import React, { useEffect, useRef } from "react";
import { NapsterCompanionApiSdk } from "@touchcastllc/napster-companion-api";
import type { NapsterCompanionApiInstance } from "@touchcastllc/napster-companion-
api";
import "@touchcastllc/napster-companion-api/styles";

export function CompanionWidget({ token }: { token: string }) {
  const containerRef = useRef<HTMLDivElement>(null);
  const instanceRef = useRef<NapsterCompanionApiInstance | null>(null);

  useEffect(() => {
    const initSDK = async () => {
      if (!containerRef.current) return;

      const result = await NapsterCompanionApiSdk.init(token, {
        mountContainer: containerRef.current,
        position: "bottom-right",
      });
      instanceRef.current = result;
    };

    initSDK();

    return () => {
      instanceRef.current?.destroy();
    };
  }, []);

  return <div ref={containerRef} style={{ width: "100%", height: "100%" }} />;
}
```

For examples in Vue, Angular, and vanilla JavaScript, see Examples in the Web SDK section, and the full Web SDK docs for configuration, events, and methods. You can also deploy the same agent to WebSockets, VoIP, or SIP.

Learn more about deploying your Omniagent

4 Monitor your agent

Once your agent is live, track how it's performing. Review analytics, browse conversation logs, and use those insights to refine your agent over time.

Learn more about monitoring your agent

Authentication

Learn how to authenticate requests to the Napster API

API keys

The Napster API uses API keys to authenticate requests. Every API key is scoped to an **organization** and a **project**. All resources — companions, tools, knowledge collections, and FAQ collections — live under a specific project.

All API requests must include the key in the `X-Api-Key` header:

```
X-Api-Key: <your-api-key>
```

If the key is missing, expired, or invalid, the server responds with `401 Unauthorized`.

> ⚠️ Your API key is a secret. Keep it private and only use it on the server side — never expose it in client-side code.

Getting an API key

No account yet? Sign up at napster.com/developer — or via the Azure Portal if you bill through Azure.

For a step-by-step walkthrough with screenshots, see the Create a New API Key guide.

In short: open the dashboard, navigate to **Keys**, and click **+ Create API key**. You'll choose a provider and a deployment type (Napster managed or your own custom deployment).

Using your API key

Include your key in the `X-Api-Key` header on every request:

```
curl https://companion-api.napster.com/public/companions/napster-stock \
  -H "X-Api-Key: $NAPSTER_API_KEY"
```

The same key works for all Napster API endpoints — managing companions, uploading knowledge, configuring tools, and everything else.

Limits

Rate limits, concurrency limits, and resource limits for the Napster API

> Need higher limits? Contact us at help.napster.com to discuss your requirements.

Rate limits

Connection endpoints are rate-limited to **1 request per second** per API key. The limit is shared across these endpoints:

- `POST /public/connections`
- `POST /public/ws-connections`
- `POST /public/agents/{agentId}/connections`
- `DELETE /public/connections/{connectionId}`

If you exceed this limit, the API returns a `429` status code with a `Retry-After` header indicating how many seconds to wait before retrying:

```
HTTP/1.1 429 Too Many Requests
Retry-After: 1
```

```
{
  "code": "RequestRateLimitExceeded",
  "description": "Too many requests. Please try again later.",
  "type": 12,
  "numericType": 12
}
```

Always respect the `Retry-After` header value rather than hardcoding a wait time.

Concurrency limits

Each organization can run up to **5 concurrent sessions per transport**. The limit is tracked independently for each transport type:

Transport	Concurrent sessions
WebRTC	5
WebSocket	5
VoIP	5
SIP	5
Total	**20**

If you try to create a session that would exceed the limit, the API returns a `409` status code:

```
{
  "code": "ConcurrentSessionLimitReached",
  "description": "Organization has reached the maximum of 5 concurrent 'webrtc' sessions.",
  "type": 3,
  "numericType": 3
}
```

Sessions count toward the limit from creation until they are closed. Idle sessions still count.

Telephony behavior

When concurrency limits are reached on phone channels (VoIP and SIP), the system plays a pre-recorded audio message informing the caller that no agents are available and asks them to try again later. The call is then dropped.

Agents per organization

Each organization can hold up to **100 agents**. The limit counts all agents in the organization, across every project and API key.

When creating an agent would exceed the limit, `POST /public/agents` returns a `409` status code:

```
{
  "code": "AgentLimitExceeded",
  "description": "Organization has reached the maximum of 100 agents (100 used).",
  "type": 3,
  "numericType": 3
}
```

Delete agents you no longer use to free up capacity, or contact us to raise the limit for your organization.

Knowledge collections

Each knowledge collection can contain up to **30 files**. Adding a file to a full collection returns `400 KnowledgeBaseFileLimitReached`. Per-file size limits depend on the file type — see supported file types.

FAQ collections

Each FAQ collection can hold up to **50 Q&A pairs**. Creating a collection or adding pairs past that limit returns `400 FaqLimitExceeded`.

FAQ collections per agent

An agent can have **1 FAQ collection** attached. The limit applies everywhere `faqCollections` is accepted:

- `POST /public/agents` and `PATCH /public/agents/{agentId}`
- `PUT /public/agents/{agentId}/channels/{channelType}`
- `POST /public/connections` and `POST /public/ws-connections`

Passing more than one ID returns a `400` validation error on `faqCollections` with the message "At most 1 FAQ collection can be attached." To cover more topics, add the Q&A pairs to the same FAQ collection.

Napster

SECTION 02

Build Your Omniagent

Identity, configuration, tools, MCP, and knowledge.

Get Started 24

Agent Configuration 29

Manage Your Agent 39

Companion 41

Digital Twins 45

Avatar Models 49

TOOLS

Overview 53

Creating Tools 55

Executing Tools 62

WebSocket Tools 68

MCP

Servers 74

Connectors 85

KNOWLEDGE

Files 89

FAQs 93



Get Started

Build and assemble your first Omniagent step by step

An **Omniagent** is a single AI agent with a consistent voice, personality, knowledge, and memory — available on any channel. Define it once, deploy it to web, WebSocket, or phone. Update it once, every channel picks up the change.

Companion vs. Omniagent

You'll work with two resources, and they answer different questions:

- The **companion** is the identity — the face, name, and personality. It answers *who is this?*
- The **Omniagent** is the deployable assembly — a companion plus voice, language, tools, knowledge, and model settings. It answers *what can it do, and how does it run?*

They're separate on purpose. A companion is **reusable**: the same character can front several agents — say, "Alex Chen" as your Support agent (support tools, support knowledge) and as your Sales agent (different tools, different knowledge) — one consistent face across products. And because the generated avatar lives on the companion, it's created **once**: you can reconfigure, retune, or rebuild your agents as often as you like without ever touching the avatar. Identity stays stable while configuration evolves.

This guide walks you through building the pieces and assembling them into your first Omniagent.

1 Pick a companion

The companion defines your agent's identity — how it looks and behaves. Browse the catalog of pre-built companions and pick an `id` from the results:

```
curl https://companion-api.napster.com/public/companions/napster-stock?pageSize=10 \
  -H "X-Api-Key: $NAPSTER_API_KEY"
```

If none of the catalog companions fit your use case, create a custom one with your own personality and avatar:

```
curl -X POST https://companion-api.napster.com/public/companions \
  -H "X-Api-Key: $NAPSTER_API_KEY" \
  -H "Content-Type: application/json" \
  -d '{
    "firstName": "Alex",
    "lastName": "Chen",
    "description": "A senior tech support specialist. Warm, patient, and detail-
oriented. Explains technical concepts in plain language.",
    "pictureUrl": "https://example.com/images/alex-avatar.png"
  }'
```

Save the `id` from the response — you'll need it when creating the agent.

For the full set of companion options (gender, ethnicity, personality), see Companion.

2 Add tools (optional)

Tools let your agent take actions during conversations — look up orders, book appointments, trigger workflows. Create a tool by defining what it does and where it executes:

```
curl -X POST https://companion-api.napster.com/public/functions \
  -H "X-Api-Key: $NAPSTER_API_KEY" \
  -H "Content-Type: application/json" \
  -d '{
    "data": {
      "name": "get_order_status",
      "description": "Look up the current status of a customer order by order ID",
      "parameters": {
        "type": "object",
        "properties": {
          "orderId": {
            "type": "string",
            "description": "The order ID to look up"
          }
        },
        "required": ["orderId"]
      }
    },
    "url": "https://your-server.com/api/order-status",
    "flow": "explicit"
  }'
```

Save the `id` from the response. You can create as many tools as you need.

For details on execution flows (implicit vs explicit), tool prompts, and responding to tool calls, see Tools.

3 Add knowledge (optional)

Give your agent domain expertise so it can answer questions using your content instead of relying solely on the base LLM.

Upload documents — create a knowledge collection and upload files:

```
# Create a collection
curl -X POST https://companion-api.napster.com/public/knowledge-bases \
  -H "X-Api-Key: $NAPSTER_API_KEY" \
  -H "Content-Type: application/json" \
  -d '{
    "name": "Product Docs"
  }'

# Upload a file to the collection
curl -X POST https://companion-api.napster.com/public/knowledge-bases/kb_abc123/files
\
  -H "X-Api-Key: $NAPSTER_API_KEY" \
  -H "Content-Type: application/json" \
  -d '{
    "url": "https://example.com/docs/product-guide.pdf"
  }'
```

Add FAQs — for questions where you need a specific, controlled answer:

```
# Create an FAQ collection
curl -X POST https://companion-api.napster.com/public/faqs \
  -H "X-Api-Key: $NAPSTER_API_KEY" \
  -H "Content-Type: application/json" \
  -d '{
    "name": "Support FAQs"
  }'

# Add a Q&A pair
curl -X POST https://companion-api.napster.com/public/faqs/faq_abc123/items \
  -H "X-Api-Key: $NAPSTER_API_KEY" \
  -H "Content-Type: application/json" \
  -d '{
    "question": "What is your return policy?",
    "answer": "You can return any item within 30 days of purchase for a full refund."
  }'
```

Save the collection IDs. For more on knowledge and FAQs, see Files and FAQs.

4 Create the Omniagent

Now bring everything together. Pass the companion ID, voice, and any tools, knowledge, and FAQs you created in the previous steps. Organizations can hold up to 100 agents — see Limits for the `409` you get past that point.

```
curl -X POST https://companion-api.napster.com/public/agents \
  -H "X-Api-Key: $NAPSTER_API_KEY" \
  -H "Content-Type: application/json" \
  -d '{
    "companionId": "comp_abc123",
    "name": "Support Agent",
    "voiceId": "alloy",
    "language": "en",
    "functions": ["fn_abc123"],
    "faqCollections": ["faq_abc123"],
    "knowledgeBaseId": "kb_abc123",
    "providerSettings": {
      "temperature": 0.7
    }
  }'
```

Parameter	Type	Required	Description
`companionId`	string	Yes	The companion from step 1.
`voiceId`	string	Conditional	The voice for speech output (e.g. `alloy`, `echo`, `shimmer`). Required for companions, not required for digital twins.
`providerSettings`	object	Yes	Model and audio settings. See Configuration.
`name`	string	Yes	A label for this agent.
`language`	string	No	Lock to a specific language using an ISO code (e.g. `en`, `es`). Defaults to English.
`functions`	string[]	No	Tool IDs from step 2.
`faqCollections`	string[]	No	The FAQ collection ID from step 3. At most one.
`knowledgeBaseId`	string	No	Knowledge collection ID from step 3.
`mode`	string	No	`conversation` (default) or `puppeteer`. See Mode.

Your Omniagent is ready. The `id` in the response is what you use to deploy it.

Agent Configuration

All the settings available when creating or updating your Omniagent

When you create an Omniagent, you define everything about how it behaves — its identity, resources, voice, language, and model settings. These apply consistently across every channel the agent is deployed to.

This page is the complete reference for all agent-level settings.

Setting and updating configuration

Every field on this page can be set when you create the agent or updated later. Only the fields you include are changed — everything else stays the same.

Create an agent with your initial configuration:

```
curl -X POST https://companion-api.napster.com/public/agents \
  -H "X-Api-Key: $NAPSTER_API_KEY" \
  -H "Content-Type: application/json" \
  -d '{
    "companionId": "comp_abc123",
    "name": "Support Agent",
    "voiceId": "alloy",
    "providerSettings": {
      "temperature": 0.7
    }
  }'
```

Update any field later with a PATCH request — only include what you want to change:

```
curl -X PATCH https://companion-api.napster.com/public/agents/agent_abc123 \
  -H "X-Api-Key: $NAPSTER_API_KEY" \
  -H "Content-Type: application/json" \
  -d '{
    "language": "es",
    "voiceId": "shimmer"
  }'
```

Identity

These fields define who your agent is.

Name

A label to identify the agent in your project — for example, `"Support Agent"`, `"Sales Bot"`, or `"Onboarding Assistant"`. Required when you create an agent.

Companion

The `companionId` is required and determines the agent's visual appearance, personality, and behavior. See Companion for how to pick or create one.

Language

When you set `language`, the agent strictly follows that language for the entire session — it will not switch even if the user speaks in a different language. Use ISO language codes — `en`, `es`, `fr`, `de`, etc.

If you omit `language`, the agent defaults to English but can switch to a different language when the user asks.

```
{
  "companionId": "comp_abc123",
  "name": "Support Agent",
  "language": "es"
}
```

Supported languages (Azure OpenAI)

The following language codes are supported when using the Azure OpenAI provider:

Code	Language	Code	Language	Code	Language
`af`	Afrikaans	`he`	Hebrew	`no`	Norwegian
`ar`	Arabic	`hi`	Hindi	`fa`	Persian
`hy`	Armenian	`hu`	Hungarian	`pl`	Polish
`az`	Azerbaijani	`is`	Icelandic	`pt`	Portuguese

Code	Language	Code	Language	Code	Language
`be`	Belarusian	`id`	Indonesian	`ro`	Romanian
`bs`	Bosnian	`it`	Italian	`ru`	Russian
`bg`	Bulgarian	`ja`	Japanese	`sr`	Serbian
`ca`	Catalan	`kn`	Kannada	`sk`	Slovak
`zh`	Chinese	`kk`	Kazakh	`sl`	Slovenian
`hr`	Croatian	`ko`	Korean	`es`	Spanish
`cs`	Czech	`lv`	Latvian	`sw`	Swahili
`da`	Danish	`lt`	Lithuanian	`sv`	Swedish
`nl`	Dutch	`mk`	Macedonian	`tl`	Tagalog
`en`	English	`ms`	Malay	`ta`	Tamil
`et`	Estonian	`mr`	Marathi	`th`	Thai
`fi`	Finnish	`mi`	Maori	`tr`	Turkish
`fr`	French	`ne`	Nepali	`uk`	Ukrainian
`gl`	Galician			`ur`	Urdu
`de`	German			`vi`	Vietnamese
`el`	Greek			`cy`	Welsh

Mode

The `mode` field sets how the agent speaks:

Mode	Behavior
`conversation`	The default. The agent listens to the user and answers out loud.
`puppeteer`	The agent speaks only the lines your client sends with the `talk` command. Use it for scripted presentations, guided tours, or when your own system decides every line.

```
{
  "mode": "puppeteer"
}
```

Set `mode` on the agent, or per session on `POST /public/connections` and `POST /public/ws-connections`. Sessions started from an agent use the agent's mode.

Puppeteer mode requires an API key that uses the Cascade architecture. With any other key, opening a session in puppeteer mode returns `400 UnsupportedSessionMode`. A Cascade key with only text-to-speech is enough. If the key also has speech recognition and a language model, the agent still listens and replies, but its replies arrive as text in `message_received` instead of being spoken.

> Agents in puppeteer mode can't use phone channels. Adding a SIP or VoIP channel config returns `400 TelephonyChannelNotAllowed`.

Knowledge

Knowledge gives your agent domain expertise through two types of resources.

Documents

The `knowledgeBaseId` field attaches a knowledge collection to the agent. The agent retrieves relevant documents from this collection during conversations to ground its responses in your content. See Files for details.

FAQs

The `faqCollections` field takes an array with the ID of one FAQ collection. When a user asks a question that matches one of your FAQs, the agent responds with the answer you defined rather than generating one. See FAQs for details.

```
{
  "knowledgeBaseId": "kb_product_docs",
  "faqCollections": ["faq_support"]
}
```

You can attach at most one FAQ collection. Passing more than one ID returns a `400` validation error.

Tools

The `functions` field accepts an array of tool IDs. These define what your agent can do during conversations — look up data, trigger workflows, generate content. You can attach as many tools as you need.

Tools must be created before attaching them to the agent. See Tools for details.

```
{
  "functions": ["fn_order_status", "fn_returns", "fn_book_appointment"]
}
```

Tags

The `tags` field accepts string-to-string key-value pairs for labeling the agent. Use tags to categorize agents by source, environment, team, or any dimension relevant to your application. Tags are stored on the agent and returned on all sessions created from it.

```
{
  "tags": {
    "env": "production",
    "team": "support",
    "source": "ios-app"
  }
}
```

Tag keys are strings with a maximum length of 64 characters. Values are strings with a maximum length of 512 characters.

Idle timeout

By default, a session closes after **3 minutes** without audio or messages, with `closeReason: "idle_timeout"`. The client is warned 60, 30, and 10 seconds before — see Session errors and shutdown. Set `disableIdleTimeout` to `true` on the agent to keep all sessions open indefinitely, even if no audio or messages are exchanged.

```
{
  "disableIdleTimeout": true
}
```

Voice

The `voiceId` field sets the voice for the agent's speech output. It is required when using a catalog or custom companion. When using a digital twin, `voiceId` is not required — the voice comes from the `voiceUrl` provided during digital twin creation.

Provider	Supported voices
Azure OpenAI	`alloy`, `ash`, `ballad`, `coral`, `echo`, `sage`, `shimmer`, `verse`, `marin`, `cedar`

```
{
  "voiceId": "alloy"
}
```

Web search

When `useWebSearch` is `true`, the agent can search the web during conversations to look up real-time information beyond its knowledge base — current events, product details, public data, and more.

```
{
  "useWebSearch": false
}
```

Web search is enabled by default. Pass `"useWebSearch": false` explicitly when creating or updating an agent if you want the agent to rely only on its provided knowledge. You can also override the setting per channel via channel overrides.

Provider settings

The `providerSettings` object controls model behavior and audio processing.

> The provider settings below apply to keys on the **Realtime** architecture (the **Azure OpenAI** and **OpenAI** providers). On a **Cascade** key, only some of them apply — see Cascade architecture. If your key uses the **Other** architecture (**HUMAIN**), contact us for platform-specific configuration details.

Field	Type	Description
`temperature`	float	Controls randomness in the agent's responses. Lower values (e.g. `0.3`) produce more focused output; higher values (e.g. `0.9`) increase variety.
`instructions`	string	Overrides the companion's system prompt. Use this to replace the default prompt with custom instructions without modifying the companion itself.
`turnDetection`	object	Voice Activity Detection (VAD) configuration. Controls how the system detects when the user starts and stops speaking.
`noiseReduction`	object	Noise reduction configuration. Reduces background noise from the user's microphone input.

```
{
  "providerSettings": {
    "temperature": 0.7,
    "turnDetection": {
      "threshold": 0.9,
      "prefix_padding_ms": 400,
      "silence_duration_ms": 500
    },
    "noiseReduction": {
      "type": "nearField"
    }
  }
}
```

Temperature

The `temperature` field controls how varied vs. consistent the agent's responses are. Lower values keep the agent on-script; higher values let it improvise.

Range	Behavior	Best for
`0.0 – 0.3`	Strict, deterministic. The agent says the same thing the same way.	Classifiers, scripted flows, agents that should never deviate
`0.3 – 0.6`	Focused but natural. Small variation in phrasing, same substance.	Customer support, technical Q&A, documentation assistants
`0.6 – 0.9`	Warm and varied. Different phrasing every time, on-message.	Sales, onboarding, conversational hosts
`1.0 – 1.2`	Loose and creative. Noticeable variety, occasionally off-script.	Brainstorming partners, casual chat
`1.5+`	Highly creative. Surprising, sometimes unpredictable.	Creative writing, ideation, experimentation

For production voice agents, `0.5` **to** `0.7` is the safe range. Higher values produce more interesting language but also more drift — answers vary between sessions, occasionally hallucinate, occasionally drop instructions.

Turn detection

The `turnDetection` object configures Voice Activity Detection (VAD) — how the system decides when the user has started and stopped speaking. This directly affects interruption behavior and response timing.

Field	Type	Description
`threshold`	float	Activation threshold for detecting speech. Higher values (e.g. `0.8`) require louder, clearer speech to trigger; lower values (e.g. `0.3`) are more sensitive.
`prefix_padding_ms`	integer	Milliseconds of audio to include before detected speech begins. Prevents clipping the start of an utterance.
`silence_duration_ms`	integer	Milliseconds of silence required before the system considers the user done speaking. Lower values (e.g. `200`) make the agent respond faster; higher values (e.g. `800`) wait longer for the user to continue.

Noise reduction

The `noiseReduction` object accepts a single `type` field:

Value	Description
`nearField`	Optimized for close-range microphones — laptops, headsets, phones held to the ear.
`farField`	Optimized for distant microphones — speakerphones, smart speakers, conference room setups.

> If you omit `noiseReduction`, no noise processing is applied. Choose the mode that matches your user's typical microphone distance for best results.

Cascade architecture

With a Cascade API key, the agent runs on separate speech-recognition, language, and text-to-speech models instead of one realtime model. Tools, knowledge, FAQs, and digital twins work the same way. These features behave differently:

- **MCP isn't supported.** Opening a session with MCP servers or connectors attached returns `400 UnsupportedProvider`.
- **Only some provider settings apply.** `instructions` works as usual, and `turnDetection` applies when the session starts. `temperature` and `noiseReduction` are ignored.
- **Mid-session `set_settings` only changes `instructions` and tools.**
- **User speech events arrive after the transcript.** `speech_started` and the related events arrive together once the user's speech is transcribed, so you can't use `speech_started` to detect the user interrupting. Interrupted responses report `reason: "interrupted"` instead of `"turn_detected"`.
- Puppeteer mode **is available.** The agent speaks only the lines your client sends.

Channel overrides

All of the settings above apply across every channel by default. If you need different behavior on a specific channel, you can override `providerSettings`, `functions`, `faqCollections`, and `knowledgeBaseId` per channel using the channel config.

For example, to use different turn detection settings and attach an additional tool on SIP calls:

```
curl -X PUT https://companion-api.napster.com/public/agents/agent_abc123/channels/sip \
  -H "X-Api-Key: $NAPSTER_API_KEY" \
  -H "Content-Type: application/json" \
  -d '{
    "functions": ["fn_order_status", "fn_returns", "fn_transfer_call"],
    "providerSettings": {
      "turnDetection": {
        "threshold": 0.7,
        "silence_duration_ms": 800
      }
    }
  }'
```

Channel overrides apply to every session on that channel. If omitted, the agent's base configuration is used. See the WebRTC, WebSockets, VoIP, and SIP guides for channel-specific setup.

Manage Your Agent

List, update, and delete your Omniagents

After creating an agent, you can list all your agents, update their configuration, or delete them.

List agents

Retrieve all agents in your project:

```
curl https://companion-api.napster.com/public/agents \
  -H "X-Api-Key: $NAPSTER_API_KEY"
```

Get a single agent

```
curl https://companion-api.napster.com/public/agents/agent_abc123 \
  -H "X-Api-Key: $NAPSTER_API_KEY"
```

Update an agent

Use `PATCH` to update any field on the agent. Only the fields you include in the request body are changed — everything else stays the same.

```
curl -X PATCH https://companion-api.napster.com/public/agents/agent_abc123 \
  -H "X-Api-Key: $NAPSTER_API_KEY" \
  -H "Content-Type: application/json" \
  -d '{
    "language": "es",
    "voiceId": "shimmer"
  }'
```

You can update any of the fields from the creation request: `companionId`, `name`, `voiceId`, `language`, `functions`, `faqCollections`, `knowledgeBaseId`, and `providerSettings`. Changes apply to all channels that use this agent.

Delete an agent

```
curl -X DELETE https://companion-api.napster.com/public/agents/agent_abc123 \
  -H "X-Api-Key: $NAPSTER_API_KEY"
```

> ⚠️ Deleting an agent removes it for all channels — WebRTC, WebSocket, VoIP, and SIP. To remove only a specific channel configuration while keeping the agent, delete the channel config instead (e.g. `DELETE /public/agents/{agentId}/channels/sip`).

Managing channel configs

Each channel can have its own configuration that overrides the agent's defaults. You can retrieve or remove a channel config without affecting the agent itself:

```
# Get a channel config (e.g. SIP)
curl https://companion-api.napster.com/public/agents/agent_abc123/channels/sip \
  -H "X-Api-Key: $NAPSTER_API_KEY"

# Delete a channel config (keeps the agent, removes channel overrides)
curl -X DELETE https://companion-api.napster.com/public/agents/agent_abc123/channels/sip \
  -H "X-Api-Key: $NAPSTER_API_KEY"
```

For details on setting up channel configs, see the channel-specific guides: WebRTC, WebSockets, VoIP, SIP.

Companion

Define the visual identity and personality of your agent

The companion is the core identity of your agent — how it looks and behaves. You can choose from an existing companion in the catalog, or create a custom one from scratch.

A companion is a separate, reusable resource: the same companion can front multiple agents with different tools and knowledge, and because the generated avatar lives here, you can reconfigure those agents freely without ever regenerating it.

> Looking to recreate a real person with their actual appearance and voice? See Digital Twins.

Using an existing companion

Napster provides a catalog of pre-built companions with ready-made avatars, voices, and personalities. Catalog companions can be used immediately when creating an agent.

To browse the public catalog, call `GET /public/companions/napster-stock`. You can filter results by `search`, `gender`, and `ethnicity`, and paginate with `pageIndex` and `pageSize`. See the API reference for the full list of query parameters.

```
curl https://companion-api.napster.com/public/companions/napster-stock?pageSize=10 \
  -H "X-Api-Key: $NAPSTER_API_KEY"
```

Creating a custom companion

To create your own companion, define:

- **Personality** — use the `description` field to describe the companion's role, personality, tone, and communication style. This shapes how the agent behaves and interacts with users.
- **Headline** — a short tagline or subtitle for the companion (e.g. `"Senior Tech Support Specialist"`). Set via update after creation.
- **Visual appearance** — provide an image or video as the reference input to generate a custom avatar. See avatar models for the difference between v1 (image) and v2 (video).
- **Gender and ethnicity** — optionally specify the companion's gender and ethnicity to influence the generated avatar's appearance.

Creating with v1 (image input)

Provide a `pictureUrl` — a publicly reachable HTTPS URL of a reference image.

```
curl -X POST https://companion-api.napster.com/public/companions \
  -H "X-Api-Key: $NAPSTER_API_KEY" \
  -H "Content-Type: application/json" \
  -d '{
    "firstName": "Alex",
    "lastName": "Chen",
    "description": "A senior tech support specialist who has been helping customers for over
a decade. Warm, patient, and detail-oriented. Explains technical concepts in plain language
and always walks users through solutions step by step.",
    "pictureUrl": "https://example.com/images/alex-avatar.png",
    "gender": "male",
    "ethnicity": "Asian"
  }'
```

Creating with v2 (video input)

Provide a `videoUrl` and set `version` to `"v2"`. v2 must be enabled for your organization first — otherwise the request fails with a `CompanionV2NotEnabled` error.

```
curl -X POST https://companion-api.napster.com/public/companions \
  -H "X-Api-Key: $NAPSTER_API_KEY" \
  -H "Content-Type: application/json" \
  -d '{
    "firstName": "Alex",
    "lastName": "Chen",
    "description": "A senior tech support specialist who has been helping customers for over
a decade. Warm, patient, and detail-oriented. Explains technical concepts in plain language
and always walks users through solutions step by step.",
    "videoUrl": "https://example.com/videos/alex-speaking.mp4",
    "gender": "male",
    "ethnicity": "Asian",
    "version": "v2"
  }'
```

Parameter reference

Parameter	Type	Required	Description
`description`	string	Yes	The companion's role, personality, and communication style. This becomes the foundation of the agent's system prompt.
`firstName`	string	No	The companion's first name.
`lastName`	string	No	The companion's last name.
`pictureUrl`	string	No	URL of a reference image for avatar generation (v1).
`videoUrl`	string	No	URL of a reference video of the person speaking for avatar generation (v2).
`version`	string	No	Avatar model. `"v1"` (default) or `"v2"`. See Avatar Models.
`gender`	string	No	One of `male`, `female`, or `nonBinary`.
`ethnicity`	string	No	Use the ethnicities endpoint to get supported values.
`tags`	object	No	Key-value string pairs for labeling the companion.

After creating a companion, you can add a `headline` — a short tagline or subtitle (e.g. `"Senior Tech Support Specialist"`) — by updating it with `PATCH /public/companions/{companionId}`.

Ethnicity

The `ethnicity` field accepts a string value from a dynamic list maintained by the API. To retrieve the available ethnicities, call `GET /public/companions/ethnicities`:

```
curl https://companion-api.napster.com/public/companions/ethnicities \
  -H "X-Api-Key: $NAPSTER_API_KEY"
```

Using a companion in an agent

Use the companion's `id` as the `companionId` when creating an agent:

```
curl -X POST https://companion-api.napster.com/public/agents \
  -H "X-Api-Key: $NAPSTER_API_KEY" \
  -H "Content-Type: application/json" \
  -d '{
    "companionId": "comp_abc123",
    "name": "Support Agent",
    "voiceId": "alloy",
    "providerSettings": {
      "temperature": 0.7
    }
  }'
```

From here, configure the agent, add tools and knowledge, and deploy to a channel.

Listing your companions

To list your companions, call `GET /public/companions`. This returns all companions created under the project your API key is scoped to.

```
curl https://companion-api.napster.com/public/companions \
  -H "X-Api-Key: $NAPSTER_API_KEY"
```

To filter by type, use the `personaType` query parameter. Use `companion` for custom companions or `digitalTwin` for digital twins:

```
curl https://companion-api.napster.com/public/companions?personaType=companion \
  -H "X-Api-Key: $NAPSTER_API_KEY"
```

Digital Twins

Create a companion based on a real person's likeness and voice

A digital twin is a companion based on a real person. It uses the person's actual voice recording and likeness to produce an avatar that looks and sounds like them. Once created, a digital twin returns a companion ID that you use the same way as any other companion — attach it to an agent, deploy to any channel.

Digital twins work with every API key type except **Humain**. Opening a session with a digital twin on a Humain key returns `400 UnsupportedProvider`.

For fictional or generated personas, use a companion instead.

> ⚠ **Grant consent first.** An admin must enable digital twins in the dashboard, or the API fails with `DigitalTwinConsentRequired`.

Creating a digital twin

Call `POST /public/digital-twins` with the person's identity, a voice recording, and a reference image or video for the avatar.

```
curl -X POST https://companion-api.napster.com/public/digital-twins \
  -H "X-Api-Key: $NAPSTER_API_KEY" \
  -H "Content-Type: application/json" \
  -d '{
    "firstName": "Sarah",
    "lastName": "Johnson",
    "description": "VP of Customer Success at Acme Corp. Confident, warm, and knowledgeable.
Speaks clearly and directly, with a focus on helping customers get the most out of the
product.",
    "gender": "female",
    "ethnicity": "Caucasian",
    "voiceUrl": "https://example.com/recordings/sarah-voice-sample.wav",
    "videoUrl": "https://example.com/videos/sarah-speaking.mp4",
    "externalClientId": "sarah-johnson-001",
    "version": "v2"
  }'
```

Parameter reference

Parameter	Type	Required	Description
`firstName`	string	Yes	The person's first name.
`lastName`	string	Yes	The person's last name.
`description`	string	Yes	Role, personality, and communication style. Becomes the foundation of the agent's system prompt.
`gender`	string	Yes	One of `male`, `female`, or `nonBinary`.
`ethnicity`	string	Yes	Use the ethnicities endpoint to get supported values.
`voiceUrl`	string	No	URL of a voice recording of the person. Used to clone their voice. Must be publicly reachable over HTTPS. Optional for `v2`, where `videoUrl` alone can supply both appearance and voice.
`externalClientId`	string	Yes	A unique identifier for the person. See External client ID below.
`pictureUrl`	string	No	URL of a reference image for avatar generation. Used by `v1`, and by `v2` when you don't have a reference video.
`videoUrl`	string	No	URL of a reference video of the person speaking. Used by `v2` for the highest quality.
`version`	string	No	Avatar model. `"v1"` (default) or `"v2"`. See Avatar Models.
`tags`	object	No	Key-value string pairs for labeling.

> All identity fields (`firstName` , `lastName` , `gender` , `ethnicity`) are required for digital twins, unlike custom companions where they are optional.

Voice recording requirements

The `voiceUrl` is what makes the digital twin sound like the real person. Provide a clear recording of the person speaking naturally — the longer and more natural the sample, the better the voice clone.

Requirement	Value
Supported formats	mp3, wav, m4a, flac, ogg, webm, mp4
Max file size	10 MB
Minimum duration	60 seconds
Maximum duration	300 seconds (5 minutes)

The URL must be publicly reachable over HTTPS.

External client ID

The `externalClientId` is your own unique identifier for the person this digital twin represents — an employee ID, CRM contact ID, or any stable ID from your system. It links the digital twin back to the real individual so you can correlate sessions, analytics, and updates across your platform. For example, if Sarah leaves the company and is replaced, you can create a new digital twin with a different `externalClientId` while keeping the old one for historical records.

Avatar models

Two avatar models are available — v1 (fast) and v2 (highest quality). Set the `version` field to `"v1"` or `"v2"`; if omitted, it defaults to `"v1"`. v1 takes a still image (`pictureUrl`). v2 takes a reference video (`videoUrl`) for the best results, or a still image (`pictureUrl`) when you don't have video. v2 must be enabled for your organization first — otherwise creation fails with a `CompanionV2NotEnabled` error. See Avatar Models for the full comparison.

Response

The response returns a companion object with an `id`:

```
{
  "id": "comp_abc123",
  "firstName": "Sarah",
  "lastName": "Johnson",
  "description": "VP of Customer Success at Acme Corp...",
  "gender": "female",
  "ethnicity": "Caucasian"
}
```

Listing your digital twins

Digital twins are listed through the same companions endpoint. Use the `personaType` query parameter to filter:

```
curl https://companion-api.napster.com/public/companions?personaType=digitalTwin \
  -H "X-Api-Key: $NAPSTER_API_KEY"
```

Using a digital twin in an agent

Use the returned `id` as the `companionId` when creating an agent — same as any other companion:

```
curl -X POST https://companion-api.napster.com/public/agents \
  -H "X-Api-Key: $NAPSTER_API_KEY" \
  -H "Content-Type: application/json" \
  -d '{
    "companionId": "comp_abc123",
    "name": "Sarah - Customer Success",
    "providerSettings": {
      "temperature": 0.7
    }
  }'
```

From here, follow the standard flow: configure the agent, add tools and knowledge, and deploy to a channel.

Avatar Models

Choose between v1 and v2 for avatar generation

> ⚠ v2 must be enabled for your organization before you can use it. Until then, creating a companion or digital twin with `"version": "v2"` — or connecting to one — fails with a `CompanionV2NotEnabled` error. To request access, contact us with your organization ID. There is no extra charge for v2.

Two avatar models are available — for both companions and digital twins. Both produce real-time video avatars. The difference is the reference input and the resulting quality.

	v1	v2
Input	Still image (`pictureUrl`)	Video (`videoUrl`) — or a still image (`pictureUrl`)
Speed	Fast generation	Slower generation (can take up to several hours)
Quality	Good quality, strong photorealism	Highest quality
Lip sync	Good	Best accuracy
Expressions	Standard	Natural facial expressions
Facial hair	Standard	Better results

Set the `version` field to `"v1"` or `"v2"` when creating a companion or digital twin. If omitted, it defaults to `"v1"`.

v1 — Image input

Provide a `pictureUrl` — a publicly reachable HTTPS URL of a reference image. For best results, use a 16:9 image with the person looking into the camera, framed waist up, and well-lit.

```
{
  "pictureUrl": "https://example.com/images/avatar.png"
}
```

v1 is fast and produces strong photorealism. It works well for quick iteration, internal testing, and use cases where speed matters more than pixel-perfect accuracy.

v2 — Video or image input

The best v2 results come from a reference video, but v2 also accepts a still image when you don't have video of the person.

Provide a `videoUrl` — a publicly reachable HTTPS URL of a reference video showing the person speaking.

```
{
  "videoUrl": "https://example.com/videos/person-speaking.mp4",
  "version": "v2"
}
```

Generating v2 from a still image

If you don't have a reference video, set `version` to `"v2"` and provide a `pictureUrl` instead. You get the v2 rendering model — better photorealism and cleaner rendering than v1 — without needing to capture and upload video.

```
{
  "pictureUrl": "https://example.com/images/person.png",
  "version": "v2"
}
```

Use the same framing guidance as v1: a 16:9 image with the person looking into the camera, framed waist up, and well-lit. A video still produces the best lip sync and expressions — see below — so prefer `videoUrl` when you have usable footage.

Why a video produces better results

A still image gives the system a single frame to work from — it has to guess how the person's face moves when speaking. A video gives it the real thing. The system learns the person's unique mouth movements, facial micro-expressions, head motion patterns, and how their features shift during natural speech. This means:

- **Lip sync stays accurate** even during fast speech, complex words, and language switching
- **Expressions feel natural** — subtle brow raises, eye movements, and smiles that match the tone of what's being said
- **Facial hair renders cleanly** — beards, mustaches, and stubble that move naturally with the jaw instead of clipping or blurring

When to choose v2

Use v2 when the avatar represents someone your users will interact with directly — customer-facing support agents, executive avatars, brand representatives, sales leads, or any use case where the realism of the avatar affects trust and engagement.

Tips for a great reference video

The quality of the output depends heavily on the quality of the input. For the best v2 results:

- **Duration**: 1–5 minutes of continuous natural speech
- **Content**: The person should speak naturally, not read a script — conversational tone produces better results than stiff delivery
- **Framing**: Head and shoulders, centered, with some space above the head
- **Lighting**: Even, front-facing light. Avoid harsh shadows, backlighting, or overhead fluorescents
- **Background**: Clean and uncluttered. A plain wall or blurred background works best
- **Camera**: Eye-level, stable (tripod or resting on a surface). Avoid handheld footage
- **Audio**: Clear audio helps the system align lip movements — use a quiet room

> v2 generation can take up to several hours. Plan accordingly — create the avatar ahead of time, not right before you need it.

Overview

What tools are, how the two execution flows work, and how to attach tools to an agent

Tools let your agent do things beyond conversation — trigger workflows, look up data, generate content, or interact with external systems during a session.

Execution flows

Every tool runs in one of two execution flows. The flow determines where the tool call is delivered when the agent invokes it. You choose the flow when you create the tool.

Implicit

The tool call is delivered to the connected client. For WebRTC sessions, this means the Web SDK running in the user's browser; for WebSocket sessions, your WebSocket client. Your client receives the tool call, executes the logic, and returns the result.

Explicit

The tool call is forwarded to a URL you specify when creating the tool — either an HTTP endpoint or a WebSocket endpoint on your server. Use this when the tool needs to reach your backend systems — for example, querying a database, calling an internal API, or triggering an action in a third-party system.

> The execution flow is set per tool. You can mix implicit and explicit tools in the same session.

There is a third way to give an agent capability: attach an MCP server. You define nothing and execute nothing — the provider connects to the server, discovers its tools, and runs the calls itself.

Choosing a flow

The tool needs to...	Flow
Act on the page or app the user is looking at — update UI, read local state	`implicit`
Use data that only exists client-side	`implicit`
Reach your backend — a database, an internal API, a third-party system	`explicit`
Keep secrets (API keys, credentials) that must never reach the client	`explicit`
Stay open for the whole session, receive the live conversation, or push updates back	`explicit` with a WebSocket endpoint — see WebSocket Tools

Attaching tools to an agent

Tools are attached when you create an Omniagent. Pass an array of tool IDs in the `functions` field:

```
{
  "companionId": "companion_abc",
  "voiceId": "alloy",
  "functions": ["fn_abc123", "fn_def456"],
  "providerSettings": {}
}
```

You can attach as many tools as you need to a single agent. Tools are reusable — create one once and attach it to as many agents as you like.

Creating Tools

Define a tool — its fields, execution flow, and the prompt that controls when it fires

Tools extend what your agent can do during a session. You define what each tool does, what parameters it accepts, and whether the tool call is delivered to your client (implicit) or to your server (explicit).

> Tools are reusable. Create a tool once, then attach it to as many agents as you need.

Creating a custom tool

Create a tool via `POST /public/functions`:

```
curl -X POST https://companion-api.napster.com/public/functions \
  -H "X-Api-Key: $NAPSTER_API_KEY" \
  -H "Content-Type: application/json" \
  -d '{
    "data": {
      "name": "get_order_status",
      "description": "Look up the current status of a customer order by order ID",
      "parameters": {
        "type": "object",
        "properties": {
          "order_id": {
            "type": "string",
            "description": "The unique order identifier"
          }
        },
        "required": ["order_id"]
      }
    },
    "flow": "explicit",
    "url": "https://your-api.example.com/tools/get-order-status",
    "headers": {
      "x-api-key": "your-internal-api-key"
    },
    "receiveMessages": false,
    "prompt": "Use this tool when the user asks about the status of their order."
  }'
```

Field reference

Field	Required	Description
`data.name`	Yes	The function name the model calls. Use snake_case.
`data.description`	Yes	What the tool does. The model reads this to decide relevance.
`data.parameters`	Yes	JSON Schema for the arguments (`type` , `properties` , `required`).
`flow`	Yes	`"implicit"` (delivered to your client) or `"explicit"` (delivered to your server). See execution flows.
`url`	Explicit only	The HTTP or WebSocket endpoint that receives the call. A `wss://` URL keeps a connection open for the whole session — see Executing Tools.
`headers`	No	Custom headers sent with explicit calls — authentication, routing.
`receiveMessages`	No	WebSocket tools only. Stream the live conversation to your endpoint. Default `false` .
`connectionBehavior`	No	WebSocket tools only. Whether the session waits for your server's socket at start (`waitBeforeStart`) and whether it survives a failed connection (`abortOnFailure`). Both default `false` — see Connection behavior.
`prompt`	No (but important)	Controls when and how the agent calls the tool. See below.

Writing effective tool prompts

The `prompt` field on a tool definition is the most important lever you have for controlling when and how the agent calls a tool. At runtime, the contents of `prompt` are injected into the system instruction alongside the tool's function declaration:

```
## tool_name(param1, param2)
[contents of the prompt field]
```

The tool name and parameters are generated automatically from your tool definition. The `prompt` field is your space to control everything else — when the tool fires, when it doesn't, and what the agent says (or doesn't say) to the user.

Structure of a good prompt

Every `prompt` should cover three things:

1. **When to use the tool** — what user intent or conversation state should trigger it.
2. **When NOT to use the tool** — what situations look similar but should not trigger it.
3. **How to behave when calling it** — should the agent announce it, ask for confirmation, or call it silently?

A well-written prompt:

```
Use when the user asks to book, reserve, or schedule a table.
Do NOT use when the user is asking about menu items, hours, or general restaurant info.
Before calling, confirm the reservation details (name, date, time, party size) with the user
and wait for explicit confirmation.
```

A poorly-written prompt:

```
Books a restaurant reservation.
```

The second version gives the agent no guidance on invocation timing, edge cases, or user interaction. It will misfire.

Invocation conditions

Write explicit positive and negative conditions using "Use when" and "Do NOT use when" to control when the agent reaches for a tool.

Positive conditions describe the user intent that should trigger the tool. Be specific — use concrete language the user might say:

- Good: `Use when the user asks to book, reserve, or schedule a table.`
- Bad: `Use when relevant.`

Negative conditions describe situations that look similar but should not trigger the tool. These matter more than most developers expect — models often call tools when the intent is close but not quite right:

- Good: `Do NOT use when the user is asking about menu items, restaurant hours, or general info.`
- Bad: *(omitted entirely)*

Sequencing conditions control the order in which tools fire. Use "only after" to require a prerequisite and "only if" to add conditional logic:

- `Only call after the user has provided their name, date, and time.`
- `Only call if check_availability was called first and returned available slots.`
- `After calling this tool, you can call confirm_booking or cancel_booking.`

This prevents the agent from calling tools out of order or with missing information. If a tool requires parameters the user hasn't provided yet, instruct the agent to ask for them first.

```
"prompt": "Use when the user wants to book an appointment. Before calling, make sure the
user has provided a preferred date, time, and service type — ask for any missing details.
Only call after all three are confirmed. Do NOT use for cancellations."
```

Behavior patterns

You control what the agent says to the user around a tool call through the `prompt` field. There are three patterns, and you can mix them across tools in the same session.

Proactive (silent)

The agent calls the tool without announcing it or asking permission.

Best for read-only or low-stakes operations: looking up context, logging events, enriching data, tagging conversations. As a rule of thumb, if the tool only reads data and has no side effects, proactive is a safe default.

```
{
  "data": {
    "name": "log_mood",
    "description": "Logs the user's current mood for session analytics"
  },
  "prompt": "Call automatically whenever the user expresses a clear emotion (happy,
frustrated, anxious, etc.). Do not announce it to the user. Do NOT call when the user is
asking a factual question without emotional content.",
  "flow": "explicit",
  "url": "https://your-api.example.com/tools/log-mood"
}
```

Preamble (brief acknowledgment)

The agent says a short bridging phrase before or during the tool call so the user doesn't hear silence. This is the recommended default for most tools.

```
{
  "data": {
    "name": "check_order_status",
    "description": "Looks up the status of a customer order by order ID"
  },
  "prompt": "Use when the user asks about their order, delivery, or shipment status. Before
calling, briefly let the user know you're looking it up (e.g., 'Let me check that for you').
Do NOT use when the user is asking about products, pricing, or returns.",
  "flow": "explicit",
  "url": "https://your-api.example.com/tools/order-status"
}
```

If you want a specific phrase instead of letting the agent generate one:

```
"prompt": "Use when the user asks about their order status. Before calling, say: 'One moment
while I pull up your order.' Do NOT use for returns or cancellations."
```

Keep bridging phrases neutral — they should not imply success or failure before the tool has returned. Good examples: "Let me check that," "One moment," "Let me look into that." Bad examples: "Great news, let me confirm that," "I found your order, one second."

You can also guide what the agent says after the tool returns. This helps the agent transition naturally back into the conversation:

```
"prompt": "Use when the user asks about their order. Before calling, say 'Let me check on that.' After receiving the result, start with a brief opener like 'Here's what I found' and then share the details."
```

Confirmation (ask first)

The agent describes what it's about to do and waits for the user to say yes before calling the tool.

Best for write operations and high-stakes actions: anything that costs money, sends a message, makes a booking, changes account settings, or can't be easily undone. As a rule of thumb, if the tool changes state or has side effects, require confirmation.

```
{
  "data": {
    "name": "book_appointment",
    "description": "Books an appointment for the user"
  },
  "prompt": "Use when the user wants to schedule an appointment. Before calling, describe the appointment details (date, time) to the user and wait for explicit confirmation before proceeding. Do NOT call without confirmation. Do NOT use for cancellations or rescheduling.",
  "flow": "explicit",
  "url": "https://your-api.example.com/tools/book-appointment"
}
```

Mixing patterns across tools

A single session can include tools with different behavior patterns. Control the behavior per tool through each tool's `prompt` field:

```
get_weather (proactive)
→ "Call automatically when the user mentions weather, outdoor plans, or travel. Do not announce it."
search_products (preamble)
→ "Use when the user asks about products or wants recommendations. Before calling, say something like 'Let me find that for you.' Do NOT use for order status or returns."
process_payment (confirmation)
→ "Use when the user is ready to pay. Before calling, confirm the total amount and payment method with the user. Do NOT call without explicit confirmation. Do NOT use for refunds."
```

Voice-specific considerations

Tool calls in voice conversations behave differently than in text chat. Keep these in mind when writing prompts:

- **Silence is bad.** In text, the user sees a loading indicator. In voice, they hear nothing — and assume the agent froze. If a tool takes more than a moment, use the preamble pattern so the user knows something is happening.
- **Fewer tools is better.** Each tool adds tokens to the system instruction. More tools also increases the chance the agent calls the wrong one. Keep the set focused.
- **Keep prompts short.** Every token in `prompt` counts against the instruction budget. Use short, direct sentences. Avoid paragraphs.
- **Don't duplicate the description.** The `description` field already tells the agent what the tool does. Use `prompt` for when, when not, and how — not to repeat the description.

> If `prompt` is left empty, the agent relies entirely on the tool's `description` to decide when to call it. This works for simple tools but gives you no control over invocation timing, negative conditions, or call behavior.

Executing Tools

Handle tool calls at runtime — implicit calls in your client, explicit calls on your server, and what to do about slow work

When the agent invokes a tool mid-conversation, the call is delivered to you and your code returns the result. Where it's delivered depends on the tool's **execution flow**: **implicit** calls go to your connected client, **explicit** calls go to a URL on your server.

Implicit tool calls

When an implicit tool is invoked during a session, the Napster API sends an event to your connected client with the tool call details. Your client handles the logic and sends the result back. This works over both WebRTC (via the Web SDK) and WebSocket connections.

Receiving a tool call

When the agent invokes an implicit tool, the server delivers a `function_implicitly_called` event to your client:

```
{
  "event": "function_implicitly_called",
  "data": {
    "item_id": "item_001",
    "call_id": "call_abc123",
    "name": "get_order_status",
    "arguments": {
      "order_id": "ORD-7890"
    }
  }
}
```

Field	Description
`item_id`	The ID of the conversation item that triggered the tool call
`call_id`	A unique identifier for this specific tool call — use this when sending the response
`name`	The name of the tool being invoked
`arguments`	The parameters the agent populated for the tool call

Sending the tool output

Once you've processed the tool call, send the result back with a `send_function_output` event:

```
{
  "type": "send_function_output",
  "data": {
    "call_id": "call_abc123",
    "output": {
      "status": "shipped",
      "tracking_number": "1Z999AA10123456784"
    },
    "delay": false
  }
}
```

Field	Description
`call_id`	The `call_id` from the `function_implicitly_called` event
`output`	The result of the tool call — this is passed back to the agent
`delay`	When `true`, the response is held until the agent finishes speaking before being processed

Timeout

Tool calls have a default timeout of **10 seconds**. If no response is received within that window, the Napster API sends `"Failed to fetch information"` to the model as the tool result and delivers a `function_call_timeout` event to your client.

```
{
  "event": "function_call_timeout",
  "data": {
    "call_id": "call_abc123",
    "function_name": "get_order_status"
  }
}
```

Match `call_id` to the pending call so your client can stop waiting on it. A result sent for that call after the timeout doesn't reach the agent.

For work that can't finish inside the window, respond with an **interim result** right away (so the agent can tell the user the result is on its way) and deliver the real outcome afterwards. The rule: **the late outcome is delivered by whoever handled the tool call, over the connection it already holds.** How that looks depends on the kind of tool — not on the session channel:

Tool	Who handled the call	Delivering the late outcome
Implicit	Your client	The client injects it with a `send_message` command — see the example below.
Explicit, WebSocket URL	Your functions server	The server sends `context_update` on the tool connection. Works whether the session channel is WebRTC or WebSocket.
Explicit, HTTP URL	Your server, per-request	No persistent connection to push through — the work must fit inside the timeout. If it can't, switch the tool to a WebSocket URL.

Deferred results from an implicit tool

Your client handled the call, so it also delivers the outcome — using the session connection it already holds. First answer the tool call with an interim result so the agent can say the work has started; later, inject the real outcome as a system message with `trigger_response: true` so the agent announces it.

On **WebRTC**, both go through the Web SDK instance:

```
// 1. Beat the timeout — answer the tool call with an interim result
instance.sendCommand({
  type: "send_function_output",
  data: {
    call_id,
    output: { status: "working", message: "Generating the report now — it'll be ready
shortly." },
  },
});

// 2. Later, when the real work finishes — inject the outcome
instance.sendCommand({
  type: "send_message",
  data: {
    role: "system",
    text: "The report is ready: 1,204 rows. Tell the user and offer to walk through it.",
    trigger_response: true,
  },
});
```

On a **WebSocket** session channel, the messages are identical — your client just writes them to the socket:

```
// 1. Interim result
ws.send(JSON.stringify({
  type: "send_function_output",
  data: { call_id, output: { status: "working", message: "Generating the report now." } },
}));

// 2. Later — the real outcome
ws.send(JSON.stringify({
  type: "send_message",
  data: { role: "system", text: "The report is ready: 1,204 rows.", trigger_response: true
},
}));
```

Explicit tool calls

When a tool's `flow` is `"explicit"`, the tool call is forwarded to the `url` you configured when creating it — either an HTTP endpoint or a WebSocket endpoint on your server, with any custom `headers` you set.

The flow is the same regardless of protocol:

1. The LLM decides to call the tool based on the conversation and populates the parameters.
2. The Napster API delivers the tool call to the `url` you configured, including the tool name and populated arguments.
3. Your server executes the logic — for example, querying a database, calling an internal API, or triggering an action in a third-party system — and returns the result.
4. The Napster API passes the result back to the LLM, which uses it to continue the conversation with the user.

Using an HTTP endpoint:

```
{
  "flow": "explicit",
  "url": "https://your-api.example.com/tools/get-order-status",
  "headers": {
    "x-api-key": "your-internal-api-key"
  }
}
```

With an HTTP URL, the platform makes a request to your endpoint per tool call. Simple to set up — but there's no connection outside of tool calls.

Using a WebSocket endpoint:

```
{
  "flow": "explicit",
  "url": "wss://your-api.example.com/tools/get-order-status",
  "headers": {
    "x-api-key": "your-internal-api-key"
  }
}
```

With a WebSocket URL, the platform connects to your server **at session start** and keeps the connection open for the entire session. Tool calls skip the connection overhead — and the open connection turns your tool into much more than a request handler. It lets your server:

- **Know who it's talking to before the first call** — receive the session ID and your own user/tenant context the moment the session starts.
- **Follow the conversation live** — stream the full user↔agent exchange in real time, so your tool answers with complete context.
- **Drive the user's screen** — push UI state to the client (confirmation cards, progress, page updates) in sync with the conversation.
- **Deliver slow results after the timeout** — return an interim answer now and inject the real outcome into the conversation whenever it's ready.

WebSocket Tools

The session-long connection between the platform and your functions server — initialize, metadata, the live conversation feed, and pushing messages back

An explicit tool with a `wss://` URL gets more than fast tool calls: the platform opens a WebSocket **to your functions server** when the session starts and keeps it open for the whole session. This page covers everything that connection can do beyond answering tool calls.

The `initialize` message

When the WebSocket connection opens at the start of a session, the platform sends an `initialize` message. It identifies the session, lists the WebSocket-based tools attached to it, and carries the session's metadata — so your server can set up any per-session state before the first tool call arrives.

```
{
  "type": "initialize",
  "data": {
    "session_id": "sess_xyz789",
    "functions": [
      {
        "name": "get_order_status",
        "flow": "explicit"
      }
    ],
    "metadata": {
      "tags": {
        "key": "value"
      },
      "external_client_id": "client_id",
      "external_client_profile": {
        "key": "value"
      }
    }
  }
}
```

Session metadata

The `metadata` object is always included in the `initialize` message. It carries the context you passed when you created the connection — so your server can tie the session to your own user, tenant, or profile without a separate lookup.

The three values come straight from the connection-creation request. Note the WebSocket message uses `snake_case`, while the connection request uses `camelCase`:

`metadata` field (WebSocket)	Connection request field
`tags`	`tags`
`external_client_id`	`externalClientId`
`external_client_profile`	`externalClientProfile`

> Fields you did not set at connection creation are simply absent from `metadata` — no per-tool opt-in is required.

Connection behavior

The platform opens the socket to your server at session start — and your server might be slow to accept, or down. The tool's `connectionBehavior` setting controls what the *session* does in each case. Set it when you create the tool (`POST /public/functions`), or update an existing tool with `PUT /public/functions/{functionId}`:

```
{
  "connectionBehavior": {
    "waitBeforeStart": false,
    "abortOnFailure": false
  }
}
```

Flag	`true`	`false` **(default)**
`waitBeforeStart`	The session waits for your socket to be fully connected before starting.	The session starts after a short wait and lets your tool join late.
`abortOnFailure`	The whole session is aborted if your socket fails to connect.	The session starts anyway and the client is notified of the failure.

Use `waitBeforeStart: true` when the agent's first responses depend on your tool being reachable; use `abortOnFailure: true` when a session without your tool is worthless. For everything else, the defaults favor fast session starts.

> ⚠️ The defaults changed. Previously the platform always behaved as if both flags were `true` — every session waited for your socket and died if it failed to connect. Sessions now start without waiting and survive a failed tool connection unless you opt back in with `{ "waitBeforeStart": true, "abortOnFailure": true }`.

To see how each tool's connection actually went — connected when, failed why, canceled by the user leaving — read the session's `functionStates`.

`receiveMessages`

Set `receiveMessages` to `true` on a tool and the conversation between the user and the agent is streamed to your WebSocket endpoint in real time as it happens — not just when the tool is called.

Because the WebSocket connection is open from the start of the session, enabling `receiveMessages` gives your server a live feed of the full exchange. By the time the agent invokes your tool, your server already has the complete conversation context and can provide a more informed response.

This is particularly useful for tools that need to understand the broader conversation — not just the specific parameters the agent populated for the tool call.

Pushing UI state to the client (`ui_update`)

Sometimes the conversation changes something the **page** should reflect, and the change happens on *your* backend — where the client can't see it. For example:

- The user books a table through the agent — your tool handler completes the booking, and the page should show a **confirmation card** with the details, not just have the agent say them out loud.
- A tool kicks off a long-running job — a report, an export, a search — and you want the page to show **live progress** while the agent keeps talking.
- The agent updates the user's cart, profile, or dashboard through a tool — the **on-screen state** should update the moment it happens.

`ui_update` is the channel for exactly this: your server pushes a message **to the end user's client** through the tool connection, and the platform relays it over whichever session channel is in use (the WebRTC data channel, or the client WebSocket). The payload is not validated or modified — it arrives exactly as you sent it, and your client renders it however it wants.

Send from your functions server — for example from an explicit tool handler after completing an action:

```
{
  "type": "ui_update",
  "data": { "view": "booking-confirmation", "bookingId": "bk_123" }
}
```

The client receives it as a server event:

```
{
  "event": "ui_update",
  "data": { "view": "booking-confirmation", "bookingId": "bk_123" }
}
```

`data` can be any JSON, and the message is **not tied to a tool call** — send it whenever you want while your WebSocket is connected.

⚠️ `ui_update` is one-way and fire-and-forget: no acknowledgement, and updates are not stored — a client that connects mid-session won't receive earlier updates.

Updating the conversation context (`context_update`)

`context_update` lets your functions server add information to the **conversation itself**. Where `ui_update` targets the page, `context_update` is injected into the session's context — the agent reads it as if it were part of the conversation, and can act on it or respond to it. It's the server-side counterpart of the client's `send_message` command, and like everything on this page, it works regardless of the session channel — WebRTC included.

The canonical use case is the **deferred tool result**. Tool calls have a 10-second timeout — but some tools do real work that takes longer: generating a report, calling a slow third-party API, waiting on a human approval. You can't keep the agent (and the user) hanging while it runs:

1. The agent calls your tool. Your handler returns an interim result right away — for example `"Started. Tell the user the result will be ready shortly."` — so the conversation isn't blocked.
2. Your server does the slow work.
3. When it completes, your server sends `context_update` with the outcome — the agent picks it up and tells the user.

```
{
  "type": "context_update",
  "data": {
    "previous_item_id": null,
    "role": "system",
    "trigger_response": false,
    "content": "The export the user asked for has finished. 1,204 rows, download link sent
by email."
  }
}
```

Field	Type	Description
`content`	`string`	The text to inject into the conversation context.
`role`	`string`	Who the injected message is from, e.g. `"system"`.
`trigger_response`	`boolean`	`true` — the agent responds to the injection immediately. `false` — the context is added silently and informs the agent from its next turn.
`previous_item_id`	`string | null`	Thread position for the injected message; `null` appends at the end of the conversation.

For a deferred result you'll usually want `trigger_response: true`, so the agent announces the outcome the moment it lands; with `false` the agent simply knows, and works it into the conversation naturally when the user next speaks.

`context_update` isn't limited to tool results, though — it can be sent at any time while the WebSocket is connected and doesn't have to follow a tool call. Use it whenever your backend knows something the agent should too: fresh instructions, changed account state, an event that happened outside the conversation.

Servers

Connect remote MCP servers so your Omniagent can call their tools during a session

An MCP server is an external service that exposes tools over the Model Context Protocol. Register one with the Napster API, attach it to an Omniagent, and the agent can call its tools mid-conversation — without you writing a tool definition or handling a single tool call.

⚠ MCP servers work with Realtime API keys — Azure OpenAI and OpenAI. Azure OpenAI is the default, so this covers most projects. With any other key (Cascade or Other), opening a session with MCP servers attached fails with `400 UnsupportedProvider`.

⚠ MCP is not available on phone channels. A server attached to an agent does not apply on VoIP or SIP sessions, so the agent has no MCP tools on a call.

How MCP servers differ from tools

A custom tool is something you define and execute — you write the JSON Schema, and the call comes back to your client or your server to run. An MCP server is something you *point at*. The provider connects to it, discovers its tools, and executes the calls itself.

	Custom tools	MCP servers
You define the schema	Yes	No — discovered from the server
You execute the call	Yes, in your client or on your server	No — executed provider-side
Attached via	`functions`	`mcp`
Tools per entry	One	Every tool the server exposes

The practical consequence: attaching an MCP server can add many tools to a session at once. Use `allowedTools` to narrow that down.

> MCP server IDs share a namespace with tool names. If a server and a tool have the same name, the session fails to start.

Types of MCP server

Most MCP servers you attach are ones somebody else built and runs — a vendor's server for their own product, or a public one. What differs between them is **what the server requires before it will let your agent in.** That determines how you register it, and how much your session code has to do.

Type	What the server requires	Register with
Public	Nothing	`url`
Authenticated	A credential you hold	`url` + `headers`
Per-user	Each end user's own authorization	`url` + `authorizationRequired: true`

The server's own documentation tells you which it is. If it gave you an API key, it is authenticated. If it sent you through a sign-in screen, it is per-user.

For Gmail, Google Drive, SharePoint and other common services, you don't need to register anything at all — attach a ready-made connector instead. Connectors require the OpenAI provider, so on Azure OpenAI register the service's own MCP server here as a per-user server.

Public

Open to anyone with the URL — documentation lookups, public reference data, search. Register it with a `url` and attach it. There is nothing to handle at session time.

Authenticated

The server sits behind an account you hold. You get an API key or token from whoever runs it, store it in `headers` when you register, and it is sent on every request.

Every session uses that one account, so the agent sees the same data no matter who is talking to it. That suits servers whose data is not tied to a person: catalogs, pricing, documentation, reference systems.

> Use `allowedTools` to limit what the agent can reach through your account. A third-party server often exposes more tools than you need — including ones that write or delete — and by default the agent can call all of them.

Per-user

The server holds data that belongs to individual people — their mail, calendar, files, or records — so it requires each user to authorize access to their own. Register it with `authorizationRequired: true` and no credential, then pass that user's OAuth token when you open the session. The token is used for that session and never stored.

This is the only shape that can answer "what's on *my* calendar" correctly. It costs you more work: you run the sign-in flow and supply a token per session. See Per-user authorization.

Register a server

Registration stores the server's address and access rules against your project. It does not attach it to anything yet.

Send the server definition to `POST /public/mcp-servers`:

```
curl -X POST https://companion-api.napster.com/public/mcp-servers \
  -H "X-Api-Key: $NAPSTER_API_KEY" \
  -H "Content-Type: application/json" \
  -d '{
    "id": "inventory",
    "url": "https://mcp.acme.com/sse",
    "headers": {
      "Authorization": "Bearer sk_live_abc123"
    },
    "allowedTools": ["check_stock", "get_price"],
    "requireApproval": "never",
    "description": "Live product inventory and pricing",
    "authorizationRequired": false,
    "tags": {
      "env": "prod"
    }
  }'
```

The response is the stored server definition — the same fields you sent, plus any defaults the API filled in. There is no generated ID to capture, since `id` is the one you chose:

```
{
  "id": "inventory",
  "url": "https://mcp.acme.com/sse",
  "headers": {
    "Authorization": "Bearer sk_live_abc123"
  },
  "allowedTools": ["check_stock", "get_price"],
  "requireApproval": "never",
  "description": "Live product inventory and pricing",
  "authorizationRequired": false,
  "tags": {
    "env": "prod"
  }
}
```

Fields

Parameter	Type	Required	Description
`id`	string	Yes	You choose it — it is not generated. Alphanumeric, underscores and hyphens only, max 48 characters. Must be unique within your project
`url`	string	Yes	The MCP server's endpoint. Must be `https://`
`headers`	object	No	String-to-string map sent to the server on every request. Use this for a static credential
`allowedTools`	array	No	Names of the tools the agent may call. Omit to expose every tool the server advertises
`requireApproval`	string	No	`never` (default) or `always`. See Approval
`description`	string	No	Stored for your own reference. It is not sent to the model and does not influence tool selection
`authorizationRequired`	boolean	No	`false` by default. `true` means sessions must supply a per-user token. See Per-user authorization
`tags`	object	No	String-to-string map, max 16 pairs. Keys and values up to 64 characters. Filters the list endpoint

> ⚠ The `id` becomes the server's label everywhere the agent refers to it, and it cannot be changed. Renaming means deleting the server, recreating it, and re-attaching it wherever it was used.

Approval

`requireApproval` controls whether a tool call runs immediately or waits for a human decision.

With `never` — the default — the agent calls the server's tools as soon as it decides to.

With `always`, every call pauses and your client decides. The exchange runs over the session's data channel:

1. You receive an `mcp_approval_request` event naming the server, the tool and its arguments.
2. Present the choice to the user however you like.
3. Answer with `send_mcp_approval`, passing the request ID and `approve: true` or `false`.

A rejection cancels the call and the MCP server is never contacted.

> ⚠ You have **15 seconds** to answer. After that the request is rejected automatically and the call never runs. Build the approval UI before you set `always` on a server your agent depends on — the agent gives no spoken indication that it is waiting.

Use `always` for tools with consequences — sending mail, issuing a refund, changing a record — and `never` for reads.

Attach it to an agent

A registered server does nothing until you attach it. Pass its ID in `mcp.servers` when you create or update an Omniagent:

```
curl -X PATCH https://companion-api.napster.com/public/agents/agent_abc123 \
  -H "X-Api-Key: $NAPSTER_API_KEY" \
  -H "Content-Type: application/json" \
  -d '{
    "mcp": {
      "servers": ["inventory"]
    }
  }'
```

Every session that agent runs now has the server attached. You can also set `mcp` on a channel configuration to override it for one channel, or on a raw connection to attach servers per session.

The `mcp` object holds a second list, `connectors`, for ready-made servers you don't register. Those require the OpenAI provider — see Connectors.

> ⚠ `mcp` is replaced wholesale, never merged. A channel configuration with its own `mcp` overrides the agent's completely, and a `PATCH` with `mcp` replaces the previous value. To add a server, send the full list including the ones already attached.

Per-user authorization

This is the mechanics of the per-user shape — two steps: register the server as needing a token, then supply the user's token per session.

Register it with `authorizationRequired` and no credential:

```
curl -X POST https://companion-api.napster.com/public/mcp-servers \
  -H "X-Api-Key: $NAPSTER_API_KEY" \
  -H "Content-Type: application/json" \
  -d '{
    "id": "crm",
    "url": "https://mcp.acme.com/crm",
    "authorizationRequired": true,
    "description": "Customer records, scoped to the signed-in user"
  }'
```

Attach it to the agent, then pass that user's token in `mcp.authorizations` when you open the session. The agent already carries the server, so the connection call sends only the token:

```
curl -X POST https://companion-api.napster.com/public/agents/agent_abc123/connections \
  -H "X-Api-Key: $NAPSTER_API_KEY" \
  -H "Content-Type: application/json" \
  -d '{
    "channelType": "webrtc",
    "mcp": {
      "authorizations": [
        {
          "mcpServerId": "crm",
          "token": "ya29.a0AfH6..."
        }
      ]
    }
  }'
```

The token is used for that session only. It is never stored.

> Running the OAuth flow is your responsibility. Napster never contacts the identity provider — you obtain the user's access token and pass it at connection time. Mint a fresh token for each session rather than reusing a cached one. If a session outlives the token, refresh it mid-session.

The rule is strict both ways

`authorizationRequired` decides whether a token is mandatory or forbidden — there is no middle ground:

Server registered with	Session provides a token	Result
`authorizationRequired: true`	Yes	Works
`authorizationRequired: true`	No	`400` — the server requires a token
`authorizationRequired: false`	Yes	`400` — the server does not accept a token
`authorizationRequired: false`	No	Works, using `headers`

A token for a server that is not attached to the same request is also rejected.

Refreshing a token mid-session

Access tokens expire, and a conversation can outlive one. To swap in a fresh token without ending the session, send `set_settings` with `mcp.authorizations` from your client:

```
{
  "type": "set_settings",
  "data": {
    "mcp": {
      "authorizations": [
        {
          "mcp_server_id": "crm",
          "token": "ya29.a0AfH6_refreshed..."
        }
      ]
    }
  }
}
```

The agent uses the new token from its next call to that server.

- The list replaces every token for the session. If several attached servers need a token, send all of them each time.
- The strict rule still applies: every server registered with `authorizationRequired: true` needs a token, and no other server can take one.
- If the update breaks a rule, nothing changes and your client receives an `error` event whose `message` explains why.
- Only tokens can change mid-session. You can't attach or detach servers.

> Fields sent over the session's data channel use snake_case: `mcp_server_id` here, where the REST request that opens the session uses `mcpServerId`.

Manage registered servers

```
# List, with optional search and tag filters
curl "https://companion-api.napster.com/public/mcp-servers?pageSize=20&search=inventory" \
  -H "X-Api-Key: $NAPSTER_API_KEY"

# Get one
curl https://companion-api.napster.com/public/mcp-servers/inventory \
  -H "X-Api-Key: $NAPSTER_API_KEY"

# Delete
curl -X DELETE https://companion-api.napster.com/public/mcp-servers/inventory \
  -H "X-Api-Key: $NAPSTER_API_KEY"
```

Updates use `PUT` and replace the entire server:

```
curl -X PUT https://companion-api.napster.com/public/mcp-servers/inventory \
  -H "X-Api-Key: $NAPSTER_API_KEY" \
  -H "Content-Type: application/json" \
  -d '{
    "url": "https://mcp.acme.com/sse",
    "headers": {
      "Authorization": "Bearer sk_live_xyz789"
    },
    "allowedTools": ["check_stock", "get_price"],
    "requireApproval": "never",
    "description": "Live product inventory and pricing"
  }'
```

> ⚠️ `PUT` is a full replace, not a patch. Any field you omit reverts to its default — `requireApproval` falls back to `never` and `headers` is cleared. Read the server first, change what you need, and send the whole object back.

Errors

Managing servers

Endpoint	Status	Code	Cause
`POST /public/mcp-servers`	`409`	`McpServerAlreadyExists`	An MCP server with that `id` already exists in your project
`POST /public/mcp-servers`	`400`	`id` or `url`	Validation error — `id` is missing or malformed, or `url` is missing or not `https://`
`POST /public/mcp-servers`	`400`	`McpServerValidationFailed`	The platform rejected the server definition. Check the details and try again
`PUT /public/mcp-servers/{id}`	`400`	`McpServerNotFound`	No server with that `id` in your project
`DELETE /public/mcp-servers/{id}`	`400`	`McpServerInUse`	Still attached to an Omniagent or a channel configuration. Detach it everywhere first
`DELETE /public/mcp-servers/{id}`	`400`	`McpServerNotFound`	No server with that `id` in your project

Attaching servers

Endpoint	Status	Code	Cause
`POST /public/agents`, `PATCH /public/agents/{id}`, `PUT /public/agents/{id}/channels/{type}`	`400`	`McpServerNotFound`	An ID in `mcp.servers` or `mcp.connectors` does not exist

Opening a session

These apply to every connection endpoint: `POST /public/agents/{id}/connections`, `POST /public/connections`, and `POST /public/ws-connections`.

Endpoint	Status	Code	Cause
All connection endpoints	`400`	`UnsupportedProvider`	The Omniagent runs on a provider that does not support MCP servers
All connection endpoints	`400`	`UnsupportedProvider`	Connectors are attached, but the Omniagent runs on Azure OpenAI. See Connectors
All connection endpoints	`400`	`McpAuthorizationRequired`	A server with `authorizationRequired: true`, or a connector, has no token in `mcp.authorizations`
All connection endpoints	`400`	`McpServerNotFound`	An attached ID no longer exists, or a token was supplied for a server that is not attached

A session also fails to start if two attached servers share a name, or if a server's name collides with one of the agent's tool names.

Connectors

Attach a ready-made MCP server for Gmail, Google Drive, SharePoint and other common services

A connector is a ready-made MCP server for a common service — Gmail, Google Drive, SharePoint, Dropbox. The provider builds and maintains it, so there is nothing for you to host or register. Attach it to your Omniagent, supply the signed-in user's access token, and the agent can search their mail or read their calendar during the conversation.

> ⚠ Connectors require the **OpenAI** provider. They are not available on Azure OpenAI — which is the default — because connectors are hosted by OpenAI and reached through their API directly. If your Omniagent runs on Azure OpenAI, attach the service's own MCP server instead: see the alternative.
>
> This is stricter than MCP servers, which work on both providers.

Connectors vs. registered servers

Both end up as tools the agent can call. The difference is whether you have to register the server yourself.

	Connector	Registered server
Who maintains it	The provider	Whoever runs the server
You register it	No	Yes, with `POST /public/mcp-servers`
Attached via	`mcp.connectors`	`mcp.servers`
Authentication	Always the end user's token	None, a credential you hold, or the end user's
Tools available	Fixed set, per service	Whatever the server exposes

Every connector acts as the signed-in user, never as your application. That is the point of them — the agent answers "what's on my calendar this afternoon" with *that user's* calendar.

If you are on Azure OpenAI

Connectors are unavailable, but the same services are still reachable — most publish their own MCP server, which you register like any other and attach with `mcp.servers`:

Service	MCP server
Google Calendar	`https://calendarmcp.googleapis.com/mcp/v1`
Gmail	`https://gmailmcp.googleapis.com/mcp/v1`
Google Drive	`https://drivemcp.googleapis.com/mcp/v1`

Register one with `authorizationRequired: true` and supply the user's OAuth token per session, exactly as described under per-user authorization. You run the same consent flow either way — only the field the ID goes in changes.

Available connectors

Service	ID	What the agent can reach
Dropbox	`dropbox`	Files, file contents, recent activity
Gmail	`gmail`	Messages, search, recent mail
Google Calendar	`googlecalendar`	Events and event details
Google Drive	`googledrive`	Documents, drives, recent files
Microsoft Teams	`microsoftteams`	Messages, chat members
Outlook Calendar	`outlookcalendar`	Events, individual and batched
Outlook Email	`outlookemail`	Messages, search, recent mail
SharePoint	`sharepoint`	Messages, sites, documents, recent activity

The ID is what you put in `mcp.connectors`.

> Connectors are not returned by `GET /public/mcp-servers` — that endpoint lists only the servers you registered yourself.

Attach a connector

Connectors go in `mcp.connectors`, alongside any servers you registered in `mcp.servers`:

```
curl -X PATCH https://companion-api.napster.com/public/agents/agent_abc123 \
  -H "X-Api-Key: $NAPSTER_API_KEY" \
  -H "Content-Type: application/json" \
  -d '{
    "mcp": {
      "servers": ["inventory"],
      "connectors": ["gmail"]
    }
  }'
```

Then pass the user's access token when you open the session, keyed by the connector's ID:

```
curl -X POST https://companion-api.napster.com/public/agents/agent_abc123/connections \
  -H "X-Api-Key: $NAPSTER_API_KEY" \
  -H "Content-Type: application/json" \
  -d '{
    "channelType": "webrtc",
    "mcp": {
      "authorizations": [
        {
          "mcpServerId": "gmail",
          "token": "ya29.a0AfH6..."
        }
      ]
    }
  }'
```

A connector always requires a token. Opening a session without an entry for it in `mcp.authorizations` returns `400 McpAuthorizationRequired`. On an Azure OpenAI key, a session with connectors attached returns `400 UnsupportedProvider`.

Access tokens and scopes

You run the OAuth flow. Napster never contacts Google, Microsoft or Dropbox — you register your own OAuth client with them, take the user through consent, hold the refresh token, and mint an access token per session. The token is used for that session and never stored.

The token has to carry the right **scopes**, or the connector's tools fail even though the session starts normally. Scopes are granted at consent time, so a missing one means sending the user back through the flow.

Scopes are defined by the service that issues the token — Google, Microsoft or Dropbox — not by Napster or by the connector. To find out what to ask for, see OpenAI's connectors documentation, which lists the scopes each connector's tools require.

Mint a fresh token per session rather than reusing a cached one. A conversation can outlive a short expiry, and the agent starts failing tool calls when it does — to avoid that, refresh the token mid-session.

Files

Upload documents into knowledge collections so your agent can ground its responses in your content

Knowledge files let you give your agent domain expertise. Instead of relying on what the underlying LLM knows from training data, you upload your own documents — product manuals, policy guides, technical specs, training materials — and the agent retrieves relevant content during conversations.

How it works

Files are organized into **knowledge collections**. A knowledge collection is a container that groups related documents together. When your agent needs to answer a question, it searches the collection and grounds its response in your content.

Each collection has a **provider** that determines how its documents are embedded and retrieved. Use `azureOpenAI` — a collection works with agents on Realtime and Cascade API keys, whichever provider they use.

Create a knowledge collection

```
curl -X POST https://companion-api.napster.com/public/knowledge-bases \
  -H "X-Api-Key: $NAPSTER_API_KEY" \
  -H "Content-Type: application/json" \
  -d '{
    "name": "Product Documentation",
    "provider": "azureOpenAI"
  }'
```

Response:

```
{
  "id": "kb_abc123",
  "name": "Product Documentation",
  "provider": "azureOpenAI",
  "created": 1710000000
}
```

The `provider` field determines which embedding backend is used. Use `azureOpenAI`.

Upload a file

Upload a file to a collection by providing a publicly accessible URL. The platform downloads and processes the file for retrieval.

```
curl -X POST https://companion-api.napster.com/public/knowledge-bases/kb_abc123/files \
  -H "X-Api-Key: $NAPSTER_API_KEY" \
  -H "Content-Type: application/json" \
  -d '{
    "url": "https://example.com/documents/product-catalog.pdf"
  }'
```

Response:

```
{
  "id": "file_xyz789",
  "originalName": "product-catalog.pdf",
  "contentType": "application/pdf",
  "createdAt": "2026-03-10T12:00:00Z"
}
```

Supported file types

Format	Max size
`.doc`	10 MB
`.docx`	50 MB
`.pdf`	50 MB
`.txt`	10 MB
`.md`	10 MB
`.png`	10 MB
`.jpeg`	10 MB

Limits

Each knowledge collection can contain up to **30 files**. Adding a file to a full collection returns `400 KnowledgeBaseFileLimitReached` — delete files you no longer need or merge related documents before uploading.

Keep in mind that each session accepts a single collection, so splitting content across multiple collections does not raise the effective limit for one agent — consolidate instead.

For platform-wide limits, see Limits.

Attaching knowledge to an agent

After creating a collection and uploading files, attach it to your Omniagent by passing the collection ID in the `knowledgeBaseId` field when creating the agent:

```
curl -X POST https://companion-api.napster.com/public/agents \
  -H "X-Api-Key: $NAPSTER_API_KEY" \
  -H "Content-Type: application/json" \
  -d '{
    "companionId": "comp_abc123",
    "name": "Support Agent",
    "voiceId": "alloy",
    "knowledgeBaseId": "kb_abc123",
    "providerSettings": {}
  }'
```

Different agents can use different collections, and you can override the collection per channel using channel overrides.

> Each session accepts a single knowledge collection. If you need your agent to reference content from multiple sources, combine those files into one collection.

File summaries

Each file in a knowledge collection can have a `summary` — a short description of what the file contains. Summaries help the agent understand what each file is about and improve the chances of retrieving the right content when answering questions. A clear, descriptive summary makes it more likely that the agent will draw on that file during relevant conversations.

To update a file's summary:

```
curl -X PATCH https://companion-api.napster.com/public/knowledge-
bases/kb_abc123/files/file_abc123/summary \
  -H "X-Api-Key: $NAPSTER_API_KEY" \
  -H "Content-Type: application/json" \
  -d '{
    "summary": "Product return policy and refund procedures for all regions"
  }'
```

FAQs

Add curated Q&A pairs so your agent gives consistent answers to important questions

Curated FAQs are manually defined question-and-answer pairs for high-stakes or policy-critical topics — pricing, compliance, eligibility, return policies, or anything where you need the agent to give a specific, controlled answer rather than generating one.

How it works

FAQs are organized into **FAQ collections**. Each collection contains Q&A pairs. When a user asks a question that matches one of your FAQs, the agent responds with the answer you defined rather than generating one.

Create an FAQ collection

You can create a collection and populate it with Q&A pairs in a single call:

```
curl -X POST https://companion-api.napster.com/public/faqs \
  -H "X-Api-Key: $NAPSTER_API_KEY" \
  -H "Content-Type: application/json" \
  -d '{
    "name": "Return Policy",
    "faqs": [
      {
        "question": "What is your return policy?",
        "answer": "You can return any item within 30 days of purchase for a full refund.
Items must be in original condition with tags attached."
      },
      {
        "question": "How long do refunds take?",
        "answer": "Refunds are processed within 5-7 business days after we receive the
returned item."
      }
    ]
  }'
```

Response:

```
{
  "id": "faq_abc123",
  "name": "Return Policy",
  "created": 1710000000,
  "items": [
    {
      "id": "faq_item_001",
      "question": "What is your return policy?",
      "answer": "You can return any item within 30 days of purchase for a full refund. Items
must be in original condition with tags attached.",
      "createdAt": 1710000000,
      "updatedAt": null
    },
    {
      "id": "faq_item_002",
      "question": "How long do refunds take?",
      "answer": "Refunds are processed within 5-7 business days after we receive the
returned item.",
      "createdAt": 1710000000,
      "updatedAt": null
    }
  ]
}
```

The `faqs` array is optional — you can create an empty collection and add pairs later.

Each collection can hold up to **50 Q&A pairs**. Creating a collection or adding pairs past that limit returns `400 FaqLimitExceeded`.

Attaching FAQs to an agent

After creating a collection, attach it to your Omniagent by passing the collection ID in the `faqCollections` array when creating the agent:

```
curl -X POST https://companion-api.napster.com/public/agents \
  -H "X-Api-Key: $NAPSTER_API_KEY" \
  -H "Content-Type: application/json" \
  -d '{
    "companionId": "comp_abc123",
    "name": "Support Agent",
    "voiceId": "alloy",
    "faqCollections": ["faq_abc123"],
    "providerSettings": {}
  }'
```

An agent can have one FAQ collection attached. Passing more than one ID in `faqCollections` returns a `400` validation error — to cover more topics, add the Q&A pairs to the same collection. The same limit applies when you override `faqCollections` on a channel config or pass it on a connection. See Limits.

Napster

SECTION 03

Deploy Your Omniagent

Put your agent on the web, in apps, on the phone, and on Napster Station.

Get Started 97

Add an AI Agent to Your Website 98

Session Configuration 102

CHANNELS

Choosing a Channel 106

WebRTC 109

WebSockets 114

VoIP 121

SIP 125

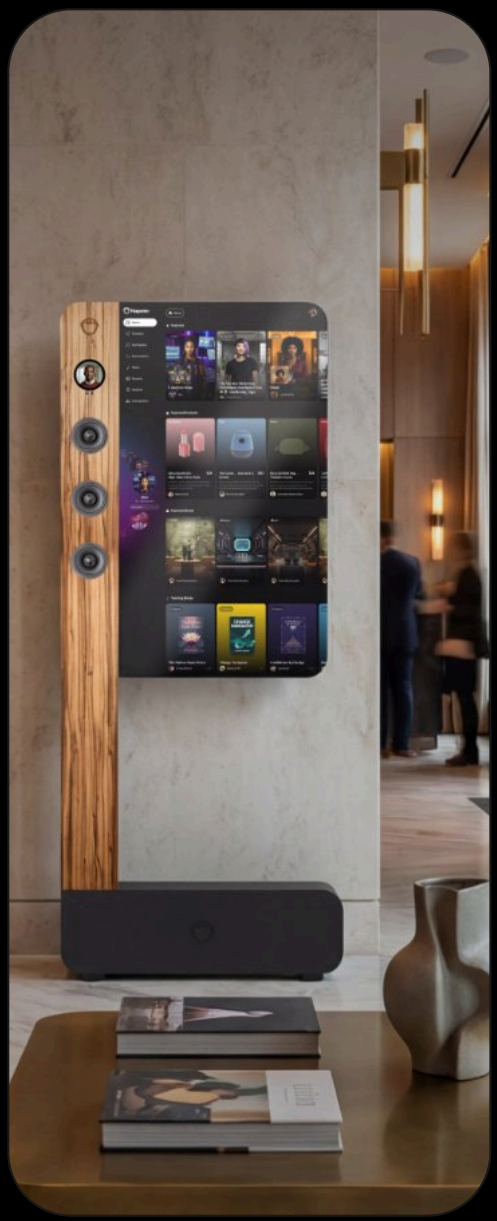
Kiosk 138

MESSAGING

Overview 143

Server Events 144

Client Commands 159



Get Started

Deploy your Omniagent to any channel — web, WebSocket, or phone

Once you've built your Omniagent, you can deploy it to any channel — or all of them at the same time. Because the agent is a single, persistent resource, the same voice, personality, knowledge, and memory are available on every surface. A user can start a conversation on the web and continue it over the phone — it's the same agent.

Pick one channel to start, or deploy to several at once. Each channel guide walks you through the full setup.

Session configuration

Configure per-session settings like memory, user context, and tags.

Messaging

Once connected, your client communicates with the agent through **events** and **commands**. The server pushes events as the conversation progresses, and your client can send text messages and update session settings in real time.

Add an AI Agent to Your Website

The end-to-end journey — agentify your site with Edge MCP, then put the Omniagent on top with the Web SDK



Do both stages with one prompt

Paste it into your AI coding tool — it agentifies the site, then puts the agent on it. Nothing lands without your approval.

You have a website. This guide puts an AI agent on it — one that doesn't just answer questions, but **operates the site on the visitor's behalf**: searches your catalog, fills the cart, prepares the booking, while the page updates in front of them. It takes two stages — first make your site's own operations available to an agent, then put the agent on top:

Stage 1 — Agentify the website

Expose the operations an agent may run and the state it may watch, using Edge MCP — an open integration on the WebMCP standard. Your AI coding tool does the work; you decide what gets exposed.

1 **Open your codebase** — In whatever AI coding tool you already use.

Agentifying a site means adding code to it, so you need access to its codebase. The best way to do the work is with an AI coding tool — Claude Code, Cursor, Codex, or any other Agent Skills-compatible tool — with your project open in it.

2 **Paste the starting prompt** — One prompt installs the Napster skills and kicks off the integration. ▶

The prompt adds the public Napster skills to your tool and walks it through the whole journey — analyze, propose, implement, verify, and keep the surface in sync as your app changes.

3 **It sets itself up** — Installs Edge MCP and proposes tools and live state from your real code. ▶

The skills install the Edge MCP library, analyze your codebase, and propose the whole agent surface: which operations become tools the agent can call (each at the right safety level), and which state becomes live resources the agent watches — the current page, the cart, a form's draft.

4 **Review and approve** — Nothing is written until you say yes. ▶

You, as the site's developer, decide what the agent can do, what it can see, and how each tool is exposed — which ones it may call freely, and which need the user's confirmation first. Nothing lands in your code until you approve it.

5 **Website agentified** — Its actions and live state are now exposed. ▼

Your site's actions and live state are now exposed through the open WebMCP standard, and you can test everything by hand from the browser console or Chrome DevTools. The site is ready for its agent — that's Stage 2.

Learn more about Edge MCP →

⚠ This stage needs access to your site's source code, because agentifying a site means adding code to it — an app you build and own, in whatever stack you use.

If your site is built on a hosted site builder or CMS and you only have the visual editor, you can't agentify it today — we don't ship integrations for those platforms yet. Stage 2 works either way.

Stage 2 — Add the AI agent

Put the Omniagent on the site: face, voice, and a live session. On an agentified page it discovers your tools automatically — no glue code.

1 **Create your Omniagent** — The persona, voice, knowledge — one agent, any channel.

Build the agent once on the Napster API — its persona, voice, tools, and knowledge — and get an agent id. The same Omniagent can then be deployed to any channel; your website is one of them.

2 **Embed with the Web SDK** — Token endpoint + init. Persistence for multi-page sites.

Mint a session token on your backend, then initialize the Web SDK in your frontend — that puts the live agent on the page. On a traditional multi-page site, turn on persistence so one conversation survives navigation instead of dying on every page load.

3 **Talk to your website** — The agent answers — and operates the site as it does.

On an agentified page the Omniagent discovers your tools automatically — no glue code. Visitors ask in their own words, and the agent searches, navigates, fills, and books through the same functions your UI uses, while the page updates in front of them.

Learn more about the Web SDK →

This stage does **not** need source access. If your platform lets you paste a JavaScript snippet into the page — as most site builders and CMSes do — that's enough to run the agent. It just won't be able to operate the site until Stage 1 is done.

Multi-page site? Turn on persistence

On a traditional multi-page site every navigation reloads the page — and kills the session. Persistence keeps one conversation alive as the visitor moves through the site. Single-page apps don't need it — client-side navigation never reloads the page, so the session survives on its own.

```
NapsterCompanionApiSdk.init(token, {
  persistence: { enabled: true },
});
```

Persistence

How the session survives navigation, when the page gets wrapped, break-out rules, and navigating from code.

Read the guide →

The result

A visitor talks to your website. The agent answers with your product knowledge, **and acts**: it searches, navigates, fills, and books — through the same functions your UI uses, as the signed-in user, within exactly the limits you approved.

> **This site is a working example.** These docs are agentified with Edge MCP: the "Ask JC" agent in the corner calls this site's own tools to search the documentation and open pages for you, and it watches a live resource for the page you're on — so it always answers about what's in front of you. Ask it to find something and watch the page move.

Session Configuration

Per-session settings for memory, user context, tags, and channel overrides, and how to end a live session from your backend

When you create a session — whether from an Omniagent or a per-session connection — you can pass additional settings that apply only to that session.

Memory and user identity

Pass an `externalClientId` to enable **memory** for the session. The agent remembers what was said and can pick up the conversation in future sessions with the same user.

```
curl -X POST https://companion-api.napster.com/public/agents/agent_abc123/connections \
  -H "X-Api-Key: $NAPSTER_API_KEY" \
  -H "Content-Type: application/json" \
  -d '{
    "channelType": "webrtc",
    "externalClientId": "user_12345"
  }'
```

Memory is scoped to a **companion + external client ID pair**:

- Same user + different companion = fresh memory
- Different `externalClientId` + same companion = treated as a new person
- Same `externalClientId` + same companion = recalls previous conversations

Use a stable, unique identifier from your system — such as a user ID or account ID.

The value must match `^[A-Za-z0-9_-]{1,32}$`: English letters, digits, underscores, and hyphens only, up to 32 characters. If your internal identifiers don't fit (for example, UUIDs or email addresses), hash or map them to a compact ID before passing them to the API.

User context

The `externalClientProfile` field lets you pass structured information about the end user into the session. The agent can reference this data during the conversation to personalize its responses.

```
curl -X POST https://companion-api.napster.com/public/agents/agent_abc123/connections \
  -H "X-Api-Key: $NAPSTER_API_KEY" \
  -H "Content-Type: application/json" \
  -d '{
    "channelType": "webrtc",
    "externalClientId": "user_12345",
    "externalClientProfile": {
      "name": "Jane",
      "plan": "premium",
      "company": "Acme Corp"
    }
  }'
```

The object is unstructured — include any fields that would help the agent tailor its behavior. For example, passing the user's name lets the agent greet them personally, and passing their subscription tier lets it adjust recommendations accordingly.

> `externalClientId`, `externalClientProfile`, and `tags` are also forwarded automatically to your WebSocket-based tools in the `initialize` message's `metadata` — so your server receives the same session context.

Opening instructions

The `initialSpeech` field gives the agent instructions on how to open the session — how it should start *this* conversation, before the user speaks. It's guidance, not a verbatim line: the agent follows the instruction in its own voice when the connection is established.

```
curl -X POST https://companion-api.napster.com/public/agents/agent_abc123/connections \
  -H "X-Api-Key: $NAPSTER_API_KEY" \
  -H "Content-Type: application/json" \
  -d '{
    "channelType": "webrtc",
    "initialSpeech": "Greet the customer warmly, introduce yourself as Acme Support, and ask
what they need help with."
  }'
```

Because it's passed per connection, you can tailor the opening to the context of each session — for example, referencing what the user was doing when they started the conversation.

`initialSpeech` is available on every connection endpoint: `POST /public/agents/{agentId}/connections`, `POST /public/connections`, and `POST /public/ws-connections`.

Setting a default per channel

To open every session on a channel the same way, set `initialSpeech` on the channel config instead of on each connection. The value applies to all sessions created on that channel unless a connection request overrides it.

```
curl -X PUT https://companion-api.napster.com/public/agents/agent_abc123/channels/webrtc \
  -H "X-Api-Key: $NAPSTER_API_KEY" \
  -H "Content-Type: application/json" \
  -d '{
    "initialSpeech": "Greet the customer warmly, introduce yourself as Acme Support, and ask what they need help with."
  }'
```

Handling invalid configuration

Configuration problems surface at one of two points, depending on what's wrong. This applies to every connection endpoint: `POST /public/agents/{agentId}/connections`, `POST /public/connections`, and `POST /public/ws-connections`.

When you create the connection. The API returns a `400` for problems it can check itself — for example, a missing `voiceId`, a digital twin whose cloned voice isn't ready, or a tool, FAQ collection, or MCP server that doesn't exist. Handle the `400` on your backend: log the message, fix the configuration, and retry.

When the session starts. The AI provider checks the rest after your client connects — whether it recognizes the voice, accepts your credentials, and can handle the instructions' length. The connection request succeeds, and if the provider rejects the

configuration, your client receives a `provider_connection_aborted` event and the session closes. Listen for it so you can report the failure instead of showing a dropped connection.

Green screen video

For WebRTC sessions, you can enable a green-screen background on the companion's video stream so you can composite the avatar over your own UI. This requires matching `useGreenVideo` on the channel config with an SDK init flag — see Green screen background on the WebRTC guide for the full setup.

Ending a session

To close a live session from your backend — for example, when a user's plan runs out mid-conversation or a kiosk needs to be freed — call `DELETE /public/connections/{connectionId}` with the `connection.id` you received when you opened the session:

```
curl -X DELETE https://companion-api.napster.com/public/connections/sess_xyz789 \
  -H "X-Api-Key: $NAPSTER_API_KEY"
```

The API returns `200` and the session ends right away. It then appears in sessions as `closed` with `closeReason: "connection_aborted"`.

This works for `webrtc`, `websocket`, and `kiosk` sessions.

Error	Status	When
`ConnectionNotFound`	`400`	The ID is unknown, or the session has already closed or failed.
`ConnectionAbortNotSupported`	`400`	The session is a `sip` or `voip` call.
`ConnectionAbortFailed`	`400`	The session couldn't be closed. Retry the request.

Choosing a Channel

Pick the right way to connect — by what you want to do, from a video avatar to a text-only chat

Connecting your Omniagent involves two separate choices:

- **Channel** — the transport, or *how* you connect: WebRTC, WebSocket, VoIP, SIP, or Kiosk.
- **Modality** — *what* the conversation is: audio (voice), text, or video.

They're independent axes. The quickest way to decide is to start from what you want to do.

Pick by what you want to do

I want to...	Use	Modality
Embed a talking **video avatar** in a web page	WebRTC (Web SDK)	audio + video
Add **voice** to my own app, backend, or headless client	WebSocket with `modality: "audio"`	audio
Offer a **text-only chat** (typed, no voice)	WebSocket with `modality: "text"`	text
Put my agent on a **phone number** through Twilio	VoIP	voice
Answer **phone calls** over my own SIP trunk	SIP	voice
Run an **in-person, walk-up** device	Kiosk (Napster Station)	voice + experience screen

> It's the same Omniagent across all of them. Deploy to one channel or several at once — the voice, personality, knowledge, and memory stay consistent everywhere.

Audio and video in the browser

If you want users to see and talk to a real-time avatar on a web page, use **WebRTC** through the Web SDK. It ships a ready-made avatar widget with controls and handles the audio + video streaming and session lifecycle for you.

Voice in your own app or backend

If you want voice but don't need a rendered avatar — a server-side integration, a headless client, or your own custom UI — use a WebSocket connection with `modality: "audio"` (the default). It streams audio in both directions in a lightweight protocol you drive yourself.

Text-only chat (no voice)

If you want a typed, text-only conversation — a chatbot with no audio or avatar — use a WebSocket connection with `modality: "text"`. The user's messages go in and the agent's text replies come back over the same connection; there's no audio streaming.

Text is a **modality**, not a separate channel — it runs over WebSocket. See WebSockets — Text sessions for the setup and message shapes.

Phone calls

To put your agent on a phone number, choose based on your telephony setup:

- VoIP — you bring a Twilio account (currently the only supported VoIP provider) and a phone number; Napster gives you a webhook URL to register as the number's incoming-call handler, and your agent answers. Simpler than SIP — no trunk credentials or SIP registration — but the account and number are still yours.
- SIP — bring your own SIP trunk when you already run your own telephony.

Napster does not provide phone numbers; both options connect your agent to telephony you already have. Both are voice-only.

In-person conversations

For a walk-up experience in a lobby, retail floor, showroom, or event, deploy to a **Kiosk** — a Napster Station that pairs the agent's avatar with a large experience screen it drives as it talks.

Next steps

Once you've picked a channel, its guide walks you through the full setup. For per-session settings like memory, user context, and tags, see **Session Configuration**.

WebRTC

Connect to an agent in the browser using the Web SDK

WebRTC is the browser channel: a real-time audio and video conversation with your agent, running in the user's browser and rendered by the Web SDK. This page covers the channel itself — creating a session, and the video behavior specific to WebRTC. Everything about the client library — installation, initialization, configuration, events, and methods — lives in the Web SDK section.

Connecting to an agent

1 **Create a session**

If you have an Omniagent, create a WebRTC session with a single call:

```
curl -X POST https://companion-api.napster.com/public/agents/agent_abc123/connections
\
  -H "X-Api-Key: $NAPSTER_API_KEY" \
  -H "Content-Type: application/json" \
  -d '{
    "channelType": "webrtc"
  }'
```

The agent's companion, voice, tools, knowledge, and provider settings are all inherited from the agent configuration. You can optionally pass `externalClientId` to enable cross-session memory.

The response returns a `token` and a `connection` object containing the session `id`. Pass the token to the Web SDK — you don't need to decode it. Store the connection `id` on your backend to retrieve session details later, such as transcripts or duration.

Per-session alternative (without an Omniagent)

If you don't have an Omniagent and want to assemble everything per request, call `POST /public/connections` instead:

2

Video policy

By default a WebRTC session becomes ready only when audio and video are both up — `avatar_state_changed` reports `ready` once, with the avatar fully rendered. If you'd rather let the conversation start as soon as audio is available, and let the video avatar appear a moment later, set `videoPolicy` to `"deferred"`.

Value	Behavior
`required` (default)	The session waits for audio and video together. `ready` fires once, with the avatar fully up.
`deferred`	The session becomes ready with audio first; the video avatar initializes in the background and comes up moments later. `ready` fires twice.

Set it on the WebRTC channel config to apply to every session on the channel:

```
curl -X PUT https://companion-api.napster.com/public/agents/agent_abc123/channels/webrtc \
  -H "X-Api-Key: $NAPSTER_API_KEY" \
  -H "Content-Type: application/json" \
  -d '{
    "videoPolicy": "deferred"
  }'
```

Per-session connections (`POST /public/connections`) accept `videoPolicy` directly in the request body. Sessions created from an agent (`POST /public/agents/{agentId}/connections`) inherit the value from the agent's WebRTC channel config.

> `videoPolicy` applies to WebRTC only. WebSocket sessions carry no video, so the parameter has no effect there.

What deferred changes in the event stream

With `"deferred"`, readiness arrives in two steps, and each `ready` carries a `details` object describing what became available:

```
avatar_state_changed  { "state": "preparing" }
avatar_state_changed  { "state": "ready",     "details": { "mode": "audio_only" } }
avatar_state_changed  { "state": "preparing", "details": { "stage": "initializing" } }
avatar_state_changed  { "state": "ready",     "details": { "mode": "audio_video",
"video_type": "hd" } }
```

The conversation is fully live from the first `ready` — the agent can hear, speak, and call tools before video exists. The second `ready` announces the video upgrade.

Write your handlers to tolerate both policies:

- **Latch readiness.** Treat `ready` as a state you enter once, not a one-shot signal. Run session-start logic on the first `ready` and ignore repeats — anything triggered naively "on ready" (a greeting nudge, analytics, a UI transition) runs twice under `"deferred"`.
- **Prefer `onAvatarReady` in the Web SDK.** It fires exactly once, when the avatar video is actually rendering. Under `"deferred"` that is *after* the conversation went live, so don't gate audio-dependent logic on it.
- **Distinguish the steps with `details.mode`** — `"audio_only"` vs `"audio_video"` — when you need to react to the video upgrade specifically.

Green screen background

By default the agent's video stream comes with its own background. If you want to composite the avatar over your own UI — your brand colors, a custom backdrop, a product page — enable the green-screen mode.

This is a two-sided setting. Both halves must match, or the visuals break.

Step 1 — Enable on the channel config

Set `useGreenVideo` to `true` on the WebRTC channel config:

```
curl -X PUT https://companion-api.napster.com/public/agents/agent_abc123/channels/webrtc \
  -H "X-Api-Key: $NAPSTER_API_KEY" \
  -H "Content-Type: application/json" \
  -d '{
    "useGreenVideo": true
  }'
```

This tells the server to send the avatar's video with a green-screen backdrop instead of the default scene.

Step 2 — Match it in the SDK init

In your client, set `avatarStyle.view` to `"silhouette"` when initializing the SDK:

```
await NapsterCcompanionApiSdk.init(token, {
  mountContainer: "#avatar-container",
  avatarStyle: { view: "silhouette" },
});
```

The SDK's silhouette view chroma-keys the green out, leaving the avatar floating over whatever you've placed behind the container.

> ⚠️ Both settings must be set together. If `useGreenVideo` is on but the SDK view is `"rectangle"`, you'll see a green background on the avatar. If the SDK view is `"silhouette"` but `useGreenVideo` is off, the chroma-key has nothing to remove and the avatar's normal background bleeds through.

WebSockets

Connect to an agent over WebSockets for audio or text sessions

WebSockets provide a lightweight connection to your agent for **audio** or **text** sessions. This is ideal for server-side integrations, headless clients, or any use case where you don't need a video avatar.

Connecting to an agent

1

Create a session

If you have an Omniagent, create a WebSocket session with a single call:

```
curl -X POST https://companion-api.napster.com/public/agents/agent_abc123/connections
\
  -H "X-Api-Key: $NAPSTER_API_KEY" \
  -H "Content-Type: application/json" \
  -d '{
    "channelType": "websocket"
  }'
```

The agent's companion, voice, tools, knowledge, and provider settings are all inherited from the agent configuration. You can optionally pass `externalClientId` to enable cross-session memory.

The response returns a JSON object containing an encoded token and a `connection` object with the session `id`:

```
{
  "token": "eyJhbGci...",
  "connection": {
    "id": "conn_abc123"
  }
}
```

Store the connection `id` on your backend to retrieve session details later, such as transcripts or duration.

Per-session alternative (without an Omniagent)

If you don't have an Omniagent and want to assemble everything per request, call `POST /public/ws-connections` instead:

```
curl -X POST https://companion-api.napster.com/public/ws-connections \
  -H "X-Api-Key: $NAPSTER_API_KEY" \
  -H "Content-Type: application/json" \
  -d '{
    "companionId": "comp_abc123",
    "providerConfig": {
      "voiceId": "alloy",
      "settings": {
        "temperature": 0.7
```

2 Decode the token

The token is base64-encoded JSON. Decode it and read the `url` field — the WebSocket endpoint already has the token embedded as a `?token=…` query parameter, so `url` is the only value you need to connect:

NODE.JS

```
const decoded = JSON.parse(Buffer.from(token, "base64").toString());
const { url } = decoded;
```

PYTHON

```
import base64
import json

decoded = json.loads(base64.b64decode(token))
url = decoded["url"]
```

BROWSER

```
const decoded = JSON.parse(atob(token));
const { url } = decoded;
```

The decoded object also includes `token`, `connection`, and `expiresAt`, but `url` is all you need to connect — the token is already embedded in it. Don't append `?token=` yourself; a second one produces an invalid URL.

Open the WebSocket before `expiresAt`. If the client doesn't connect in time, the session closes with `closeReason: "connection_timeout"`.

3 Open the WebSocket connection

Pass the decoded `url` straight to your WebSocket client — the token is already in the URL, so you don't set any auth header or query parameter yourself. The `https` URL is upgraded to a secure WebSocket connection automatically; you don't need to change the scheme:

NODE.JS

```
import WebSocket from "ws";

const ws = new WebSocket(url);

ws.on("open", () => {
  console.log("Connected to agent");
});

ws.on("message", (data) => {
  const event = JSON.parse(data);
  // Handle incoming events from the agent
});
```

PYTHON

```python
import websocket
import json

ws = websocket.WebSocketApp(
    url,
    on_open=lambda ws: print("Connected to agent"),
    on_message=lambda ws, msg: print(json.loads(msg)),
)

ws.run_forever()
```

BROWSER

```javascript
const ws = new WebSocket(url);

ws.addEventListener("open", () => {
  console.log("Connected to agent");
});

ws.addEventListener("message", (event) => {
  const data = JSON.parse(event.data);
  // Handle incoming events from the agent
});
```

Audio protocol

Audio is streamed as **16-bit integer PCM, 16 kHz, mono**, base64-encoded in both directions.

Sending audio

Send audio to the agent using the `send_audio` message type:

```
{
  "type": "send_audio",
  "data": {
    "data": "<base64-encoded PCM audio>"
  }
}
```

Receiving audio

The agent sends audio back via `audio_received` events:

```
{
  "event": "audio_received",
  "data": {
    "data": "<base64-encoded PCM audio>"
  }
}
```

Text sessions

By default, a WebSocket session runs in **audio** mode — you stream microphone audio and receive the agent's voice, as described above. To run a **text-only** session instead — a typed text chat with no voice or audio (plain text in, plain text out, chatbot-style) — set `modality` to `"text"` when you create the connection:

```
curl -X POST https://companion-api.napster.com/public/agents/agent_abc123/connections \
  -H "X-Api-Key: $NAPSTER_API_KEY" \
  -H "Content-Type: application/json" \
  -d '{
    "channelType": "websocket",
    "modality": "text"
  }'
```

`modality` accepts `audio` (the default) or `text`, and is also available on the per-session `POST /public/ws-connections` call. For a video avatar, use WebRTC instead.

Text mode is specifically *text without audio*. In an audio session you already receive the agent's text transcript through `message_received` events alongside the streamed audio — text mode simply drops the audio and keeps the text.

In a text session you don't use `send_audio` / `audio_received`. Send the user's message with the `send_message` command, and read the agent's reply from `message_received` events:

```
// send — the user's message
{ "type": "send_message", "data": { "role": "user", "text": "Hello", "trigger_response":
true } }
```

```
// receive — the agent's reply (note the fields are nested under data.message, not data)
{ "event": "message_received", "data": { "message": { "role": "assistant", "action":
"delta", "content": "Hi" } } }
```

See Client Commands for the full `send_message` reference and Server Events for the other events a session emits.

> A text session has no speech, so speaking-related events never fire — there's no turn detection or barge-in, and no `talk_state_changed`. You still receive `message_received`, `avatar_state_changed`, and the other non-speech events — including tool calls (`function_implicitly_called`) and the session errors and shutdown events.

Events and commands

The server sends events to your client throughout the session — state changes, speech activity, transcription, and response streaming. For the full event reference, message structure, and lifecycle details, see Server Events.

You can also send commands back to the server during a session to inject text messages or update configuration in real time. See Client Commands.

Barge-in (interruption)

Turn detection is always active in audio sessions. The server continuously listens for user speech during agent output and sends a `speech_started` event when the user interrupts. To handle barge-in correctly, your client should:

1. **Stop all queued audio playback immediately.** When the server detects an interruption, it cancels the current response — but any audio already buffered on the client will keep playing unless you clear it.
2. **Keep sending microphone audio at all times.** Do not mute the mic during agent playback. If the mic is muted, the server cannot detect that the user is speaking, and interruption becomes impossible.
3. **Enable echo cancellation.** In browser environments, set `echoCancellation: true` in `getUserMedia`. Without it, the agent's own audio gets picked up by the microphone and creates a feedback loop where the agent constantly interrupts itself.

> With a Cascade API key, `speech_started` arrives after the user's speech is transcribed rather than when they start speaking, so interruptions are detected later.

> WebSocket sessions run in audio or text mode. If you need a video avatar, use WebRTC with the Web SDK instead.

VoIP

Let your agent answer phone calls via Twilio

VoIP is the simplest way to put your Omniagent on a phone number. Napster gives you a webhook URL — you register it with Twilio as the incoming-call webhook for a phone number. When someone calls, Twilio routes the call to Napster, and your agent answers.

No trunk credentials or SIP registration to manage — you bring your own Twilio account and phone number, and point it at Napster.

> Twilio is currently the only supported VoIP provider. To use different telephony infrastructure, see the SIP channel.

How it works

```
Caller → Your phone number (Twilio)
       → Incoming-call webhook → voipEndpoint (Napster)
       → Your Omniagent answers
```

You configure the VoIP channel once. Napster returns a `voipEndpoint` URL. You set that URL as the incoming-call webhook on your phone number in Twilio. Every call to that number is answered by your agent.

Setting up VoIP

1 Create an agent

If you don't already have one, create an agent. See Building Your Omniagent for the full walkthrough.

```
curl -X POST https://companion-api.napster.com/public/agents \
  -H "X-Api-Key: $NAPSTER_API_KEY" \
  -H "Content-Type: application/json" \
  -d '{
    "companionId": "comp_abc123",
    "name": "Phone Support Agent",
    "voiceId": "alloy",
    "providerSettings": {
      "temperature": 0.7,
      "turnDetection": {
        "threshold": 0.9,
        "silence_duration_ms": 800
      }
    }
  }'
```

> For phone calls, a longer `silence_duration_ms` (e.g. 800ms) works better than the default 500ms. Phone audio quality is lower and callers tend to pause more.

2 Enable the VoIP channel

Call `PUT /public/agents/{agentId}/channels/voip`. The request body can be empty, or include per-channel overrides for provider settings, tools, knowledge, or FAQs.

```
curl -X PUT https://companion-
api.napster.com/public/agents/agent_abc123/channels/voip \
  -H "X-Api-Key: $NAPSTER_API_KEY" \
  -H "Content-Type: application/json" \
  -d '{}'
```

Response:

```
{
  "id": "chanconfig_abc123",
  "channelType": "voip",
  "voipEndpoint": "https://companion-api.napster.com/voip/webhook/abc123",
  "created": 1710000001
}
```

The `voipEndpoint` is the URL you need for the next step.

3 Register the webhook with Twilio

Take the `voipEndpoint` from the response and set it as the incoming-call webhook for your phone number in the Twilio Console:

1. Go to **Phone Numbers** → **Manage** → **Active Numbers** and select your number.
2. Under **Voice Configuration**, set the `voipEndpoint` URL as the **"A call comes in"** webhook.
3. Save the configuration.

Once set, any call to that phone number is routed to your agent.

4 Test it

Call the phone number. Your agent should answer. If it doesn't, verify:

- The `voipEndpoint` is correctly set as the webhook in Twilio
- The agent has a valid `companionId` (and `voiceId` if not using a digital twin)
- Your API key is active

Channel overrides

You can override the agent's default settings specifically for VoIP calls. Include any of these fields when enabling the channel:

```
curl -X PUT https://companion-api.napster.com/public/agents/agent_abc123/channels/voip \
  -H "X-Api-Key: $NAPSTER_API_KEY" \
  -H "Content-Type: application/json" \
  -d '{
    "providerSettings": {
      "turnDetection": {
        "silence_duration_ms": 800
      }
    },
    "functions": ["fn_phone_only_tool"],
    "faqCollections": ["faq_phone_support"]
  }'
```

Field	Description
`providerSettings`	Override temperature, turn detection, noise reduction for phone calls
`functions`	Use different tools for phone vs web
`faqCollections`	Use different FAQs for phone vs web
`knowledgeBaseId`	Use a different knowledge base for phone
`useWebSearch`	Enable or disable web search for phone calls

If omitted, the agent's base configuration is used.

Managing the VoIP channel

```
# Get current VoIP config
curl https://companion-api.napster.com/public/agents/agent_abc123/channels/voip \
  -H "X-Api-Key: $NAPSTER_API_KEY"

# Remove VoIP channel
curl -X DELETE https://companion-api.napster.com/public/agents/agent_abc123/channels/voip \
  -H "X-Api-Key: $NAPSTER_API_KEY"
```

Deleting the VoIP channel invalidates the `voipEndpoint`. Calls to the phone number will no longer be routed to your agent.

SIP

Let your agent answer incoming phone calls — and place outbound ones — over SIP

> ⚠ SIP connections require dedicated capacity. The default quota is **0** — you must request SIP quota before creating connections.

SIP integration lets your Omniagent answer incoming phone calls — and place outbound ones. You register a SIP extension with the Napster API, and every call to that extension is picked up by your agent — the same agent you can deploy to WebRTC and WebSockets.

Unlike WebRTC and WebSocket connections — which are created per session — SIP is a persistent setup. You configure it once, and the agent answers calls continuously until you remove the configuration.

SIP quota

To request SIP quota, call `POST /public/quota-requests`:

```
curl -X POST https://companion-api.napster.com/public/quota-requests \
  -H "X-Api-Key: $NAPSTER_API_KEY" \
  -H "Content-Type: application/json" \
  -d '{
    "quotaType": "sipPods",
    "requestedValue": 2,
    "reason": "Production SIP deployment for customer support line"
  }'
```

Parameter	Type	Required	Description
`quotaType`	string	Yes	The type of quota to request. Use `"sipPods"` for SIP.
`requestedValue`	integer	Yes	The number of concurrent SIP connections you need.
`reason`	string	No	A description of your use case. Helps expedite the review.

The request is reviewed by the Napster team. Once approved, the requested number of concurrent SIP connections becomes your maximum. You can check the status of your requests with `GET /public/quota-requests`.

How it works

SIP uses three resources that you manage separately:

Resource	What it defines
Agent	*Who answers the call* — the companion, voice, language, tools, FAQs, knowledge, and provider settings. Agents are channel-agnostic and can be used across SIP, WebRTC, and WebSocket.
SIP Channel Config	*SIP-specific overrides* — configures human handoff. Can optionally override the agent's default resources and provider settings for SIP calls.
SIP Connection	*Where to listen for calls* — the SIP credentials (server, port, domain, username, password, transport) that connect to your telephony provider.

A SIP Connection links to an Agent. One agent can serve multiple connections — for example, the same agent answering on different phone numbers or extensions.

```
Agent (companion + voice + config)
  └── SIP Channel Config (human handoff)
        ├── SIP Connection A (extension 101)
        ├── SIP Connection B (extension 102)
        └── SIP Connection C (extension 103)
```

Setting up SIP

1

Create an Agent

Define which companion answers and how it behaves. The agent holds the base configuration that applies across all channels.

```
curl -X POST https://companion-api.napster.com/public/agents \
  -H "X-Api-Key: $NAPSTER_API_KEY" \
  -H "Content-Type: application/json" \
  -d '{
    "companionId": "comp_abc123",
    "name": "Support Agent",
    "voiceId": "alloy",
    "language": "en",
    "functions": ["fn_order_status", "fn_returns"],
    "faqCollections": ["faq_support"],
    "knowledgeBaseId": "kb_product_docs",
    "providerSettings": {
      "temperature": 0.7,
      "turnDetection": {
        "threshold": 0.9,
        "silence_duration_ms": 500
      }
    }
  }'
```

Response:

```
{
  "id": "agent_abc123",
  "companionId": "comp_abc123",
  "name": "Support Agent",
  "voiceId": "alloy",
  "language": "en",
  "functions": ["fn_order_status", "fn_returns"],
  "faqCollections": ["faq_support"],
  "knowledgeBaseId": "kb_product_docs",
  "providerSettings": {
    "temperature": 0.7,
    "turnDetection": {
      "threshold": 0.9,
      "silence_duration_ms": 500
    }
  },
  "created": 1710000000
}
```

Parameter	Type	Required	Description
`companionId`	string	Yes	The companion that answers calls.
`name`	string	Yes	A label for this agent.
`voiceId`	string	Conditional	The voice for speech output (e.g. `alloy`, `echo`, `shimmer`). Required for companions, not required for digital twins.
`language`	string	No	Lock the agent to a specific language using an ISO code (e.g. `en`, `es`). If omitted, defaults to English but can switch when asked.
`functions`	string[]	No	Tool IDs to attach.
`faqCollections`	string[]	No	FAQ collection ID. At most one.
`knowledgeBaseId`	string	No	Knowledge collection ID.
`providerSettings`	object	Yes	Model and audio settings — `temperature`, `turnDetection`, `noiseReduction`. See Configuration for details.

2 Set up the SIP channel

This step is required even if you don't need human handoff — it registers the SIP channel for the agent under your API key. Without it, creating a SIP connection in the next step will fail with an error.

This uses `PUT` — it creates the channel config if it doesn't exist, or updates it if it does.

```
curl -X PUT https://companion-api.napster.com/public/agents/agent_abc123/channels/sip
\
  -H "X-Api-Key: $NAPSTER_API_KEY" \
  -H "Content-Type: application/json" \
  -d '{
    "humanHandoff": {
      "enabled": true,
      "transferExtension": "200"
    }
  }'
```

Response:

```
{
  "id": "chanconfig_abc123",
  "apiKeyId": "key_abc123",
  "channelType": "sip",
  "humanHandoff": {
    "enabled": true,
    "transferExtension": "200"
  },
  "created": 1710000001
}
```

Parameter	Type	Required	Description
`humanHandoff`	object	No	Configure live agent transfer. See Human handoff below.
`functions`	string[]	No	Override the agent's default tool IDs for SIP calls.
`faqCollections`	string[]	No	Override the agent's default FAQ collection for SIP calls. At most one ID.
`knowledgeBaseId`	string	No	Override the agent's default knowledge collection for SIP calls.
`providerSettings`	object	No	Override the agent's default provider settings for SIP calls.

> The channel config fields for `functions`, `faqCollections`, `knowledgeBaseId`, and `providerSettings` are optional overrides. If omitted, the agent's base configuration is used.

3 Create a SIP Connection

This requires the SIP channel to already be enabled on the agent (see the previous step) — without it, this call returns an error.

Provide the SIP credentials from your telephony provider and link them to the agent. This deploys a listener that registers with your SIP server and starts accepting calls.

```
curl -X POST https://companion-api.napster.com/public/sip-connections \
  -H "X-Api-Key: $NAPSTER_API_KEY" \
  -H "Content-Type: application/json" \
  -d '{
    "agentId": "agent_abc123",
    "name": "Main Support Line",
    "settings": {
      "server": "sip.your-provider.com",
      "port": 5060,
      "domain": "your-provider.com",
      "username": "your-sip-username",
      "password": "your-sip-password",
      "transport": "TCP",
      "callerId": "support-line-01"
    }
  }'
```

Response:


```
{
  "id": "sipconn_abc123",
  "organizationId": "org_abc123",
  "projectId": "proj_abc123",
  "agentId": "agent_abc123",
  "name": "Main Support Line",
  "settings": {
    "server": "sip.your-provider.com",
    "port": 5060,
    "domain": "your-provider.com",
    "username": "your-sip-username",
    "transport": "TCP",
    "callerId": "support-line-01"
  },
  "status": {
    "lifecycleStatus": "Pending",
    "sipStatus": null,
    "message": null,
    "lastRefreshed": 1710000005
  },
  "created": 1710000005
}
```

Parameter	Type	Required	Description
`agentId`	string	Yes	The agent to use for incoming calls on this connection.
`name`	string	No	A label for this connection.
`settings.server`	string	Yes	Your SIP provider's server address.
`settings.port`	integer	Yes	The SIP server port (typically `5060`).
`settings.domain`	string	Yes	The SIP domain.
`settings.username`	string	Yes	Your SIP account username.
`settings.password`	string	Yes	Your SIP account password. Only sent on creation — not returned in responses.
`settings.transport`	string	Yes	The transport protocol (e.g. `TCP`).
`settings.callerId`	string	No	A unique identifier for this connection on your SIP server. Not a phone number shown to callers.

4 Check connection status

After creating a connection, the Napster API deploys a listener and registers it with your SIP server. The `status` object tracks two independent states:

1. `lifecycleStatus` — whether the listener itself is deployed and running.
2. `sipStatus` — whether the listener has successfully registered with your SIP provider.

Both must be healthy for the agent to receive calls.

```
curl -X POST https://companion-api.napster.com/public/sip-
connections/sipconn_abc123/status \
  -H "X-Api-Key: $NAPSTER_API_KEY"
```

Response:

```
{
  "status": {
    "lifecycleStatus": "Running",
    "sipStatus": "online",
    "message": null,
    "lastRefreshed": 1710000010
  },
  "created": 1710000005
}
```

Lifecycle status

Tracks whether the listener is deployed and running.

Value	Description
`Pending`	The listener is being provisioned.
`Running`	The listener is up and operational.
`Failed`	The listener failed to start. Check `status.message` for details.
`Error`	The listener encountered a runtime error. Check `status.message` for details.
`Unknown`	The state could not be determined.

SIP status

Tracks the SIP registration state. This is only relevant once `lifecycleStatus` is `Running`.

Value	Description
`registering`	The listener is running but has not yet registered with your SIP provider.
`online`	Registered and ready to receive calls.
`calling`	Registered and an active call is in progress.

Once `lifecycleStatus` is `Running` and `sipStatus` is `online`, your agent is live — incoming calls to your SIP extension are answered by the companion.

5 Troubleshoot registration errors

If `lifecycleStatus` is `Running` but `sipStatus` stays on `registering` and never transitions to `online`, the listener is up but failing to register with your SIP provider. Use the errors endpoint to retrieve recent error logs for the connection.

```
curl https://companion-api.napster.com/public/sip-connections/sipconn_abc123/errors \
  -H "X-Api-Key: $NAPSTER_API_KEY"
```

Response:

```
[
  {
    "timestamp": "2025-03-10T12:00:05Z",
    "message": "SIP registration failed: 401 Unauthorized",
    "code": 401
  }
]
```

Field	Type	Description
`timestamp`	string	When the error occurred.
`message`	string	A human-readable description of the error.
`code`	integer	The error or SIP response code.

Common issues include incorrect credentials (`401`), unreachable server, or wrong transport protocol. Double-check the `settings` on your SIP Connection if registration fails.

Outbound calls

Your agent can also dial out. Once a SIP Connection is registered, start a call with `POST /public/sip-connections/{sipConnectionConfigurationId}/call` — the agent calls the number and holds the conversation exactly as it would on an incoming call:

```
curl -X POST https://companion-api.napster.com/public/sip-
connections/$SIP_CONNECTION_ID/call \
  -H "X-Api-Key: $NAPSTER_API_KEY" \
  -H "Content-Type: application/json" \
  -d '{
    "destination": "15551234567",
    "initialSpeech": "Hi, this is Ava from Acme — I'\''m calling about your recent order."
  }'
```

```
{
  "status": "dialing"
}
```

Parameter	Type	Required	Description
`destination`	string	Yes	The phone number or extension to call: `+` followed by 7–15 digits, or up to 15 digits. Digits only — no spaces, `*` / `#` codes, or SIP URIs.
`initialSpeech`	string	No	What the agent says as soon as the call is answered. Without it, the agent opens with its default greeting behavior.

The response confirms the call is being placed (`status: "dialing"`). The conversation then shows up in sessions like any other SIP call, with `direction: "outbound"` . If the call isn't answered, the session is `failed` with a `closeReason` of `no_answer` or `rejected` .

Human handoff

The `humanHandoff` setting lets the agent transfer a call to a live person over SIP. When enabled, the agent tells the caller it's transferring them, then hands the call to the extension you configure.

Human handoff is configured on the SIP channel config (the same `PUT` call used during channel setup). You can include it when you first create the channel config, or add it later by updating the same endpoint:

```
curl -X PUT https://companion-api.napster.com/public/agents/agent_abc123/channels/sip \
  -H "X-Api-Key: $NAPSTER_API_KEY" \
  -H "Content-Type: application/json" \
  -d '{
    "humanHandoff": {
      "enabled": true,
      "transferExtension": "200"
    }
  }'
```

Parameter	Type	Required	Description
`humanHandoff.enabled`	boolean	No	Set to `true` to allow the agent to transfer calls.
`humanHandoff.transferExtension`	string	Conditional	The extension to transfer the call to. Required when `enabled` is `true` or `routes` is set. `+` followed by 7–15 digits, or up to 15 digits.
`humanHandoff.routes`	array	No	Up to 10 additional extensions, each for a different kind of request. See Routing to different teams.

Routing to different teams

To send callers to different teams, add `routes`. Each route pairs an extension with a `purpose` that describes what that line handles:

```
curl -X PUT https://companion-api.napster.com/public/agents/agent_abc123/channels/sip \
  -H "X-Api-Key: $NAPSTER_API_KEY" \
  -H "Content-Type: application/json" \
  -d '{
    "humanHandoff": {
      "enabled": true,
      "transferExtension": "200",
      "routes": [
        {
          "extension": "210",
          "purpose": "Billing questions, invoices, and refunds"
        },
        {
          "extension": "220",
          "purpose": "Technical support for devices and connectivity"
        }
      ]
    }
  }'
```

Parameter	Type	Required	Description
`humanHandoff.routes[].extension`	string	Yes	The extension for this line. Same format as `transferExtension`, and unique across routes.
`humanHandoff.routes[].purpose`	string	Yes	What this line handles, in plain language. Up to 200 characters.

When the agent transfers a call, it uses the route whose `purpose` clearly matches what the caller needs. Otherwise, the call goes to `transferExtension`.

> Human handoff is available on the SIP channel only. On other channels, `humanHandoff` in the channel config is ignored.

Managing your agent

For listing, updating, and deleting agents and channel configs, see Managing Your Agent.

Managing SIP Connections

You can list, update, and delete your SIP connections:

```
# List all SIP connections
curl https://companion-api.napster.com/public/sip-connections \
  -H "X-Api-Key: $NAPSTER_API_KEY"

# Update a SIP connection
curl -X PATCH https://companion-api.napster.com/public/sip-connections/sipconn_abc123 \
  -H "X-Api-Key: $NAPSTER_API_KEY" \
  -H "Content-Type: application/json" \
  -d '{
    "name": "Updated Support Line"
  }'

# Delete a SIP connection
curl -X DELETE https://companion-api.napster.com/public/sip-connections/sipconn_abc123 \
  -H "X-Api-Key: $NAPSTER_API_KEY"
```

Deleting a connection stops the listener and unregisters from your SIP server.

Kiosk

Deploy your Omniagent to a Napster Station for in-person, walk-up conversations

Kiosk puts your Omniagent on a **Napster Station** — a dedicated device for in-person, walk-up conversations in places like lobbies, retail floors, showrooms, and events. Unlike the other channels, kiosk runs on Napster hardware rather than a client you build and host.

> ⚠ Kiosk requires a Napster Station. See napster.com/station for more details, and contact us to get set up.

How a Station works

A Napster Station has two screens:

- **The experience screen** — a large display that runs your **experience**: a custom, branded app the agent presents and drives. In a retail mall, for instance, this is where a visitor sees products, wayfinding, or promotions.
- **The agent screen** — a smaller display embedded in the Station that shows the agent's avatar. This is who the visitor talks to.

A visitor speaks to the agent on the small screen, and the agent drives the experience on the large screen — surfacing the right content as the conversation unfolds.

The experience

The experience is a **separate application you build and host** — it isn't generated by Napster. Because it's bespoke to your Station and use case, building and wiring up the experience is done together with the Napster team. Contact us for the details, at least for now.

When you set up the kiosk channel, you define your experience — its **name** (the `theme` field) and the **URL** where it's hosted (the `url` field) — alongside the usual agent settings.

Setting up kiosk

1 Create an agent

If you don't already have one, create an agent. See Building Your Omniagent for the full walkthrough.

```
curl -X POST https://companion-api.napster.com/public/agents \
  -H "X-Api-Key: $NAPSTER_API_KEY" \
  -H "Content-Type: application/json" \
  -d '{
    "companionId": "comp_abc123",
    "name": "Lobby Concierge",
    "voiceId": "alloy",
    "providerSettings": {
      "temperature": 0.7
    }
  }'
```

2 Configure the kiosk channel

Call `PUT /public/agents/{agentId}/channels/kiosk`. Name the experience with `theme`, point the channel at it with `url`, and use `initialSpeech` to tell the agent how to open with each visitor.

```
curl -X PUT https://companion-
api.napster.com/public/agents/agent_abc123/channels/kiosk \
  -H "X-Api-Key: $NAPSTER_API_KEY" \
  -H "Content-Type: application/json" \
  -d '{
    "theme": "Lobby Concierge",
    "url": "https://experiences.example.com/lobby-concierge",
    "initialSpeech": "Welcome the visitor and offer to help them find their way
around."
  }'
```

3 Pair the Station

Pairing the physical Station with your agent — and deploying your experience to it — is handled during setup with the Napster team. Once paired, the Station runs your agent and experience.

Channel overrides

You can override the agent's default settings specifically for the kiosk. Include any of these fields when configuring the channel:

```
curl -X PUT https://companion-api.napster.com/public/agents/agent_abc123/channels/kiosk \
  -H "X-Api-Key: $NAPSTER_API_KEY" \
  -H "Content-Type: application/json" \
  -d '{
    "theme": "Exhibit Guide",
    "url": "https://experiences.example.com/exhibit",
    "initialSpeech": "Welcome the visitor to the exhibit and ask what they would like to
explore.",
    "functions": ["fn_kiosk_directory"],
    "faqCollections": ["faq_venue_info"]
  }'
```

Field	Description
`theme`	The name of the experience that runs on the Station's main screen
`url`	The hosted URL of the experience that runs on the Station's main screen
`initialSpeech`	Instructions for how the agent should open with each visitor — see Session Configuration — Opening instructions
`providerSettings`	Override temperature, turn detection, and noise reduction for the kiosk
`functions`	Use different tools on the kiosk than on other channels
`faqCollections`	Use different FAQs on the kiosk
`knowledgeBaseId`	Use a different knowledge base on the kiosk
`useWebSearch`	Enable or disable web search on the kiosk
`idleTimeoutSeconds`	How long to wait before ending an inactive conversation and resetting for the next visitor

If omitted, the agent's base configuration is used.

Managing the kiosk channel

```
# Get current kiosk config
curl https://companion-api.napster.com/public/agents/agent_abc123/channels/kiosk \
  -H "X-Api-Key: $NAPSTER_API_KEY"

# Remove kiosk channel
curl -X DELETE https://companion-api.napster.com/public/agents/agent_abc123/channels/kiosk \
  -H "X-Api-Key: $NAPSTER_API_KEY"
```

Reviewing kiosk conversations

Every kiosk interaction is captured as a session. List them and filter by type in the Sessions API — the `sessionType` is `kiosk`, and each record includes timing, cost, and the full transcript. Because walk-up visitors are anonymous, kiosk sessions typically have no `externalClientId`.

Overview

Communicate with your agent during a session through events and commands

Messaging Overview

Once a WebRTC or WebSocket connection is established, all communication with your agent happens through a real-time message protocol. Messages flow in two directions:

- **Server to client**: The server pushes events as the conversation progresses — speech detection, transcription updates, response streaming, and state changes.
- **Client to server**: Your client can send text messages to the agent and update session settings on the fly.

Server Events

Server-to-client events published during WebRTC and WebSocket sessions

The server sends events to your client throughout a session. Events notify you of state changes, speech activity, and conversation content as it happens.

Listening for events

WebRTC (Web SDK)

Pass an `onData` callback when initializing the SDK. Every server event is delivered through this handler:

```
const instance = await NapsterCompanionApiSdk.init(token, {
  onData: (data) => console.log("Event received:", data),
});
```

WebSocket

Listen for messages on the WebSocket connection. Each message is a JSON-encoded event:

```
ws.onmessage = (event) => {
  const data = JSON.parse(event.data);
  console.log("Event received:", data);
};
```

Event types

Event	Description
`avatar_state_changed`	The agent's readiness state changed
`talk_state_changed`	The agent's speaking state changed
`message_received`	A conversation message event with an `action` field describing the lifecycle stage
`ui_update`	Arbitrary JSON pushed by your own functions server — the platform relays it to the client untouched
`avatar_connection_warning`	The session is idle and will close soon — see Session errors and shutdown
`session_expired`	The session reached its time limit and is closing
`no_credits_left`	Your organization ran out of credits and the session is closing
`provider_connection_aborted`	The AI provider rejected the session's configuration and the session is closing
`reaching_rate_limits`	The AI provider is close to a rate limit

`avatar_state_changed`

Fires when the agent's readiness state changes.

```
{
  "event": "avatar_state_changed",
  "data": {
    "state": "ready",
    "timestamp": "2026-07-22T08:03:14.309123+00:00",
    "details": { "mode": "audio_only" }
  }
}
```

State	Description
`preparing`	The agent is initializing
`ready`	The agent is ready to receive input

`data.details`, when present, describes what became available — for example `mode: "audio_only"` or `mode: "audio_video"`.

> ⚠ `ready` can fire more than once per session. On WebRTC with `videoPolicy: "deferred"`, the session becomes ready with audio first and reports `ready` again when the video avatar comes up — see Video policy. Latch on the first `ready` instead of re-running session-start logic on every occurrence.

`talk_state_changed`

Fires when the agent starts or stops speaking.

```
{
  "event": "talk_state_changed",
  "data": {
    "talk_id": "item_1",
    "state": "started",
    "sentence_index": 0
  }
}
```

Field	Description
`talk_id`	Identifies what the agent is saying. In puppeteer mode, it's the `talkId` you sent with `talk`.
`state`	The speaking state — see below.
`sentence_index`	Which sentence of the reply is being spoken, starting at `0`. Present on `started`, `in_progress`, and `ended`.

State	Description
`preparing`	The agent is preparing to speak
`started`	The agent is actively speaking
`in_progress`	The agent is still speaking and has moved on to another sentence
`ended`	The agent has finished speaking
`canceled`	The agent stopped before finishing — for example, because the user started speaking

`message_received`

The primary event for tracking the conversation. The conversation fields are nested under `data.message`:

```
{
  "event": "message_received",
  "data": {
    "message": {
      "type": "message",
      "role": "user | assistant",
      "action": "string",
      "response_id": "string",
      "item_id": "string",
      "content": "string",
      "content_index": 0,
      "previous_item_id": "string",
      "reason": "string",
      "error": "string",
      "tokens": 0,
      "is_forced": false,
      "timestamp": "2026-03-04T12:00:00Z"
    }
  }
}
```

> The same structure applies on both WebSocket and WebRTC. Over WebRTC, the Web SDK delivers the event to your `onData` handler exactly as it arrives on the wire.

Not every field is present in every event. The fields included depend on the `action`.

Field	Type	Description
`type`	`"message"` or `"session"`	Frame kind. A `"session"` frame is a session marker with no `role` / `content` — skip it (see below)
`role`	`"user"` or `"assistant"`	Who the message is from
`action`	string	The lifecycle stage of the message
`response_id`	string	Groups related events in the same response
`item_id`	string	Unique identifier for this conversation item
`content`	string	Text content (transcription or response text)

Field	Type	Description
`content_index`	number	Index of the content part within the item
`previous_item_id`	string	ID of the preceding item, used to preserve ordering
`reason`	string	Present in `cancelled` events (e.g., `"turn_detected"`)
`error`	object	Present in `failed` events
`tokens`	number	Token count for the response (assistant events)
`is_forced`	boolean	`true` when the message was injected by the server rather than produced by user turn-taking
`timestamp`	datetime	When the event occurred (included in every message)

> ⚠️ The **first** `message_received` frame of a session is a session marker: `type: "session"` with no `role`, `action`, or `content`. Skip any frame where `data.message.type === "session"` before processing.

The examples below show the contents of the `message` object — i.e. what you find at `data.message`.

User message lifecycle

These events track the user's speech input and transcription, all with `role: "user"`.

`speech_started`

The user began speaking. The `item_id` identifies the user message item that will be created when speech stops.

```
{
  "role": "user",
  "action": "speech_started",
  "item_id": "item_APuyFRtVtA9m8o9Qhyj9e"
}
```

`speech_stopped`

The user stopped speaking.

```
{
  "role": "user",
  "action": "speech_stopped",
  "item_id": "item_APuyFRtVtA9m8o9Qhyj9e"
}
```

`created`

The user's speech transcription has started. The `previous_item_id` links this item to the preceding item in the conversation.

```
{
  "role": "user",
  "action": "created",
  "item_id": "item_APuyFRtVtA9m8o9Qhyj9e",
  "previous_item_id": "item_APuxlnjrohdgargU6tHe9"
}
```

`completed`

The user's speech transcription is finalized. The `content` field contains the full transcription text.

```
{
  "role": "user",
  "action": "completed",
  "item_id": "item_APuyFRtVtA9m8o9Qhyj9e",
  "content": "Hi there!"
}
```

⚠ This event can arrive **after** assistant response events have already started. Do not assume user transcription completes before the assistant begins responding.

ℹ If `content` is empty, the user's speech could not be recognized. You may want to display a placeholder like "[inaudible]" in your UI.

`failed`

The user's speech transcription failed. The `error` field contains details.

```
{
  "role": "user",
  "action": "failed",
  "item_id": "item_APuyFRtVtA9m8o9Qhyj9e",
  "error": { }
}
```

Assistant response lifecycle

These events track the agent's response generation, all with `role: "assistant"`.

`created` (response-level)

A new response has started generating. The `response_id` groups all subsequent events for this response.

```
{
  "role": "assistant",
  "action": "created",
  "response_id": "resp_APuyF0H6ffvKBP0MJE6h9"
}
```

`created` (item linked to response)

A response item has been linked to the response.

```
{
  "role": "assistant",
  "action": "created",
  "response_id": "resp_APuyF0H6ffvKBP0MJE6h9",
  "item_id": "item_APuy7jTGdjgsW0gOGOko2"
}
```

`created` (item-level)

The response item has started generating. The `previous_item_id` preserves ordering in the conversation.

```
{
  "role": "assistant",
  "action": "created",
  "item_id": "item_APuy7jTGdjgsW0gOGOko2",
  "previous_item_id": "item_APuyFRtVtA9m8o9Qhyj9e"
}
```

`delta`

An incremental text chunk of the response. These arrive in rapid succession as the agent generates its reply.

```
{
  "role": "assistant",
  "action": "delta",
  "response_id": "resp_APuyF0H6ffvKBP0MJE6h9",
  "item_id": "item_APuy7jTGdjgsW0gOGOko2",
  "content": "Hello! How can I"
}
```

`completed`

The response item is finished. The `content` field contains the full response text.

```
{
  "role": "assistant",
  "action": "completed",
  "response_id": "resp_APuyF0H6ffvKBP0MJE6h9",
  "item_id": "item_APuy7jTGdjgsW0gOGOko2",
  "content": "Hello! How can I assist you today?"
}
```

`interrupted`

The user barged in while the agent was responding. This fires on the user side with `role: "user"`.

```
{
  "role": "user",
  "action": "interrupted",
  "item_id": "item_APuyFRtVtA9m8o9Qhyj9e"
}
```

`cancelled`

The response was cancelled. The `reason` field explains why (e.g., `"turn_detected"` when the user started speaking).

```
{
  "role": "assistant",
  "action": "cancelled",
  "response_id": "resp_APuyF0H6ffvKBP0MJE6h9",
  "reason": "turn_detected"
}
```


`failed`

The response generation failed. The `error` field contains details.


```
{
  "role": "assistant",
  "action": "failed",
  "response_id": "resp_APuyF0H6ffvKBP0MJE6h9",
  "error": { }
}
```


MCP tool lifecycle

These events report what the agent is doing with an attached MCP server. They have no `role` — instead, `type` tells you which stage you are looking at.

Here is the order they arrive in during a session:

1. **The session starts.** You receive one `mcp_tools` message for each attached server, listing the tools that server exposes — or reporting that it could not be reached.
2. **The agent decides to call a tool.** You receive an `mcp_call` message with `action: "created"`, naming the tool and the arguments it will use.
3. **The call finishes.** You receive a second `mcp_call` message, same `item_id`, with `action: "completed"` or `"failed"`.

Steps 2 and 3 repeat for every tool call the agent makes.

If the server was registered with `requireApproval: "always"`, one more message arrives between steps 2 and 3: an `mcp_approval_request`. The call does not run until you answer it, and if you never do, it is dropped after 15 seconds.

`mcp_tools` event

Tool discovery, sent once per attached server as the session starts.

```
{
  "type": "mcp_tools",
  "action": "completed",
  "server": "gmail",
  "tool_names": ["search_emails", "get_recent_emails"]
}
```

Field	Type	Description
`action`	string	`completed` or `failed` — see below
`server`	string	ID of the MCP server this message is about, as you attached it (for example `gmail`)
`tool_names`	array of strings	The tools this server exposes, for example `["search_emails", "read_email"]`. Present on `completed`; an empty array means the server exposes nothing
`error`	string	A message naming the server that failed, for example `"Failed to list tools for MCP server 'crm'"`. Present on `failed`. It doesn't include the underlying cause — check the server's URL, credentials, and token

Actions

- `completed` — the server was reached and its tools listed. The agent can now call anything in `tool_names`.
- `failed` — the server could not be reached. The agent starts with no tools from it, and the server does not recover on its own. This is your earliest warning of a bad URL, an expired credential or a missing token. For an expired per-user token, send a fresh one.

`mcp_call` event

A tool invocation. You get one message when the call starts and another when it resolves.

```
{
  "type": "mcp_call",
  "action": "created",
  "item_id": "mcp_a1b2c3",
  "server": "gmail",
  "tool": "get_recent_emails",
  "content": "{\"top_k\": 3}"
}
```

Field	Type	Description
`action`	string	`created`, `completed` or `failed` — see below
`item_id`	string	Identifies this call, prefixed `mcp_`. The same value appears on the start and the resolution, so use it to pair them
`server`	string	ID of the MCP server providing the tool
`tool`	string	Name of the tool being called, for example `get_recent_emails`
`content`	string	The call's arguments as a JSON string on `created`, and the tool's output on `completed`
`error`	object	Details of the failure. Present on `failed`

Actions

- `created` — the agent invoked a tool. `content` holds the arguments it chose. The call has not returned yet, and on an approval-gated server it has not even run.
- `completed` — the call returned. `content` holds the output the agent will speak from.
- `failed` — the call errored, typically a rejection from the MCP server itself such as an expired token. The agent says nothing about it.

`mcp_approval_request` event

A tool call is waiting for your decision. Answer it with `send_mcp_approval`.

```
{
  "type": "mcp_approval_request",
  "action": "created",
  "item_id": "mcpr_x9y8z7",
  "server": "gmail",
  "tool": "get_recent_emails",
  "content": "{\"top_k\": 3}"
}
```

Field	Type	Description
`action`	string	`created`, `completed` or `cancelled` — see below
`item_id`	string	The approval request ID, prefixed `mcpr_`. Present on every action. Send it back as `approval_request_id`
`server`	string	ID of the MCP server requesting the call. Present on `created` and `cancelled`
`tool`	string	Name of the tool awaiting approval — show this to the user. Present on `created` and `cancelled`
`content`	string	The arguments the tool would run with, as a JSON string. Show these too: they are what the user is approving. Present on `created`
`result`	string	`approved` or `rejected`. Present on `completed`
`source`	string	How the decision was made. `client` when your app answered with `send_mcp_approval`. Present on `completed`

Actions

- `created` — approval is needed. Present the choice now; the 15-second clock is already running.
- `completed` — somebody answered. `result` tells you which way. Stop showing the prompt.
- `cancelled` — nobody answered within 15 seconds, so the call was rejected on your behalf and will not run. Disable the approval controls.

> `completed` and `cancelled` are mutually exclusive. A request that times out is reported as `cancelled` only — you never receive a `completed` for it. So `completed` always means a real decision was made, and `cancelled` always means nobody made one.

> ⚠️ You have **15 seconds** to answer before the request is rejected automatically. Show the choice as soon as `created` arrives, and disable the controls on `cancelled`.

Session errors and shutdown

These events warn your client that a session is about to end, or tell it why it ended. Handle them so you can tell the user what happened instead of showing a dropped connection.

Event	Payload	What happens next
`avatar_connection_warning`	`{ "disconnect_after": 60 }`	The session has been idle. It closes after `disconnect_after` seconds unless the user speaks or sends a message. You receive the warning at 60, 30, and 10 seconds.
`session_expired`	`null`	The session closes with `closeReason: "session_expired"`.
`no_credits_left`	`null`	The session closes with `closeReason: "no_credits_left"`.
`provider_connection_aborted`	`{ "error": { "code", "message" } }`	The session closes with `closeReason: "provider_connection_aborted"`.
`reaching_rate_limits`	`{ "name", "remaining_rate", "reset_seconds" }`	Nothing — a warning that the AI provider is close to a rate limit.

An idle session closes after 3 minutes without activity, with `closeReason: "idle_timeout"`, unless the agent sets `disableIdleTimeout`. With the idle timeout disabled, you don't receive `avatar_connection_warning`.

`provider_connection_aborted`

The AI provider checks the session's configuration when the session starts — after your client connects, not when you create the connection. If the provider rejects it, you receive this event and the session closes:

```
{
  "event": "provider_connection_aborted",
  "data": {
    "error": {
      "code": "invalid_voice",
      "message": "Invalid voice provided"
    }
  }
}
```

code	Cause
`invalid_voice`	The provider doesn't recognize the agent's `voiceId`.
`invalid_credentials`	The provider rejected the credentials on your API key.
`instructions_too_long`	The agent's instructions exceed the provider's limit.
`connection_failed`	The platform couldn't connect to the provider.

`data` is `null` when the cause is unknown. Fix the configuration on your agent or API key, then open a new session.

`ui_update`

Unlike the other events, `ui_update` doesn't originate from the platform — it's sent by **your own functions server** over the WebSocket that a WebSocket-based explicit tool keeps open during the session. The platform relays it to the client without validating or modifying the payload.

```
{
  "event": "ui_update",
  "data": { "view": "booking-confirmation", "bookingId": "bk_123" }
}
```

`data` is exactly what your server sent — any JSON, any shape. Use it to drive UI from your backend in sync with the conversation: show a confirmation card when a booking tool completes, stream progress for a long-running action, or update page state the agent just changed.

Because the payload is entirely yours, dispatch on a field you define (for example a `data.view` or `data.kind` discriminator) rather than assuming a single shape:

```
onData: (data) => {
  if (data.event === "ui_update") {
    renderFromServer(data.data);
  }
}
```

Delivery notes:

- One-way, fire-and-forget: no acknowledgement, and updates are **not stored** — a client that connects later won't receive past updates. Re-send current state if the UI must survive reconnects.
- Your server can send it at any time while its WebSocket is connected — it is not tied to a tool call.
- Requires at least one WebSocket-based explicit tool on the session; HTTP-based tools have no persistent connection to send from.

Client Commands

Send messages and update session settings from your client during a session

Your client sends commands to the server during an active WebRTC or WebSocket session: `send_message` to send information to the agent, `set_settings` to update session configuration on the fly, `send_mcp_approval` to answer an MCP tool approval, and `talk` to make the agent speak a line in puppeteer mode.

Sending commands

WebRTC (Web SDK)

Call `sendCommand` on the SDK instance. The Web SDK also ships typed command definitions and usage patterns for this — see Web SDK → Messaging.

```
instance.sendCommand({
  type: "send_message",
  data: {
    role: "user",
    text: "Hello",
    trigger_response: true,
    delay: false
  }
});
```

WebSocket

Send the command as JSON over the WebSocket connection:

```
ws.send(JSON.stringify({
  type: "send_message",
  data: {
    role: "user",
    text: "Hello",
    trigger_response: true,
    delay: false
  }
}));
```

`send_message`

Send text to the agent as a user message or a system context update.

```
{
  "type": "send_message",
  "data": {
    "role": "user",
    "text": "What is the status of my order?",
    "trigger_response": true,
    "delay": false
  }
}
```

Parameters

Parameter	Type	Required	Description
`role`	`user` or `system`	Yes	Who the message is from
`text`	string	Yes	The message content
`trigger_response`	boolean	Yes	When `true`, the agent responds immediately. When `false`, the agent absorbs the information silently and may or may not act on it based on conversation context
`delay`	boolean	No	When `true`, waits for the agent to finish speaking before delivering the message and triggering a response. When `false`, delivers immediately — interrupting the agent if it is speaking. Default: `false`

User messages

When `role` is `user`, the agent treats the text as if the user typed it instead of speaking. This is useful for text-based input alongside audio, or for injecting user intent programmatically.

```
{
  "type": "send_message",
  "data": {
    "role": "user",
    "text": "Tell me more about the premium plan",
    "trigger_response": true
  }
}
```

System messages

When `role` is `system`, the text is treated as a context update. The agent incorporates the information into its context without displaying it to the user. Use this when something changes in your application that the agent needs to know about.

```
{
  "type": "send_message",
  "data": {
    "role": "system",
    "text": "The user just added 'Wireless Headphones' to their shopping cart.",
    "trigger_response": false
  }
}
```

Triggering a response

When `trigger_response` is `true`, the agent generates a response immediately. When `false`, the agent absorbs the information silently — it does not respond right away, but may incorporate it into future replies based on the conversation context.

The `delay` parameter controls **when** the message is delivered. When `delay` is `false` (the default), the message is delivered immediately, interrupting the agent if it is currently speaking. When `delay` is `true`, the message is held until the agent finishes its current speech, then delivered — and if `trigger_response` is `true`, the agent responds after that.

```
{
  "type": "send_message",
  "data": {
    "role": "system",
    "text": "The user's subscription has been upgraded to premium.",
    "trigger_response": true,
    "delay": true
  }
}
```

`set_settings`

Update session parameters in real time without creating a new connection. Change the agent's instructions, temperature, turn detection, modality, or tools, or refresh MCP tokens.

```
{
  "type": "set_settings",
  "data": {
    "instructions": "You are a customer support agent. Be concise and helpful.",
    "temperature": 0.7,
    "turn_detection": {
      "threshold": 0.5,
      "prefix_padding_ms": 300,
      "silence_duration_ms": 500
    }
  }
}
```

> All fields are optional. Include only the settings you want to change — omitted fields keep their current values.

Parameters

Parameter	Type	Architecture	Description
`instructions`	string	Realtime, Cascade	Replace the agent's system instructions
`temperature`	number	Realtime	Set the response temperature (0-1)
`turn_detection`	object	Realtime	Configure voice activity detection
`modality`	string	Realtime	Switch between `audio` and `text` mid-session. Switching to `audio` fails if the session has no voice
`inline_functions`	array	Realtime, Cascade	Register tools for the rest of the session. Replaces any tools registered this way earlier. See the example under Common patterns
`mcp.authorizations`	array	Realtime	Replace the tokens for per-user MCP servers. Each item is `{ "mcp_server_id", "token" }`. See Refreshing a token mid-session.

`turn_detection`

Parameter	Type	Description
`threshold`	number	Sensitivity threshold for detecting speech
`prefix_padding_ms`	number	Milliseconds of audio to include before detected speech
`silence_duration_ms`	number	Milliseconds of silence before a turn is considered complete

Common patterns

Replacing instructions mid-conversation — `instructions` fully replaces the agent's system prompt. Use this when the agent's entire role or behavior needs to change, for example when the user moves to a different part of your application:

```
{
  "type": "set_settings",
  "data": {
    "instructions": "You are a checkout assistant for an e-commerce store. Guide the user through completing their purchase. Confirm the shipping address, suggest available shipping options, and answer questions about return policies. Do not suggest additional products."
  }
}
```

> ⚠️ This replaces the full system prompt — it is not appended to the existing instructions. If you want to provide a contextual update without changing the agent's instructions, use `send_message` with `role: system` instead.

Providing context updates — use `send_message` with `role: system` to give the agent situational context without replacing its instructions. For example, notify the agent when the user navigates to a different page:

```
{
  "type": "send_message",
  "data": {
    "role": "system",
    "text": "The user just navigated to the checkout page.",
    "trigger_response": false
  }
}
```

Registering tools mid-session — add tools the agent can call for the rest of the session, without creating them with the API first. Each item uses the same shape as a tool definition:

```
{
  "type": "set_settings",
  "data": {
    "inline_functions": [
      {
        "id": "get_cart",
        "flow": "implicit",
        "data": {
          "name": "get_cart",
          "description": "Get the items currently in the user's shopping cart.",
          "parameters": { "type": "object", "properties": {} }
        }
      }
    ]
  }
}
```

Only implicit tools, and explicit tools served over an `http` or `https` URL, can be registered this way. Names must be unique and can't collide with the agent's own tools. Each `set_settings` with `inline_functions` replaces the tools registered this way before.

Refreshing an MCP token — when a per-user MCP server's access token is about to expire, send the fresh token. The list replaces all tokens, so include every server that needs one:

```
{
  "type": "set_settings",
  "data": {
    "mcp": {
      "authorizations": [
        {
          "mcp_server_id": "crm",
          "token": "ya29.a0AfH6_refreshed..."
        }
      ]
    }
  }
}
```

Adjusting turn detection for noisy environments — increase `silence_duration_ms` so the agent waits longer before treating silence as the end of the user's turn:

```
{
  "type": "set_settings",
  "data": {
    "turn_detection": {
      "silence_duration_ms": 1000
    }
  }
}
```

`send_mcp_approval`

Answer a pending `mcp_approval_request`. Only relevant when an attached MCP server is registered with `requireApproval: "always"`.

```
{
  "type": "send_mcp_approval",
  "data": {
    "approval_request_id": "mcpr_x9y8z7",
    "approve": true
  }
}
```

Parameters

Parameter	Type	Required	Description
`approval_request_id`	string	Yes	The `item_id` from the `mcp_approval_request` event
`approve`	boolean	Yes	`true` runs the tool call; `false` cancels it and the MCP server is never contacted
`reason`	string	No	Free text explaining the decision, passed upstream with it

Send exactly one decision per request — later or duplicate decisions for the same `approval_request_id` are ignored. If nothing arrives within **15 seconds**, the request is rejected automatically and you receive a `cancelled` event.

> ⚠️ The Web SDK does not yet include `send_mcp_approval` in its typed command definitions. In TypeScript you will need to cast the command until a typed method ships.

`talk`

In puppeteer mode, send `talk` to make the agent speak a line of text exactly as written.

```
{
  "type": "talk",
  "data": {
    "talkId": "intro-1",
    "text": "Welcome to Acme. Let me show you around the new dashboard."
  }
}
```

Parameters

Parameter	Type	Required	Description
`talkId`	string	Yes	Your ID for this line. It comes back as `talk_id` on `talk_state_changed` events, so you can tell when the line starts and finishes.
`text`	string	Yes	The text for the avatar to speak.
`split`	boolean	No	Speak the text sentence by sentence. Defaults to `true`.

Track each line with `talk_state_changed`: `preparing`, then `started` and `ended`. If the user starts speaking while the agent is talking, the current line stops and you receive `canceled`.

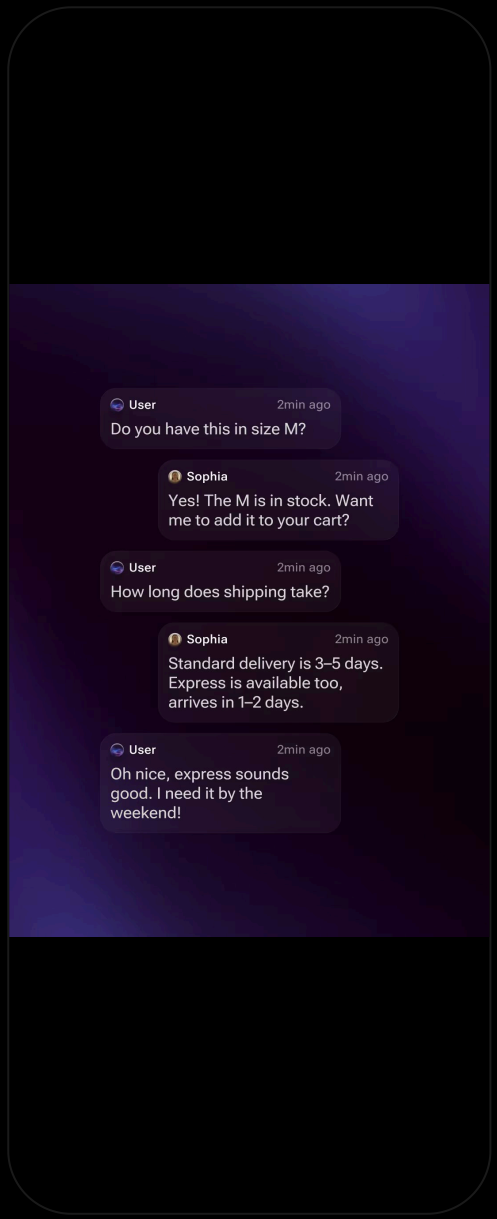
Napster.

SECTION 04

Monitor Your Omniagent

Sessions, transcripts, cost, and webhooks.

Sessions	169
Webhooks	181



Sessions

List sessions, filter by companion or client, and retrieve conversation transcripts

Every connection creates a session. A session captures the full configuration — which companion, tools, knowledge, and FAQs were attached — along with timing data, cost, and the complete conversation transcript between the user and the agent.

The Sessions API gives you three endpoints:

Endpoint	What you get
`GET /public/sessions`	A paginated list of sessions with filters for companion, client, tags, and free-text search
`GET /public/sessions/{sessionId}`	Full session details including configuration, timing, and the complete conversation transcript
`DELETE /public/sessions/{sessionId}`	Permanently deletes a finished session, its transcript, and the memory stored from it

Getting the session ID

When you create a session from your Omniagent, the response includes a `connection.id`. This is the session ID you use to query session data later.

```
curl -X POST https://companion-api.napster.com/public/agents/agent_abc123/connections \
  -H "X-Api-Key: $NAPSTER_API_KEY" \
  -H "Content-Type: application/json" \
  -d '{
    "channelType": "webrtc"
  }'
```

Response:

```
{
  "token": "eyJhbGci...",
  "connection": {
    "id": "sess_abc123"
  }
}
```

Store the `connection.id` on your backend. You need it to retrieve session details and transcripts.

Listing sessions

Retrieve a paginated list of sessions across your project:

```
curl https://companion-api.napster.com/public/sessions?pageSize=10 \
  -H "X-Api-Key: $NAPSTER_API_KEY"
```

Response:


```
{
  "items": [
    {
      "id": "sess_abc123",
      "companionId": "comp_abc123",
      "companion": {
        "id": "comp_abc123",
        "firstName": "Alex",
        "lastName": "Chen"
      },
      "functions": ["fn_def456"],
      "knowledgeBaseId": "kb_abc123",
      "knowledgeBase": {
        "id": "kb_abc123",
        "name": "Product Docs"
      },
      "faqIds": ["faq_abc123"],
      "faqCollections": [
        {
          "id": "faq_abc123",
          "name": "Support FAQs"
        }
      ],
      "externalClientId": "user_12345",
      "tags": {
        "environment": "production",
        "channel": "web"
      },
      "sessionType": "webrtc",
      "modality": "audio",
      "status": "closed",
      "closeReason": "idle_timeout",
      "agent": {
        "id": "agent_abc123",
        "name": "Support Agent"
      },
      "cost": 0.42,
      "createdAt": 1710000000,
      "startedAt": 1710000005,
      "closedAt": 1710000300
    }
  ],
  "filteredCount": 1,
  "totalCount": 42,
  "pageIndex": 0,
  "pageSize": 10
}
```

Filtering

Use query parameters to narrow results:

Parameter	Type	Description
`companionId`	string	Filter by companion
`externalClientId`	string	Filter by your external user or client identifier
`search`	string	Free-text search across sessions
`tags`	string	Filter by tag key-value pairs
`pageIndex`	integer	Page number (zero-based)
`pageSize`	integer	Number of results per page

```
curl "https://companion-api.napster.com/public/sessions?companionId=comp_abc123&pageSize=20"
\
  -H "X-Api-Key: $NAPSTER_API_KEY"
```

Retrieving a session

Get the full details of a session, including the conversation transcript:

```
curl https://companion-api.napster.com/public/sessions/sess_abc123 \
  -H "X-Api-Key: $NAPSTER_API_KEY"
```

Response:

```
{
  "id": "sess_abc123",
  "companionId": "comp_abc123",
  "companion": {
    "id": "comp_abc123",
    "firstName": "Alex",
    "lastName": "Chen"
  },
  "functions": ["fn_def456"],
  "knowledgeBaseId": "kb_abc123",
  "knowledgeBase": {
    "id": "kb_abc123",
    "name": "Product Docs"
```

```
  },
  "faqIds": ["faq_abc123"],
  "faqCollections": [
    {
      "id": "faq_abc123",
      "name": "Support FAQs"
    }
  ],
  "externalClientId": "user_12345",
  "tags": {
    "environment": "production"
  },
  "sessionType": "webrtc",
  "modality": "audio",
  "status": "closed",
  "closeReason": "idle_timeout",
  "agent": {
    "id": "agent_abc123",
    "name": "Support Agent"
  },
  "cost": 0.42,
  "createdAt": 1710000000,
  "startedAt": 1710000005,
  "closedAt": 1710000300,
  "conversation": {
    "items": [
      {
        "role": "agent",
        "text": "Hi there! How can I help you today?",
        "timestamp": 1710000006
      },
      {
        "role": "user",
        "text": "I'd like to check the status of my order.",
        "timestamp": 1710000010
      },
      {
        "role": "agent",
        "text": "Let me look that up for you. Your order #ORD-7890 shipped yesterday and is
expected to arrive by Friday.",
        "timestamp": 1710000015
      }
    ]
  }
}
```

The `conversation.items` array contains each message in chronological order. Each item includes:

Field	Description
`role`	Who sent the message (`agent` or `user`)
`text`	The message content
`timestamp`	Unix timestamp of the message

Session fields

Beyond the transcript, each session record includes:

Field	Description
`agent`	The Omniagent that served the session, as `{ id, name }`. `null` for per-session connections created without an agent.
`cost`	The billed cost of the session, in USD. See Session cost below.
`sessionType`	The channel the session ran on: `webrtc`, `websocket`, `voip`, `sip`, or `kiosk`.
`modality`	The interaction mode the session ran in: `audio`, `text`, or `video`.
`mode`	How the agent spoke: `conversation` or `puppeteer`. See Mode.
`status`	Session state: `pending`, `started`, `closed`, or `failed`. A session moves from `pending` to `started`, then `closed` — or straight to `failed` if the connection is never established. Failed sessions appear in the session list like any other; use this field to separate them.
`closeReason`	Why the session ended. Present once the session is `closed` or `failed`. See Close reasons below.
`direction`	For SIP calls: `inbound` if someone called your agent, `outbound` if your agent placed the call.
`sipConnection`	For SIP calls: the SIP Connection the call went through, as `{ id, name }`.
`createdAt` / `startedAt` / `closedAt`	Unix timestamps for when the session was created, connected, and ended.
`functionStates`	Connection state for each of the session's WebSocket-based tools. Returned only by `GET /public/sessions/{sessionId}`, not on list items. See Tool connection states below.

Close reasons

A session that ends before it starts has a `status` of `failed`; otherwise it's `closed`. Either way, `closeReason` says why:

closeReason	Meaning
`connection_closed`	The connection ended — the user closed the page or hung up, or the network dropped.
`idle_timeout`	The session was idle for too long. See Idle timeout.
`connection_aborted`	Your backend ended the session.
`connection_timeout`	The client didn't connect in time after the session was created.
`not_started`	The session was created, but no client ever connected.
`session_expired`	The AI provider's maximum session length was reached.
`no_credits_left`	Your organization ran out of credits.
`provider_connection_aborted`	The AI provider rejected the session's configuration. See `provider_connection_aborted`.
`function_connection_failed`	A WebSocket tool set to `abortOnFailure: true` couldn't connect.
`function_connection_lost`	A WebSocket tool set to `abortOnFailure: true` lost its connection during the session.
`avatar_allocation_failed`	The avatar couldn't be started.
`initialization_failed` / `session_initialization_failed`	The session couldn't be set up after the client connected (WebRTC / WebSocket).
`application_shutdown`	The server handling the session shut down.
`no_answer`	An outbound SIP call wasn't answered.
`rejected`	An outbound SIP call was declined, or the line was busy.
`failed`	An outbound SIP call couldn't be placed.

closeReason	Meaning
`unknown`	The session stopped reporting and was closed without a reason.

Session cost

The `cost` field reports the billed cost of the session in USD, calculated from the minutes the agent was active and your API key's model configuration. Use it to attribute spend per session, per companion, or per end user.

```
{
  "id": "sess_abc123",
  "sessionType": "webrtc",
  "status": "closed",
  "cost": 0.42,
  "closedAt": 1710000300
}
```

`cost` is `null` while a session is still `pending` or `started` — it is populated once the session closes and billing is finalized. To total spend across many sessions, list sessions and sum `cost` across the returned items, filtering by `companionId` or `externalClientId` as needed.

Tool connection states

When a session uses WebSocket-based tools, the platform opens a socket to each tool's server at session start. `functionStates` records that connection's lifecycle per tool — use it to answer "did my tool connect, when, and why not". It's returned by `GET /public/sessions/{sessionId}` as one flat array covering every tool in the session:

```
{
  "id": "sess_abc123",
  "functionStates": [
    {
      "name": "get_order_status",
      "state": "connected",
      "startedAt": "2026-08-14T10:15:02Z",
      "connectedAt": "2026-08-14T10:15:03Z"
    },
    {
      "name": "check_inventory",
      "state": "failed",
      "startedAt": "2026-08-14T10:15:02Z",
      "failedAt": "2026-08-14T10:15:07Z",
      "error": {
        "code": "connection_failed",
        "message": "WebSocket handshake timed out"
      }
    }
  ]
}
```

Field	Description
`name`	The tool's name.
`state`	The connection's current state.
`startedAt`	When the platform began connecting to the tool's server (ISO 8601).
`connectedAt`	When the socket connected successfully.
`failedAt`	When the connection attempt failed. `error` carries the cause as `{ code, message }`.
`canceledAt`	When the connection attempt was abandoned because the session ended first — for example, the user closed the tab before the tool finished connecting.

Only the timestamps that occurred are present. Tools with other execution types (HTTP, implicit) don't hold a session-long connection, so they never appear in `functionStates`. To control what happens to the *session* when a tool connects slowly or fails, see the tool's `connectionBehavior` setting.

> ⚠ This field was previously named `functionMetrics` and returned an object keyed by function library ID, with an array of tool entries under each ID. If your code reads `functionMetrics`, switch to `functionStates` — the entries themselves have the same fields, but they now come in a single array.

> ℹ Cost is finalized on session close. Poll `GET /public/sessions/{sessionId}` after a session ends, or read it from the session list, rather than expecting a value mid-session.

Deleting a session

To remove a session permanently — for example, when an end user asks you to delete their data — call `DELETE /public/sessions/{sessionId}`:

```
curl -X DELETE https://companion-api.napster.com/public/sessions/sess_abc123 \
  -H "X-Api-Key: $NAPSTER_API_KEY"
```

The API returns `200` once the session is deleted. Deleting a session also removes its conversation transcript and the memory stored from it, so the agent no longer recalls that conversation in future sessions with the same user.

> ⚠ Deletion is permanent. The session, its transcript, and its memory can't be recovered.

You can only delete sessions that are `closed` or `failed`. To end a session that's still running, close it first.

Error	Status	When
`SessionNotFound`	`404`	No session with this ID exists in your project.
`SessionNotClosed`	`400`	The session is still `pending` or `started`.
`SessionDeleteFailed`	`400`	The session couldn't be deleted. Nothing was removed — retry the request.

Webhooks

Get notified on your server when sessions start and end, and when companions are created, updated, or deleted

A webhook sends an HTTP `POST` to your server whenever something happens in your project — a session starts or ends, or a companion is created, updated, or deleted. Use webhooks to react in real time instead of polling the API: record usage as sessions close, or update your own records when a companion finishes generating.

Creating a webhook

You create webhooks in the developer dashboard, per project.

1 Open the Webhooks page

In the dashboard, select your project and go to **Webhooks**, then click **New webhook**.

2 Configure the webhook

Field	Description
Endpoint URL	The URL that receives the `POST` requests. Must be an `https://` URL that's reachable from the internet.
Events	The events that trigger this webhook. See Events.
Status	When enabled, the webhook receives events. Disable it to pause deliveries without deleting it.

3 Save the signing secret

Click **Create webhook**. The confirmation page shows the webhook's ID and its **signing secret** — a base64 string such as `q3Zr8WmN1xTbV0yLc7sKd2Pf9uHaE4jRgO6nYwB5tQA=`. Copy the secret and store it securely, for example in an environment variable on your server. You use it to verify that requests come from Napster.

> ⚠ The signing secret is shown only once. If you lose it, delete the webhook and create a new one.

> ℹ A project can have one enabled webhook at a time. You need the organization **Admin** role to create, edit, or delete webhooks.

Events

Event	When it's sent
`session.started`	A session connected and started.
`session.closed`	A session that started has ended. Sessions that fail before starting don't send this event.
`companion.created`	A companion was created. Its `status` is `pending` while the avatar is generated.
`companion.updated`	A companion changed, including each `status` change while its avatar is generated.
`companion.deleted`	A companion was deleted.

Payload format

Every event shares the same envelope. The event itself is in `data`, under `session` or `companion`:

```
{
  "eventId": "67d67303ec4f8672fefb704952206d48",
  "eventType": "companion.created",
  "timestamp": 1790685304,
  "organizationId": "org_abc123",
  "projectId": "proj_abc123",
  "data": {
    "companion": { }
  }
}
```

Field	Description
`eventId`	A unique ID for the event. Use it to detect duplicate deliveries.
`eventType`	The event name, for example `session.closed`.
`timestamp`	When the event happened, as a Unix timestamp in seconds.
`organizationId`	The organization the event belongs to.
`projectId`	The project the event belongs to.
`data`	The event payload — `data.session` for session events, `data.companion` for companion events.

Session events

`session.started` and `session.closed` describe the session in `data.session`:

```
{
  "eventId": "9f1c2e7a-4b3d-4f6e-8a1b-2c3d4e5f6a7b",
  "eventType": "session.closed",
  "timestamp": 1790690142,
  "organizationId": "org_abc123",
  "projectId": "proj_abc123",
  "data": {
    "session": {
      "id": "sess_xyz789",
      "type": "webrtc",
      "companionId": "comp_abc123",
      "externalClientId": "user_42",
      "createdAt": 1790689801,
      "startedAt": 1790689803,
      "closedAt": 1790690142,
      "reason": "connection_closed"
    }
  }
}
```

Field	Description
`id`	The session ID. Use it with `GET /public/sessions/{sessionId}` to fetch the full session, including cost and transcript.
`type`	The channel: `webrtc`, `websocket`, `voip`, or `sip`. Kiosk sessions report `websocket`.
`companionId`	The companion the session used.
`externalClientId`	The `externalClientId` you passed when opening the session, or `null`.
`createdAt` / `startedAt`	When the session was created and when it started, as Unix timestamps in seconds.
`closedAt`	When the session ended. `session.closed` only.
`reason`	Why the session ended. `session.closed` only. See Close reasons.

> `session.closed` doesn't include the session's cost. Billing is finalized as the session closes — fetch the session with its `id` to read `cost`.

Companion events

`companion.created`, `companion.updated`, and `companion.deleted` describe the companion in `data.companion`. For `companion.deleted`, it's the companion as it was before deletion. Companion events cover the companions your project owns, not the Napster stock catalog.


```
{
  "eventId": "e10b49a2b70d4657d728873dd473874c",
  "eventType": "companion.updated",
  "timestamp": 1790685858,
  "organizationId": "org_abc123",
  "projectId": "proj_abc123",
  "data": {
    "companion": {
      "id": "comp_abc123",
      "firstName": "Maya",
      "lastName": "Bennett",
      "previewUrl": "https://cdn.napster.com/companions/comp_abc123/preview.jpg",
      "videoLoopUrl": "https://cdn.napster.com/companions/comp_abc123/video.mp4",
      "ethnicity": "multiracial",
      "gender": "female",
      "headline": "Helping people find clarity, momentum, and practical next steps.",
      "tags": {},
      "status": "completed",
      "createdAt": 1790685277,
      "externalClientId": null,
      "versions": ["v1"]
    }
  }
}
```

Field	Description
`id`	The companion ID.
`firstName` / `lastName`	The companion's name.
`previewUrl`	URL of the companion's preview image.
`videoLoopUrl`	URL of the companion's idle video loop, or `null` until generation finishes.
`ethnicity` / `gender`	The companion's appearance attributes.
`headline`	The companion's short description.
`tags`	Key–value tags on the companion.
`status`	Generation status: `pending`, `generationCompleted`, `readyToUse`, or `completed`.
`createdAt`	When the companion was created, as a Unix timestamp in seconds.
`externalClientId`	The external client ID associated with the companion, or `null`.
`versions`	The avatar model versions available for the companion, for example `["v1"]`. Empty until generation finishes.

A new companion sends `companion.created` with `status: "pending"`, then one `companion.updated` for each status change as it generates: `generationCompleted`, `readyToUse`, and finally `completed`. You can attach it to an agent from `readyToUse`; edit it once it's `completed`.

Verifying requests

Every delivery is signed with your webhook's signing secret. Verify the signature before trusting a request, so you only act on events that come from Napster.

Each request carries an `X-Webhook-Signature` header: the HMAC-SHA256 of the raw request body, keyed with your signing secret and hex-encoded.

```
POST /webhooks/napster HTTP/1.1
Content-Type: application/json
X-Webhook-Signature: 077820582aecb562633e656670fa842e81a391a0daebaa85f3bf71d39e933b9b
```

To verify a request, compute the same HMAC over the body you received and compare it to the header:

> NODE.JS

```
import express from "express";
import { createHmac, timingSafeEqual } from "node:crypto";

const app = express();

// express.raw keeps the body as the exact bytes Napster signed
app.post("/webhooks/napster", express.raw({ type: "application/json" }), (req, res) => {
  const expected = createHmac("sha256", process.env.NAPSTER_WEBHOOK_SECRET)
    .update(req.body)
    .digest("hex");
  const received = req.get("X-Webhook-Signature") ?? "";

  if (
    received.length !== expected.length ||
    !timingSafeEqual(Buffer.from(received), Buffer.from(expected))
  ) {
    return res.sendStatus(401);
  }

  const event = JSON.parse(req.body.toString("utf8"));
  res.sendStatus(200);
  // Process event.eventType after responding
});
```

PYTHON

```python
import hashlib
import hmac
import os

from flask import Flask, abort, request

app = Flask(__name__)

@app.post("/webhooks/napster")
def napster_webhook():
    expected = hmac.new(
        os.environ["NAPSTER_WEBHOOK_SECRET"].encode(),
        request.get_data(),
        hashlib.sha256,
    ).hexdigest()
    received = request.headers.get("X-Webhook-Signature", "")

    if not hmac.compare_digest(expected, received):
        abort(401)

    event = request.get_json()
    # Process event["eventType"] after responding
    return "", 200
```

⚠ Sign the raw body exactly as received. Parsing the JSON and serializing it again changes the bytes, and the signature won't match. Use the signing secret exactly as shown in the dashboard: the base64 string itself is the key, so don't decode it.

Handling deliveries

A delivery succeeds when your endpoint responds with any `2xx` status within 10 seconds. Any other status, a network error, or a timeout counts as a failed attempt.

A failed attempt is retried up to 3 more times with exponential backoff: about 1, 2, and 4 seconds between attempts. If every attempt fails, the delivery is marked failed and isn't retried again. You can still inspect it in the delivery history.

- **Respond quickly with a `2xx` status.** Do any slow work after you respond, so the delivery isn't treated as failed.

- **Expect duplicates.** If your endpoint handles a request but responds too late, the retry delivers the same event again. Use `eventId` to skip events you've already processed.
- **Don't rely on order.** Use `timestamp` to order events for the same session or companion.

Delivery history

The dashboard keeps a history of every delivery for each webhook. Open a webhook to see each delivery's event, status, and timestamp. Open a delivery to see the request that was sent — its body and headers — and each attempt, with the response status, body, and latency your endpoint returned. Response bodies are stored up to 4 KB.

Use it to debug your endpoint: if deliveries fail, the response your endpoint returned is recorded on each attempt.

Napster.

SECTION 05

SDKs

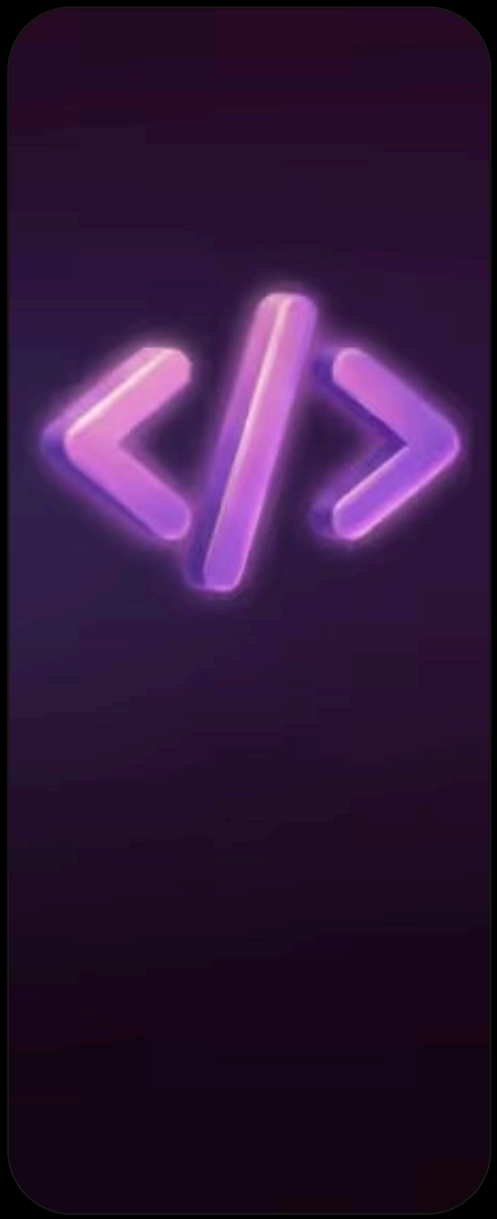
The Web SDK and Edge MCP.

WEB SDK

Overview 191
Installation 193
Initialization 195
Configuration 198
Features 204
Persistence 211
Events 219
Messaging 222
Methods 225
Examples 230
Troubleshooting 234

EDGE MCP

Overview 238
Installation 241
Tools 244
Resources 247
Testing 252



Overview

Embed a real-time audio and video Omniagent in the browser

The Web SDK embeds an Omniagent directly in your web application — a real-time agent your users can see, hear, and talk to. It ships a ready-made UI — the avatar widget, built-in controls, and tooltips — that you drop into any page, and a typed API for driving the session from your own code.

Start it with `init`, and on multi-page sites, **persistence** can keep one session alive as the user navigates — an always-on agent that survives page changes.

The SDK is framework-agnostic and works with React, Vue, Angular, or vanilla JavaScript. It ships with full TypeScript support, tree-shakeable exports, and zero external dependencies in standalone mode.

The package is published to npm as `@touchcastllc/napster-companion-api`.

> New to the Napster API? The **Quickstart** walks the whole path end to end — create an Omniagent, mint a connection token on your backend, and embed it with this SDK — in a few minutes.

Supported environments

Browsers

Chrome, Firefox, Safari, Edge

React

16 and up

Vue

2 and up

Angular

12 and up

Vanilla JS

No build tools required

How it fits together

The Web SDK is the browser client for the WebRTC channel. A typical integration has three parts:

1. **Your backend** creates a session and gets a connection token. See WebRTC channel for creating the session, and Authentication for the API key.
2. **Your frontend** passes that token to the SDK and mounts the widget.
3. **The SDK** opens the WebRTC connection, renders the avatar, and exposes an instance you use to control the session.

> The token is short-lived and scoped to a single session. Always mint it on your server — never ship your API key to the browser.

Installation

Add the Web SDK to your project with the ESM or standalone build

The Web SDK ships in two build types. Use the ESM build for any project with a bundler; use the standalone build to drop the SDK into a plain HTML page with a single script tag.

Choose a build

	ESM build	Standalone build
Install	`npm install`	`<script>` tag
Best for	React, Vue, Angular, any bundler	Plain HTML, no build step
Dependencies	`@reduxjs/toolkit` peer dependency	All bundled
Tree-shaking	Yes	No
Entry point	`NapsterCompanionApiSdk` import	`window.napsterCompanionApiSDK`

ESM build

Install the SDK and its peer dependency:

```
npm install @touchcastllc/napster-companion-api @reduxjs/toolkit
```

`@reduxjs/toolkit` is a required peer dependency. Import the SDK and its stylesheet in your application:

```
import { NapsterCompanionApiSdk } from "@touchcastllc/napster-companion-api";
import "@touchcastllc/napster-companion-api/styles";
```

The stylesheet can also be loaded via a `<link>` tag or CSS `@import` if you prefer not to import it in JavaScript:

```
<link rel="stylesheet" href="https://cdn.jsdelivr.net/npm/@touchcastllc/napster-companion-api@latest/lib/index.css" />
```

Standalone build

Add a single script tag to your HTML. All dependencies are bundled — no build tools required.

```
<script src="https://cdn.jsdelivr.net/npm/@touchcastllc/napster-companion-api@latest/lib/index.standalone.js"></script>
```

The SDK attaches itself to `window.napsterCompanionApiSDK`. The stylesheet is included in the standalone bundle, so no separate CSS import is needed.

Get a token

Before you can initialize the SDK you need a connection token. Tokens are short-lived and minted on your server — never expose your API key in the browser.

- To create a session from an existing Omniagent, see **Create a session** on the WebRTC channel page.
- To authenticate your backend, see **Authentication**.

Initialization

Connect to a session with init and mount the widget

Start the SDK with `init` — it connects immediately and mounts the widget. Its config is a `NapsterCompanionApiConfig`.

Connect

`init` starts a session the moment you call it — it mounts the avatar and begins connecting right away. Pass the token from the connection response and a container element to mount into.

> ESM

```
import { NapsterCompanionApiSdk } from "@touchcastllc/napster-companion-api";
import "@touchcastllc/napster-companion-api/styles";

const instance = await NapsterCompanionApiSdk.init(token, {
  mountContainer: "#avatar-container",
});
```

> STANDALONE

```
<div id="avatar-container"></div>

<script>
  window.napsterCompanionApiSDK.init(token, {
    mountContainer: "#avatar-container",
  });
</script>
```

`init(token, config?)` takes two arguments:

Argument	Type	Required	Description
`token`	`string`	Yes	The connection token from your backend. Pass it as-is; you don't need to decode it.
`config`	`NapsterCompanionApiConfig`	No	Options for placement, avatar style, features, and callbacks. See Configuration.

`mountContainer` accepts a CSS selector string or an `HTMLElement`. If you omit it, the widget mounts on `document.body`. If a selector matches no element, the SDK warns and falls back to `document.body`.

What you get back

`init` returns a `Promise<NapsterCompanionApiInstance>` — the object you use to drive the session. It resolves to the live instance once the SDK has mounted and started connecting.

```
import type { NapsterCompanionApiInstance } from "@touchcastllc/napster-companion-api";

const instance: NapsterCompanionApiInstance = await NapsterCompanionApiSdk.init(token, {
  mountContainer: "#avatar-container",
});
```

The avatar video comes up shortly after connecting — use the `onAvatarReady` callback to know when it's actually rendering, and read the `sessionId` getter for the live session identifier. See Methods for everything the instance exposes.

Tear down

Call `destroy()` when the widget leaves the page. It closes the WebRTC connection, removes the DOM nodes, and releases the microphone — plus any screen-share or face-tracking camera capture, if you were using those.

```
instance.destroy();
```

In component frameworks, call it from your unmount hook — see the framework examples. Failing to call `destroy()` leaves the connection and media tracks open.

Configuration

The full config object passed to init — placement, avatar style, features, callbacks, and persistence

The second argument to `init` is a `NapsterCompanionApiConfig` object. Every field is optional — the SDK applies sensible defaults. This page is the full reference, grouped by concern.

All options

Every top-level field on `NapsterCompanionApiConfig`. All are optional; jump to a section for the details.

Option	Type	Default	Description
`mountContainer`	`HTMLElement | string | null`	`document.body`	Where to mount the SDK. See Placement and layout.
`position`	`Position`	`"bottom-right"`	Corner the floating widget anchors to. See Placement and layout.
`layout`	`"fixed" | "inline"`	`"fixed"`	Float as a corner widget, or fill `mountContainer`. See Placement and layout.
`avatarStyle`	`avatarStyleConfig`	`{ view: "round" }`	Avatar shape and border. See Appearance.
`style`	`StyleObject`	—	Inline styles on the SDK root container. See Appearance.
`className`	`string`	—	CSS class(es) on the SDK root container. See Appearance.
`features`	`FeatureConfig`	—	Toggle and tune built-in features. See Features.
`persistence`	`PersistenceOptions`	—	Keep one session alive across page navigation. See Persistence.

Option	Type	Default	Description
`debug`	`boolean`	`false`	Verbose SDK logging to the console. See Debug.
Callbacks	functions	—	`onReady`, `onError`, `onData`, `onAvatarReady`, `onInactivityStatusChange`, `onDestroy`, `onFeaturesUpdate`, `onScreenShareStateChange`. See Callbacks.

Placement and layout

Option	Type	Default	Description
`mountContainer`	`HTMLElement | string | null`	`document.body`	Where to mount the SDK. A CSS selector or an element.
`position`	`Position`	`"bottom-right"`	Where the floating widget anchors to the viewport. Ignored when `layout` is `"inline"`.
`layout`	`"fixed" | "inline"`	`"fixed"`	`"fixed"` floats the avatar as a corner widget; `"inline"` renders it inside `mountContainer` and fills it.

Position values

`position` accepts any value from the `Position` enum:

- `"bottom-right"` (default)
- `"bottom-center"`
- `"bottom-left"`
- `"top-right"`
- `"top-center"`
- `"top-left"`
- `"center"`

```
await NapsterCcompanionApiSdk.init(token, {
  mountContainer: "#my-container",
  position: "bottom-right",
});
```

Inline layout

By default the avatar is a floating corner widget (`layout: "fixed"`). Set `layout: "inline"` to render the avatar **inside your own container** and fill it — you control placement, size, and shape by styling that container. `position` is ignored in inline mode.

```
// <div id="avatar-box" style="width: 320px; height: 480px"></div>
await NapsterCcompanionApiSdk.init(token, {
  mountContainer: "#avatar-box",
  layout: "inline",
  avatarStyle: { view: "rectangle" },
});
```

Sizing rules in inline mode:

- The avatar fills your container's width and height.
- If your container has **no explicit height**, an internal **4:5** ratio is used (matching the floating widget's proportions) so the avatar is never invisible.
- `view: "round"` always renders a circle that fits the smaller side and centers, regardless of the container's aspect ratio.
- Video is cropped to cover — never stretched or distorted.

Appearance

Option	Type	Default	Description
`avatarStyle`	`avatarStyleConfig`	`{ view: "round" }`	Avatar shape and border.
`style`	`StyleObject`	—	Inline styles applied to the SDK root container. A partial `CSSStyleDeclaration`.
`className`	`string`	—	CSS class name(s) added to the SDK root container. Useful for theming.

avatarStyle

Field	Type	Default	Description
`view`	`"round" | "rectangle" | "silhouette"`	`"round"`	Avatar shape. Use `"rectangle"` for custom styling that needs a rectangular container; use `"silhouette"` to chroma-key a green-screen stream (see green screen).
`borderWidth`	`string`	—	Border width, as a CSS value.
`borderColor`	`string`	—	Border color, as a CSS color string.
`borderStyle`	`string`	—	Border style, e.g. `"solid"`, `"dashed"`.

Custom position with styles

You can override placement with `style` and `className` on the root container:

```
await NapsterCompanionApiSdk.init(token, {
  mountContainer: "#my-container",
  position: "bottom-right",
  style: {
    top: "20px",
    left: "20px",
    zIndex: "1000",
  },
  className: "my-custom-class",
});
```

Features

The `features` object toggles built-in behaviors. Each feature is a small object with its own `enabled` flag plus optional tuning values.

> Every feature — what it does, its defaults, and its per-feature options — is documented on the Features page.

Callbacks

All callbacks are optional. They are covered in full on the Events page.

Callback	Signature	Fires when
`onReady`	`() => void`	The SDK has finished initialization and is ready to render.
`onError`	`(error: Error) => void`	A runtime or network error occurs.
`onData`	`(data: EventMessage) => void`	Data arrives over the SDK's connection.
`onAvatarReady`	`(isReady?: boolean) => void`	The visual avatar is fully prepared and rendering.
`onInactivityStatusChange`	`(isInactive: boolean) => void`	The inactivity status changes.
`onDestroy`	`() => void`	The SDK is destroyed.
`onFeaturesUpdate`	`(features: FeatureConfig) => void`	Feature configuration changes at runtime.
`onScreenShareStateChange`	`(isSharing: boolean) => void`	Screen sharing starts or stops.

Persistence

The `persistence` object keeps one session alive across page navigation on a multi-page site:

```
NapsterCompanionApiSdk.init(token, {
  persistence: { enabled: true },
});
```

It has its own page — the behavior (iframe wrap, break-out rules, deferred wrap, programmatic navigation) and the full option table are covered in Persistence.

Debug

Option	Type	Default	Description
`debug`	`boolean`	`false`	Enable verbose SDK logging to the console. Warnings and errors always print; the detailed channel logs appear only with `debug: true`. See Troubleshooting → Debug mode.

Features

Every built-in feature — inactivity timeout, loader, controls, picture-in-picture, screen sharing, background removal, and disclaimer

Features are built-in behaviors controlled through the `features` **config object**. Each feature has an `enabled` flag and, in some cases, tuning options — several are **on by default**, so the table below is also the list of what to turn off. You can also toggle features at runtime — see **feature control methods**.

Feature key	Default	Purpose
`inactiveTimeout`	**on** — 60 s idle, 30 s countdown	Disconnect after a period of inactivity, with a countdown.
`showSDKLoader`	**on**	Loading overlay while the avatar loads.
`controls`	**on**	Built-in controls block (mute / volume / screen-share / end).
`pictureInPicture`	off	Auto pop-out the avatar when the user switches tabs.
`screenShare`	off	Let the user share their screen with the avatar.
`backgroundRemoval`	**on**	Remove the avatar's background.
`disclaimer`	**on**	Small disclaimer text shown over the widget.

Inactivity timeout

Automatically disconnect the avatar after a period of user inactivity. Users see a countdown notification before disconnection.

```
await NapsterCompanionApiSdk.init(token, {
  features: {
    inactiveTimeout: {
      enabled: true,
      duration: 60000, // idle time before the countdown, in ms
      countdown: 10,   // countdown length before disconnect, in seconds
    },
  },
});
```

Option	Type	Default	Description
`enabled`	`boolean`	—	Turn the timeout on.
`duration`	`number`	`60000`	Idle time before the countdown starts, in milliseconds. Max `180000` (3 minutes).
`countdown`	`number`	`30`	Countdown shown before disconnecting, in seconds. Max `60`.

Use `onInactivityStatusChange` to react when the inactivity status flips.

Loading screen

Display a loading overlay while the avatar assets load. It is enabled by default. Customize the background color, loader color, and animation type, or supply your own CSS class.

```
await NapsterCompanionApiSdk.init(token, {
  features: {
    showSDKLoader: {
      enabled: true,
      bgColor: "#ffffff",         // overlay background color
      color: "#000000",           // loader animation color
      type: "spinner",            // "spinner" | "pulse"
      className: "custom-loader", // optional custom class
    },
  },
});
```

Option	Type	Description
`enabled`	`boolean`	Show the loader overlay.
`bgColor`	`string`	Background color of the overlay.
`color`	`string`	Color of the loader animation.
`type`	`"spinner" | "pulse"`	Loader animation style.
`className`	`string`	CSS class added to the loader element for custom styling.

With `className` you can override the loader's appearance entirely:

```
.my-brand-loader {
  width: 60px !important;
  height: 60px !important;
  border-width: 5px !important;
  border-style: solid !important;
  border-color: rgba(99, 102, 241, 0.2) !important;
  border-top-color: #6366f1 !important;
  border-radius: 50% !important;
  animation: spin 0.8s linear infinite !important;
}

@keyframes spin {
  to { transform: rotate(360deg); }
}
```

Built-in controls

The controls block is the mute / volume / screen-share / end buttons rendered over the avatar. It is **enabled by default**. The volume control toggles the sound on and off; to set a specific volume level, use `setAudioVolume` from your own UI. Disable the block entirely when your application provides its own controls using the instance methods (`muteMic`, `setAudioVolume`, `stopAvatarTalking`, `destroy`, and so on).

```
await NapsterCompanionApiSdk.init(token, {
  features: {
    controls: {
      enabled: false, // hide built-in buttons, render your own
    },
  },
});
```

You can toggle it at runtime:

```
instance.updateFeatureConfig("controls", { enabled: false });
// later
instance.enableFeature("controls");
```

Picture-in-picture

Pop the avatar into a Document Picture-in-Picture window when the user switches browser tabs, so the conversation stays visible while they work elsewhere. The PiP window mirrors the live avatar video and renders the built-in controls. Closing the PiP window or returning to the original tab restores the in-page avatar.

```
await NapsterCompanionApiSdk.init(token, {
  features: {
    pictureInPicture: {
      enabled: true,
      width: 400,  // optional, default 400
      height: 300, // optional, default 300
    },
  },
});
```

Option	Type	Default	Description
`enabled`	`boolean`	—	Enable auto picture-in-picture.
`width`	`number`	`400`	PiP window width in pixels.
`height`	`number`	`300`	PiP window height in pixels.

The auto-PiP trigger requires Chrome or Edge 116+ and:

- A secure context (HTTPS).
- Active media playback with audio.
- The site has met the browser's Media Engagement Index threshold.

If the requirements aren't met, the avatar stays in-page when the tab loses focus — no errors are thrown. To check support programmatically, use the exported `checkPiPSupport` helper.

PiP support helpers

The SDK exports two helpers for detecting Document Picture-in-Picture support:

```
import { checkPiPSupport, resetPiPSupportCache } from "@touchcastllc/napster-companion-api";

if (checkPiPSupport()) {
  // the browser supports Document Picture-in-Picture
}
```

`checkPiPSupport()` returns whether the browser supports the API; the result is cached. `resetPiPSupportCache()` clears that cache, which is mainly useful in tests.

Screen sharing

Share the user's screen with the avatar in real time. The SDK captures the display, renders each frame to an internal canvas, and streams it over WebRTC so the avatar can "see" what the user sees. Fully browser-based — no plugins or extensions.

```
await NapsterCompanionApiSdk.init(token, {
  features: {
    screenShare: {
      enabled: true,
    },
  },
});
```

> `features.screenShare.enabled` must be `true` for the screen-share controls to appear and for `isScreenShareSupported` to return `true`.

Start and stop sharing from your own UI with the screen-sharing methods:

```
await instance.startScreenShare(); // opens the browser's screen picker
instance.stopScreenShare();
await instance.toggleScreenShare();
```

The SDK sends `start_video` / `stop_video` commands to the avatar automatically. Screen sharing stops automatically when the user calls `destroy()`, when the WebRTC connection closes, or when the user clicks the browser's native "Stop sharing" button. Listen for `onScreenShareStateChange` to keep your UI in sync.

Background removal

Remove the avatar's background so it composites over your page. Enabled by default.

```
await NapsterCompanionApiSdk.init(token, {
  features: {
    backgroundRemoval: {
      enabled: true,
    },
  },
});
```

For chroma-keying a green-screen stream, see `avatarStyle.view: "silhouette"` and the green screen background channel option.

Disclaimer

A small disclaimer shown over the widget — legal text or usage hints. It is **enabled by default** with the default text; set `enabled: false` to remove it, or provide your own `text`.

```
await NapsterCompanionApiSdk.init(token, {
  features: {
    disclaimer: {
      enabled: true,
      text: "AI-generated responses. May contain errors.",
      color: "#666666", // optional
    },
  },
});
```

Option	Type	Description
`enabled`	`boolean`	Show the disclaimer.
`text`	`string`	The disclaimer text. Defaults to `"This is an AI avatar. Check important info."` when omitted.
`color`	`string`	CSS color for the text.

You can change it at runtime:

```
instance.updateFeatureConfig("disclaimer", { text: "New disclaimer text" });
```

Persistence

Keep one session alive across page navigation on a multi-page site

On a traditional multi-page site, clicking a link triggers a full page reload — which tears down the running session, so the agent would restart and reconnect on every page. Persistence keeps **one** session alive as the user moves around the site.

It works by loading your site into a full-window iframe — an iframe of your own site — while the avatar stays outside it, in the top-level page. This happens when a visitor **starts a session with the agent**, not when the SDK loads. From then on they browse inside that frame, so pages can change without the document holding the session ever reloading.

Before you enable it

Your site has to allow itself to be framed. The iframe is same-origin, but your own response headers still decide whether it renders: `X-Frame-Options: SAMEORIGIN` is fine, while `X-Frame-Options: DENY`, or a Content-Security-Policy with `frame-ancestors 'none'`, blocks it — and those are common defaults on security-hardened sites.

Then check the known limitations at the end of this page.

Enable it

Turn persistence on with the `persistence` object when you initialize the SDK:

```
NapsterCompanionApiSdk.init(token, {
  persistence: { enabled: true },
});
```

Include the same snippet on every page of the site. When persistence loads a page into the frame, the copy of the snippet running inside that frame detects it and exits without doing anything — it never starts a second session — so it's safe everywhere.

> ⚠️ By default a session persists only across pages on the **same domain** — the domain it started on, exactly as it appears in the address bar. Those are the pages the SDK loads in the iframe. Click a link to anywhere else — another site, or one of your own subdomains like `auth.example.com` — and the SDK doesn't put it in the iframe: it ends the session and navigates the whole tab there. You can allow specific domains inside the frame with `persistence.include`, at a cost. See Break-out rules.

Options

Field	Type	Default	Description
`persistence.enabled`	`boolean`	—	Turn persistence on. Required — there is no boolean shorthand. When on, `mountContainer` is ignored (the avatar must live in the top document).
`persistence.iframeOnNavigate`	`boolean`	`false`	Wait for the visitor's first link click before loading the site into the frame, keeping the page they started on fully native. Off by default — the frame appears as soon as the session starts. See When the frame appears.
`persistence.exclude`	`{ urls?: string[] }`	—	Pages of your own site excluded from persistence — they open as a normal top-level navigation, which ends the session. See Break-out rules.
`persistence.include`	`{ domains?: string[] }`	—	Other domains allowed to load **inside** the frame, so a link to them keeps the session alive. Everything off your domain breaks out by default; this is the exception list. See Allowing another domain inside the frame.

Field	Type	Default	Description
`persistence.navigationProgress`	`boolean | { color?, height? }`	`false`	Show a thin progress bar at the top of the page while a page loads into the site frame. Navigations inside the frame drive no browser chrome (no tab spinner), so without an indicator a click the engine took over looks dead for the whole load. `true` for default styling, or restyle with `{ color, height }` (defaults `#be369d`, `3px` — also overridable via the `--np-persist-progress-color` / `--np-persist-progress-height` CSS custom properties). Leave off if your site shows its own navigation indicator.
`persistence.iframeAllowAttribute`	`string`	`"autoplay; clipboard-write"`	Value for the site iframe's `allow` attribute (Permissions Policy).
`persistence.syncHistory`	`boolean`	`true`	Mirror the iframe's path into the address bar and follow back/forward. Same domain only.

When the frame appears

Persistence works by putting your site inside an iframe. `iframeOnNavigate` controls **when** that happens — which gives a session two possible states, and they behave differently:

	`iframeOnNavigate: false` **— the default**	`iframeOnNavigate: true`
When the iframe appears	Straight away, as soon as the session starts	Only when the visitor first clicks a link
The page they started the session on	Inside the iframe from the start	Never inside the iframe — it renders exactly as it would without persistence
...and while they're still on it	Everything is safe. The iframe is already in place, so navigation happens inside it	Only link clicks are safe. Anything else that navigates — your code, a redirect — ends the session

Creating the frame at session start is the default because it protects everything from the first moment: there is no window where a redirect or a script can end the session.

Set `iframeOnNavigate: true` to keep that page fully native until the visitor actually navigates — nothing about how it renders changes. The cost is that the page is unprotected until the first link click: anything else that navigates there must go through the persist-navigate event, or the session ends.

```
NapsterCompanionApiSdk.init(token, {
  persistence: {
    enabled: true,
    iframeOnNavigate: true, // stay out of the frame until the first link click
  },
});
```

What survives a navigation

The rule behind this whole table in one line: **the SDK sees what the visitor does — clicks and form submissions. It cannot see a navigation your code performs.** Once the page is inside the iframe that stops mattering, because the frame absorbs the navigation and the top document (where the avatar lives) never reloads.

What navigates	Before the frame exists	Inside the frame
Visitor clicks a link on your domain	✅ Loads into the frame	✅ Loads in the frame
Visitor clicks a link to another domain	❌ Ends — by design, unless the domain is in `include`	❌ Ends — by design, unless the domain is in `include`
Visitor submits a form to your domain	✅ Routed into the frame	✅ Submits inside the frame
Your code: `location.assign()`, `location.href = …`, `location.replace()`	❌ **Ends** — use the event	✅ Only the frame navigates
Your code: `form.submit()`	❌ **Ends** — use `requestSubmit()`	✅ Submits inside the frame
A server redirect, or `<meta http-equiv="refresh">`	❌ **Ends**	✅ The frame follows it
SPA router (`history.pushState`)	✅ No page load at all	✅ No page load at all
`target="_blank"` or `window.open()`	✅ Opens a new tab; this one stays put	✅ Opens a new tab; this one stays put

Under the default (`iframeOnNavigate: false`), only the second column applies — the iframe exists from the moment the session starts, so there is no unprotected page. The first column describes the opt-in deferred mode (`iframeOnNavigate: true`), where the page the session started on stays outside the frame until the first link click.

Back and forward are handled inside the frame: the iframe follows them, and the address bar stays in step as long as `syncHistory` is on.

Navigating from code

This section fixes the ❌ rows in the table above — they apply while a page is still outside the frame, which under the default never happens, and they're all one problem. A click produces an event the SDK can intercept; `location.assign()`, `location.href = …` and `location.replace()` produce nothing. They're instructions straight to the browser, with no event to listen for, so on a page that is not inside the frame yet the tab simply leaves and the session goes with it.

The fix is to announce the navigation instead of performing it. Dispatch the exported `PERSIST_NAVIGATE_EVENT` on `window`:

```
import { PERSIST_NAVIGATE_EVENT } from "@touchcastllc/napster-companion-api";

function navigate(url) {
  const e = new CustomEvent(PERSIST_NAVIGATE_EVENT, {
    detail: { url },
    cancelable: true,
  });
  window.dispatchEvent(e);
  // If nothing handled it (no active persisted session, or an excluded URL),
  // fall back to a normal navigation.
  if (!e.defaultPrevented) location.assign(url);
}
```

The SDK listens while the page is still native, creates the frame on the event, loads `detail.url`, and calls `preventDefault()` so you know it was handled. Excluded URLs (`persistence.exclude`) are ignored so they still navigate normally.

Forms submitted from code

The `form.submit()` row fails for a different reason than the rest. A form the visitor submits is safe — the browser fires a `submit` event first, and the SDK routes the submission into the frame. But `form.submit()` called from code fires **no** event at all, by design, so there is nothing for the SDK to hear.

Call `form.requestSubmit()` instead. It performs the same submission but fires the event first, exactly as if the visitor had pressed the button — which is all the SDK needs:

```
form.requestSubmit(); // not form.submit()
```

`requestSubmit()` also runs the browser's own validation, which `submit()` skips.

Break-out rules

A link that leaves your domain breaks out by default, with no configuration: the SDK ends the session and navigates the whole tab to it. That covers any other site, and any subdomain of your own — `auth.example.com` is a different domain from `www.example.com` as far as the browser is concerned, so it breaks out on its own. To keep a specific domain inside the frame instead, see Allowing another domain inside the frame.

What you configure are **your own pages** that should end the session instead of running inside the frame — a checkout flow, a sign-out route. List them in `persistence.exclude`:

```
NapsterCompanionApiSdk.init(token, {
  persistence: {
    enabled: true,
    exclude: {
      urls: ["/checkout", "/signout"], // paths (and everything under them), or absolute
https URLs
    },
  },
});
```

When a link matches, the SDK ends the session and navigates the whole tab to that URL.

A `urls` entry matches its exact path and everything under it — at **path boundaries**, not by spelling. `"/check"` matches `/check`, `/check/out`, and `/check?step=1`, but **not** `/checkout` — a different page that merely shares the prefix. So excluding one route never accidentally excludes look-alike siblings. End an entry with `/` to say "everything under this directory" explicitly.

> ⚠ These rules are applied to the URL of the link the visitor clicks, and nothing after that. If a page on your domain redirects somewhere else, the SDK never saw it coming — a `/login` route that redirects to `auth.example.com` passes the check, because `/login` is on your domain. The other domain then loads **inside the frame**, where many login pages refuse to render at all. Exclude the **starting path** so the click leaves the frame up front: `exclude: { urls: ["/login"] }`.

Allowing another domain inside the frame

Sometimes a domain that isn't yours belongs in the session — a docs site you also run, a partner's help centre. List it in `persistence.include` and links to it load in the frame instead of ending the session:

```
NapsterCompanionApiSdk.init(token, {
  persistence: {
    enabled: true,
    include: {
      domains: ["docs.example.com", "partner.io"],
    },
  },
});
```

An entry matches the exact host and any subdomain of it. `"example.com"` covers `docs.example.com` and `shop.example.com`, but not `notexample.com` — a different site that merely ends with the same letters.

> ⚠ The other site has to permit framing at all. A site that sends `X-Frame-Options: DENY` or a restrictive `frame-ancestors` renders nothing, and the SDK cannot detect that — the visitor sees a blank frame. Test every domain you add before shipping it, and only add domains you control or have agreed with.

Known limitations

Running your pages inside a frame costs some ordinary page behavior. None of these break a site, but check the ones that touch your flows.

Browser permissions narrow to what the frame allows. The framed page gets only what `iframeAllowAttribute` grants, and the default is `"autoplay; clipboard-write"`. If your pages use the camera, microphone, geolocation or payment, add them — otherwise those features stop working once the frame exists:

```
persistence: {
  enabled: true,
  iframeAllowAttribute: "autoplay; clipboard-write; camera; microphone",
}
```

Code that checks for framing behaves differently. Inside the frame, `window.top !== window.self`. Frame-busting scripts on your own site will fight the frame — remove them, or scope them to cross-origin parents. Third-party widgets that refuse to run framed (some SSO, payment and consent flows) need their pages in `persistence.exclude`, so they open as a normal top-level navigation instead.

Events

Lifecycle callbacks and the incoming data your client receives

Events are the SDK's *observe* surface: callbacks you register in config to follow the session as it happens. For the other direction — sending messages and commands to the agent — see Messaging.

Lifecycle callbacks

Register these in the config object. All are optional.

```
const instance = await NapsterCompanionApiSdk.init(token, {
  onReady: () => console.log("SDK ready"),
  onError: error => console.error("SDK error:", error),
  onAvatarReady: isReady => console.log("Avatar ready:", isReady),
  onInactivityStatusChange: isInactive => console.log("Inactive:", isInactive),
  onScreenShareStateChange: isSharing => console.log("Sharing:", isSharing),
  onDestroy: () => console.log("SDK destroyed"),
  onData: data => console.log("Data:", data),
});
```

Callback	Signature	Fires when
`onReady`	`() => void`	The SDK has finished initialization and is ready to render.
`onError`	`(error: Error) => void`	A runtime or network error occurs.
`onAvatarReady`	`(isReady?: boolean) => void`	The visual avatar is fully prepared and rendering. Fires exactly once.
`onInactivityStatusChange`	`(isInactive: boolean) => void`	The user's inactivity status changes. Pairs with the inactivity timeout feature.
`onDestroy`	`() => void`	The SDK is destroyed, via `destroy()` or the built-in end control.
`onData`	`(data: EventMessage) => void`	A message arrives over the data channel. See incoming data.
`onFeaturesUpdate`	`(features: FeatureConfig) => void`	Feature configuration changes at runtime.
`onScreenShareStateChange`	`(isSharing: boolean) => void`	Screen sharing starts or stops.

> Prefer `onAvatarReady` over reacting to the raw `ready` state when you need to know the avatar video is actually visible. Under deferred video the session goes live on audio first and the avatar renders a moment later — `onAvatarReady` fires once, when video is up.

Incoming data

`onData` receives each message the avatar sends over the data channel as an `EventMessage`:

```
interface EventMessage {
  event: string;
  data?: {
    state?: string;
    message?: { action?: string; content?: string; role?: string };
    [key: string]: unknown;
  };
}
```

The `event` field names the message type (for example `avatar_state_changed` or `message_received`) and `data` carries the payload. For the full protocol — the server events you receive and how transcripts and state changes are shaped — see **Server events**.

Some events tell you the session is ending — an idle warning, expired time, no credits left, or a provider that rejected the configuration. See **Session errors and shutdown** to handle them.

One event doesn't originate from the platform: `ui_update` carries arbitrary JSON that your own functions server **pushed to the client** — use it to drive your page's UI from your backend in sync with the conversation.

Messaging

Send messages and commands to the agent from your code

Voice is the primary way users talk to the agent — but your code can message it too. This page covers the SDK's sending surface: injecting messages into the conversation and driving the session with typed commands. For what you *receive*, see Events.

Send a message

Use `sendCommand` with a `SEND_MESSAGE` command. Commands are typed as `DataChannelCommand` — a discriminated union keyed by `type`, from the exported `DataChannelMessageType` enum.

```
import {
  NapsterCompanionApiSdk,
  DataChannelMessageType,
} from "@touchcastllc/napster-companion-api";

// Send a text message as the user and trigger a response
instance.sendCommand({
  type: DataChannelMessageType.SEND_MESSAGE,
  data: {
    text: "Hello",
    role: "user",
    trigger_response: true,
  },
});
```

The `role` decides how the message enters the conversation:

- `"user"` — treated as if the user typed it. Pair with `trigger_response: true` so the agent answers.
- `"system"` — a silent context update: the agent absorbs the information and uses it when relevant. Ideal for feeding page state or app events into the conversation ("The user just opened the billing page").

```
// Silent context: inform the agent without triggering a spoken response
instance.sendCommand({
  type: DataChannelMessageType.SEND_MESSAGE,
  data: {
    text: "The user just opened the billing page.",
    role: "system",
    trigger_response: false,
    delay: true,
  },
});
```

SendMessageData

Field	Type	Required	Description
`text`	`string`	Yes	The message text.
`role`	`"user" | "system"`	Yes	Who the message is from.
`trigger_response`	`boolean`	Yes	Whether the agent should respond.
`previous_item_id`	`string`	No	ID of the previous message in the thread.
`delay`	`boolean`	No	Hold the message until the agent finishes speaking.

Command reference

`sendCommand` accepts the full command union:

DataChannelMessageType	Value	Payload	Purpose
`SEND_MESSAGE`	`"send_message"`	`SendMessageData`	Send a text message into the conversation.
`CANCEL`	`"cancel"`	—	Interrupt the agent's current response.
`SET_SETTINGS`	`"set_settings"`	`{ modality?, functions?, inline_functions? }`	Update session settings, including registering tools mid-session.
`START_VIDEO`	`"start_video"`	`null`	Start the video stream.
`STOP_VIDEO`	`"stop_video"`	`null`	Stop the video stream.
`FUNCTION_CALL_OUTPUT`	`"function_call_output"`	`{ call_id, output }`	Return the result of an agent function call.

Most integrations don't send raw commands — the SDK sends `start_video` / `stop_video` for screen sharing, and `stopAvatarTalking()` wraps `cancel`. Reach for `sendCommand` when you need to inject a message or manage settings directly. For the protocol as a whole — the same commands over both WebRTC and WebSocket — see Client Commands.

From your backend

Your client isn't the only side that can put words into the conversation. A WebSocket-based explicit tool's functions server can inject context server-side with `context_update` — the backend sibling of `send_message`, typically used to deliver deferred tool results. See Updating the conversation context.

Methods

Every method on the instance returned by init, plus the top-level SDK object

This page is the full reference for everything you can call. Most of it covers the `NapsterCcompanionApiInstance` — the object that drives a live session, which `init` resolves to — grouped by area. At the end you'll find the top-level SDK object.

Lifecycle

Method	Signature	Description
`showAvatar`	`() => void`	Show the avatar if it's hidden.
`hideAvatar`	`() => void`	Hide the avatar from the screen.
`avatarIsVisible`	`() => boolean`	Whether the avatar is currently visible.
`destroy`	`() => void`	Tear down the SDK, remove DOM nodes, close the connection, and release media.

```
if (!instance.avatarIsVisible()) instance.showAvatar();
// ...
instance.destroy();
```

Always call `destroy()` when the widget leaves the page. See teardown.

Styling and position

Method	Signature	Description
`updateStyles`	`(styles: StyleObject) => void`	Merge new inline styles into the SDK root container.
`setPosition`	`(position: Position) => void`	Move the widget to one of the position values.
`clearPosition`	`() => void`	Clear a programmatic position and revert to the configured/default one.
`updateAvatarStyle`	`(style: Partial<avatarStyleConfig>) => void`	Update the avatar shape and border. Only the keys you pass change.

```
instance.setPosition("top-left");
instance.updateAvatarStyle({ view: "rectangle" });
instance.updateStyles({ zIndex: "2000" });
```

Feature control

Toggle and reconfigure features after init. `feature` is one of the keys of the `features` config object.

Method	Signature	Description
`enableFeature`	`(feature: keyof FeatureConfig) => void`	Enable a feature by name.
`disableFeature`	`(feature: keyof FeatureConfig) => void`	Disable a feature by name.
`updateFeatureConfig`	`(feature: keyof FeatureConfig, config) => void`	Update a single feature's config. Partial — only the keys you pass change.

```
instance.disableFeature("controls");
instance.updateFeatureConfig("disclaimer", { text: "New disclaimer text" });
instance.enableFeature("controls");
```

Feature names are typed as `keyof FeatureConfig`, so your editor autocompletes them and rejects typos. The valid keys are listed in the features config.

Screen sharing

Requires `features.screenShare.enabled: true`. See the screen sharing feature.

Member	Signature	Description
`startScreenShare`	`() => Promise<void>`	Start sharing. Opens the browser's screen picker.
`stopScreenShare`	`() => void`	Stop sharing.
`toggleScreenShare`	`() => Promise<void>`	Toggle sharing on or off.
`isScreenSharing`	`boolean` (getter)	Whether sharing is currently active.
`isScreenShareSupported`	`boolean` (getter)	Whether the browser supports sharing **and** the feature is enabled.

```
if (instance.isScreenShareSupported) {
  await instance.toggleScreenShare();
}
```

Audio control

Member	Signature	Description
`muteAudio`	`() => void`	Mute the avatar's audio output.
`unmuteAudio`	`() => void`	Unmute the avatar's audio output.
`isAudioMuted`	`boolean` (getter)	Whether the avatar's audio is muted.
`setAudioVolume`	`(volume: number) => void`	Set output volume. Range `0`–`1`; values outside are clamped.
`getAudioVolume`	`() => number`	Current output volume, `0`–`1`. Defaults to `1`.

Microphone control

Member	Signature	Description
`muteMic`	`() => void`	Mute the user's microphone.
`unmuteMic`	`() => void`	Unmute the user's microphone.
`isMicMuted`	`boolean` (getter)	Whether the microphone is muted.

Speech control

Member	Signature	Description
`stopAvatarTalking`	`() => void`	Interrupt the avatar's current response so it stops talking.
`isAvatarSpeaking`	`boolean` (getter)	Whether the avatar is currently speaking.
`isUserTalking`	`boolean` (getter)	Whether the user is currently talking, detected from the microphone.

```
// Build your own push-to-talk control
if (instance.isAvatarSpeaking) instance.stopAvatarTalking();
```

Session and advanced

Member	Signature	Description
`sessionId`	`string | undefined` (getter)	The current session identifier. Set once the connection is established, `undefined` after it closes.
`sendCommand`	`(command: DataChannelCommand) => void`	Send a typed command over the data channel. See Messaging.

sessionId

Read `sessionId` to correlate the live session with your backend records — for example to fetch the transcript later via Monitor sessions.

```
const id = instance.sessionId;
```

sendCommand

`sendCommand` takes a `DataChannelCommand` — the typed union documented on the Messaging page.

```
import { DataChannelMessageType } from "@touchcastllc/napster-companion-api";

instance.sendCommand({
  type: DataChannelMessageType.SEND_MESSAGE,
  data: { text: "Hi", role: "user", trigger_response: true },
});
```

SDK object

The top-level `NapsterCcompanionApiSdk` object exposes:

Member	Signature	Description
`init`	`(token, config?) => Promise<NapsterCompanionApiInstance>`	Connect and mount the widget. See Initialization.
`version`	`string`	The SDK version string, useful for diagnostics.

Examples

Web SDK integration examples for React, Vue 3, Angular, and vanilla JS

Copy-paste examples showing how to integrate the SDK in the most common frameworks. Each one mounts the widget on load and calls `destroy()` on unmount, so the connection and media tracks are always cleaned up.

In every example, `token` comes from your backend after creating a session — see Create a session.

> REACT

```
import React, { useEffect, useRef } from "react";
import { NapsterCompanionApiSdk } from "@touchcastllc/napster-companion-api";
import type { NapsterCompanionApiInstance } from "@touchcastllc/napster-companion-api";
import "@touchcastllc/napster-companion-api/styles";

export function CompanionWidget({ token }: { token: string }) {
  const containerRef = useRef<HTMLDivElement>(null);
  const instanceRef = useRef<NapsterCompanionApiInstance | null>(null);

  useEffect(() => {
    const initSDK = async () => {
      if (!containerRef.current) return;

      const result = await NapsterCompanionApiSdk.init(token, {
        mountContainer: containerRef.current,
        position: "bottom-right",
      });
      instanceRef.current = result;
    };

    initSDK();

    return () => {
      instanceRef.current?.destroy();
    };
  }, []);

  return <div ref={containerRef} style={{ width: "100%", height: "100%" }} />;
}
```

VUE 3

```
<template>
  <div ref="containerRef"></div>
</template>

<script setup lang="ts">
import { onMounted, onUnmounted, ref } from "vue";
import { NapsterCompanionApiSdk } from "@touchcastllc/napster-companion-api";
import type { NapsterCompanionApiInstance } from "@touchcastllc/napster-companion-api";
import "@touchcastllc/napster-companion-api/styles";

const props = defineProps<{ token: string }>();
const containerRef = ref<HTMLElement>();
let instance: NapsterCompanionApiInstance | null = null;

onMounted(async () => {
  if (!containerRef.value) return;

  instance = await NapsterCompanionApiSdk.init(props.token, {
    mountContainer: containerRef.value,
    position: "bottom-right",
  });
});

onUnmounted(() => {
  instance?.destroy();
});
</script>
```

ANGULAR

```
import {
  Component,
  OnInit,
  OnDestroy,
  ElementRef,
  ViewChild,
  Input,
} from "@angular/core";
import {
  NapsterCompanionApiSdk,
  NapsterCompanionApiInstance,
} from "@touchcastllc/napster-companion-api";

@Component({
  selector: "app-companion",
  template: "<div #containerRef></div>",
})
export class CompanionComponent implements OnInit, OnDestroy {
  @Input() token!: string;
  @ViewChild("containerRef") containerRef!: ElementRef<HTMLDivElement>;
  private instance: NapsterCompanionApiInstance | null = null;

  async ngOnInit() {
    this.instance = await NapsterCompanionApiSdk.init(this.token, {
      mountContainer: this.containerRef.nativeElement,
      position: "bottom-right",
    });
  }

  ngOnDestroy() {
    this.instance?.destroy();
  }
}
```

VANILLA JS

```
<div id="sdk-container"></div>
<script src="https://cdn.jsdelivr.net/npm/@touchcastllc/napster-companion-
api@latest/lib/index.standalone.js"></script>
<script>
  // token comes from your server after calling POST /public/agents/{agentId}/connections
  const sdk = window.napsterCompanionApiSDK;

  sdk.init(token, {
    mountContainer: "#sdk-container",
    position: "bottom-right",
  });
</script>
```

Initialize the SDK after the mount container exists in the DOM. In React, Vue, and Angular the lifecycle hooks above guarantee this. In plain HTML, put the `init` call after the container element, or wrap it in a `DOMContentLoaded` listener.

Troubleshooting

Common Web SDK issues, debug mode, and performance notes

Common problems when integrating the Web SDK and how to resolve them. For channel- and account-level issues (a session that won't connect, no audio, the avatar not loading), see the troubleshooting guide for the API and the WebRTC channel page.

Common issues

"Module not found"

Make sure the package and its peer dependency are both installed:

```
npm install @touchcastllc/napster-companion-api @reduxjs/toolkit
```

`@reduxjs/toolkit` is a required peer dependency of the ESM build. See Installation.

Invalid or expired token

Tokens are short-lived. If `init` rejects with a token error:

- Mint a fresh token from your backend right before connecting.
- Make sure you pass the token as the first argument to `init(token, config)` and don't decode or modify it.

Microphone access is blocked

If the user denies microphone permission (or the browser blocks it), the SDK shows built-in guidance walking the user through enabling microphone access in their browser settings. For iframe-embedded integrations, also make sure your embedding frame grants the permission — see the API troubleshooting guide.

The avatar appears unstyled

The stylesheet isn't loaded. In the ESM build you must import it once:

```
import "@touchcastllc/napster-companion-api/styles";
```

Or load it via a `<link>` tag — see Installation. The standalone build bundles the styles, so this doesn't apply there.

The mount container doesn't exist

Initialize the SDK only after the container is in the DOM. In plain HTML, wait for the document to be ready:

```
document.addEventListener("DOMContentLoaded", async () => {
  await NapsterCompanionApiSdk.init(token, {
    mountContainer: "#avatar-container",
  });
});
```

If the selector matches no element, the SDK warns and falls back to `document.body`. In React, Vue, and Angular, the framework examples mount inside lifecycle hooks that guarantee the element exists.

"SDK is already initialized"

`init` throws if a session is already running. Call `destroy()` on the previous instance before initializing a new one.

The session ends when the user navigates

A full page load tears down the session. To keep one session alive across pages, enable persistence — and note its rules:

- Cross-origin links break out **by default** — including links to your own subdomains, which are a different origin. To keep a specific domain inside the frame instead, list it in `persistence.include`; the session then survives the link, at the cost of the SDK being blind inside that domain.
- URLs listed in `persistence.exclude` intentionally break out.
- Under `persistence.iframeOnNavigate: true`, the landing page stays unframed until the first link click — and while unframed, only anchor clicks are intercepted, so a programmatic `location.assign` there ends the session unless you dispatch the persist-navigate event instead. The default (wrap on connect) doesn't have this window.

Debug mode

Set `debug: true` to get verbose SDK logging in the console. Pair it with `onError` and `onData` to see what's flowing:

```
await NapsterCompanionApiSdk.init(token, {
  debug: true,
  onError: error => console.error("SDK error:", error),
  onData: data => console.log("SDK data:", data),
});
```

Performance

Bundle size

- Use the ESM build rather than the standalone bundle so your bundler can tree-shake unused code.
- Import only the features you use so your bundler can drop the rest.

Memory

Always call `destroy()` when the widget unmounts — it closes the connection and releases the microphone (plus any screen-share or face-tracking camera capture, if used). In component frameworks, call it from the unmount hook:

```
useEffect(() => {
  return () => {
    instance?.destroy();
  };
}, [instance]);
```

Overview

Let an AI agent operate your website — expose your app's real operations as tools on the WebMCP standard

An agent embedded on your website can talk — but when the user says *"add the OLED to my cart,"* the best it can do is point at the button. **Edge MCP lets the agent operate the app itself**: your site declares which of its real operations an agent may run (`products.search`, `cart.add`, `checkout.placeOrder`), and the agent calls them — the cart opens, the page navigates, the order gets placed. `@napster-corp/edge-mcp` is a thin layer on the WebMCP standard: it polyfills `document.modelContext` in every browser and adds live-state resources and console debugging on top — your registrations stay pure standard WebMCP, so any compatible agent can drive them, the Napster Omniagent included.

What Edge MCP is — and isn't

The rule that explains everything: **the agent operates your website through the website's own functions.** If the website can do something, the agent can do it. If the website can't, neither can the agent — Edge MCP adds no capabilities your app doesn't already have; it makes the ones you choose *callable by an agent*.

Edge MCP is...	Edge MCP is not...
An integration you build into **your own website's source code** — each tool wired to the app's real functions, the same ones your buttons and forms call.	A crawler, screen-reader, or plugin you can drop onto any page — a locked-down site you can't modify won't work.
Bounded by your app: the agent can do exactly what the website can do.	Magic: it grants no abilities beyond your app's existing features.
A **curated** surface — the agent can only call what you deliberately expose, at the safety level you set.	Free-roaming access to everything on the page.
Running **as the signed-in user**, through your existing auth, validation, and business rules.	A permission escalation or a parallel backend to keep in sync.
The open **WebMCP standard** — any compatible agent can drive the surface.	A Napster lock-in: remove the toolkit and your tools still work against the native API.

What people can suddenly do

The agent can now **operate the website** — anything your site can do, a visitor can now have done by asking:

- *"I'm flying to Venice on May 3rd. Find me somewhere nice near the canal, around $300 a night."* — the agent searches your booking site, compares options, and prepares the reservation for the user to confirm. Fifteen seconds instead of fifteen screens.
- *"Refill my prescription, and book me the first pharmacist slot on Tuesday morning."* — someone who would never learn a new app just says it, in their own words, in any language.
- A prospect lands on your product at midnight — instead of a contact form and a wait, the agent walks them through the real product, doing the clicks and answering as it goes.

How it works

1. Your app registers **tools** on `document.modelContext` — each one a name, a description the agent reads, a JSON Schema for arguments, and an `execute` that calls your app's own code.
2. Optionally, it registers **resources** — live state slices (the current page, the cart) pushed to the agent the moment they change.
3. An agent on the page — embedded with the Web SDK, or any WebMCP-compatible agent — **discovers the surface at runtime**: it detects `document.modelContext`, reads the tool list, and wires itself up. No glue code on your side.
4. Because the UI and the agent call the same functions on the same stores, **the page stays in sync automatically** — an agent's `cart.add` re-renders every component bound to the cart.

Where it fits

Edge MCP is one half of putting an agent on a website; the Web SDK is the other:

	What it gives the agent
Edge MCP	Hands and eyes — the ability to operate your app and observe its state
Web SDK	Presence — the face, voice, and session your users talk to

For the end-to-end journey — agentify the site first, then add the agent — follow Add an AI Agent to Your Website.

How to get started

You don't have to build the integration by hand: with the Napster agent skills, your AI coding tool installs the library, proposes what to expose from your actual code, and writes the tools — nothing lands without your approval. Head to Installation to begin.

Installation

Add Edge MCP to your site — npm, script tag, or one prompt in your AI coding tool

With your AI coding tool

The fastest path: let a coding agent do the integration. The public Napster agent skills work with any Agent Skills-compatible tool (Claude Code, Cursor, Codex, and others):

```
npx skills add napster/omniagent-api-skills
```

Then ask your tool to *"agentify this app"* — or paste the ready-made starting prompt. It covers the whole journey, agentifying included, and then puts the Omniagent on the site; approve each step and stop wherever you like:

The skills install the library, analyze your codebase, propose which operations and state to expose, and write the tools — **nothing lands without your review and approval**. They also set up a keep-in-sync mechanism so the agent surface follows your app as it changes.

npm

```
npm install @napster-corp/edge-mcp
```

Import it once, early in your app's client entry:

```
import '@napster-corp/edge-mcp';
```

The import has one side effect: it polyfills `document.modelContext` (the standard WebMCP surface) and installs the resource extension. From there, everything you write is standard WebMCP. Outside the browser — SSR, workers, edge runtimes — the

import is a deliberate no-op, so it's safe in universal code.

Script tag

No npm or bundler — server-rendered templates (Rails, Django, PHP), static HTML? The package ships a pre-bundled browser build (~42 KB before gzip), served by any npm CDN:

```
<script src="https://cdn.jsdelivr.net/npm/@napster-corp/edge-mcp@0.3/dist/edge-
mcp.iife.min.js"></script>
```

Loading the script does exactly what the import does. Register tools in plain inline scripts:

```
<script>
  document.modelContext.registerTool({
    name: 'contact.submit',
    description: 'Submit the contact form.',
    inputSchema: { type: 'object', properties: { email: { type: 'string' } }, required:
['email'] },
    annotations: { destructiveHint: true },
    async execute({ email }) {
      return submitContactForm(email); // the site's real code
    },
  });
</script>
```

> Pin the version in the URL (`@0.3` , not `@latest`), and load the script — and your registrations — **before** the agent SDK initializes, so the tools exist by the time the agent reads `document.modelContext` . Helper exports are available on `window.EdgeMCP` .

Recommended layout

Keep the integration in one place — a `src/edge-mcp/` folder:

```
src/edge-mcp/
├── index.ts        # imports the toolkit, wires everything up
├── tools/          # one descriptor file per tool
│   └── index.ts    # registers them all via document.modelContext.registerTool
└── resources.ts    # registerResource calls (live state)
```

One wrinkle worth knowing: a tool's `execute` lives in a plain module, but things like navigation often exist only inside your framework (React Router's `useNavigate`, for example). Register a module-level handle from an in-tree component at mount, and have `execute` call through it.

Tools

What the agent can DO — standard WebMCP registerTool with safety annotations

A tool is one operation of your app the agent is allowed to run. Registration is **pure standard WebMCP** — there is no Napster-specific wrapper, and any WebMCP-compatible agent can call the result:

```
document.modelContext.registerTool({
  name: 'checkout.placeOrder',
  description: 'Submit the cart for purchase.',
  inputSchema: {
    type: 'object',
    properties: {
      paymentMethodId: { type: 'string' },
      addressId: { type: 'string' },
    },
    required: ['paymentMethodId', 'addressId'],
  },
  annotations: {
    readOnlyHint: false,
    destructiveHint: true, // ⇒ the agent confirms with the user before calling
    idempotentHint: true,  // safe to retry
  },
  async execute({ paymentMethodId, addressId }) {
    const order = await placeOrder(paymentMethodId, addressId); // your app's own code
    return { content: [{ type: 'text', text: `Order ${order.id} placed` }] };
  },
});
```

The `execute` calls the same functions your buttons and forms already call — the toolkit doesn't replace them, it exposes the ones you choose. The return value uses the standard MCP result shape (`{ content: [{ type: 'text', text }] }`).

Safety annotations

Safety rides on the **standard** MCP annotation hints — portable to any consumer, no proprietary tiers:

Hint	Meaning
`readOnlyHint`	The tool only reads state — the agent may call it freely.
`destructiveHint`	**Convention: confirm with the user before calling.** Use for anything final — payments, sends, deletions.
`idempotentHint`	Safe to retry with the same arguments.
`untrustedContentHint`	The output may carry untrusted or injected content.
`openWorldHint`	The tool reaches beyond the site itself — external systems, the open web. Most website tools are closed-world and leave this unset.

A tool with no annotations is treated as a **reversible** write: the agent announces what it's doing and runs it. The rule of thumb: pure read → `readOnlyHint: true`; reversible write → neither hint; final/irreversible → `destructiveHint: true`.

Writing tools agents actually use well

- **Names** are `namespace.verb` — `products.search`, `cart.add` — snake or camel within segments.
- **Descriptions are the agent's manual.** Say what the tool does, when to reach for it, and what it returns — the agent decides relevance from this text alone.
- **Schemas are contracts.** Mark required fields, describe each property; the agent fills arguments from conversation, and a precise schema is the difference between clean calls and garbage.
- **Expose operations, not endpoints.** One user-meaningful action per tool ("book a table"), not a thin wrapper per API route.

Reading the surface back (consumer side)

Only relevant if you build something that *consumes* tools — an inspector, a custom bridge. Two wire-contract details: `getTools()` returns each `inputSchema` as a JSON **string** (parse before use — that's Chromium's native contract, matched for interoperability), and `executeTool(tool, argsJson)` resolves with a JSON **string** of the result envelope. Always pass the handle `getTools()` returned, never a hand-built object, and re-read the list on the `toolchange` event.

> To unregister, pass an AbortSignal when registering — `registerTool(descriptor, { signal: controller.signal })` — and call `controller.abort()` when the tool should go away. `registerTool` itself returns nothing.

Resources

What the agent can SEE — live state slices pushed to the agent as they change

Tools let the agent act; **resources let it see**. A resource is a live slice of your app's state — the page the visitor is on, the cart contents — that the agent receives the moment it changes, without calling anything:

```
import { registerResource } from '@napster-corp/edge-mcp';

registerResource({
  uri: 'state://cart',
  name: 'cart',
  description: 'The current shopping cart',
  mimeType: 'application/json',
  get: () => cartStore.getCurrent(),
  subscribe: (onChange) => cartStore.subscribe(onChange), // return an unsubscribe fn
});
```

With this, the agent always knows what's in the cart — when the *user* adds an item by hand, the agent sees it and can react ("I see you added the OLED — want a wall mount to go with it?").

The consumer surface

Installed on `document.modelContext` by the package's import side effect:

Member	Purpose
`getResources()`	List registered resources
`readResource(uri)`	Read one resource's current value
`subscribeResource(uri, handler)`	Subscribe to changes
`resourceupdated` event	`CustomEvent` with `detail = { uri, value }`
`resourcelistchanged` event	Fired when the resource list changes

> Resources are a **Napster extension**: WebMCP hasn't formalized resources yet, so this channel is consumed by the Napster agent over its own path and is not interoperable with third-party WebMCP agents — unlike the tool surface, which is fully standard today. When the standard lands resources, this API is designed to converge with it.

The resource descriptor

Parameter	Type	Required	Description
`uri`	`string`	Yes	Stable identity, mirroring MCP — `state://cart`. Registering the same `uri` again replaces the previous registration.
`name`	`string`	Yes	Short logical name — `cart`.
`description`	`string`	No	What the value represents. The agent reads this to decide what the state means.
`mimeType`	`string`	No	MIME type of the value — `application/json`.
`get`	`() => T | Promise<T>`	Yes	Returns the current value. Called on every read **and** every push, so keep it cheap and side-effect-free.
`subscribe`	`(onChange) => () => void`	No	The push source. Call `onChange` when your state changes, and return an unsubscribe function. Omit it to make the resource pull-only.

Don't pass the new value through `onChange` — it takes no arguments. `get` is the value, `subscribe` is only the *when*: on every `onChange`, the extension re-reads `get()` itself.

`registerResource` returns an unregister function. Calling it tears down the push subscription and removes the resource from the list.

When something earns a resource

> ⚠ Register a resource **only for state that changes without the agent acting** — the user navigating, clicking, typing; other tabs; server pushes. If the only way a value changes is through one of the agent's own tools, the tool's return value already tells it — a resource there is pure noise. This is the resource gate; apply it before every `registerResource`.

Typical resources that pass the gate: the **current page** (pushed on navigation), the **cart** (user can modify it by hand), a **form's draft state**. Typical rejects: a search-results list only the agent's `search` tool produces.

Behavior and guarantees

How the extension treats your `get` and `subscribe`. These rules decide what you'll see when you debug a resource that seems to update too often — or not at all — and what you can assert in a test.

Reads are always live

`readResource(uri)` calls your `get()` at the moment of the read and returns what it returns. Nothing is cached: no stored copy, no TTL, no staleness window. Two reads in a row call `get()` twice.

That makes `get()` a sound place to instrument reads in a test — every read goes through it. One caveat if you count calls: **pushes also call** `get()` (that's how the extension learns the new value), so a naive counter registers pushes as well as reads.

Identical consecutive pushes are filtered

When your `subscribe` callback fires, the extension re-reads `get()` and compares the result against the last value it dispatched for that URI. If they serialize identically, the push is dropped — so a store that fires several times per operation (`pending → value → pending`) doesn't spam the agent with a value that never changed.

The comparison is against the **immediately previous dispatched value only** — it is not a history:

Values your resource reads	Pushes the agent receives
A → A → A	1
A → B → A	3
A → B → B → C	3

So "three user actions must produce three updates" only holds if each action genuinely changes the value. When an expected push doesn't arrive, turn on debug mode — a filtered push logs `push suppressed (value unchanged)`.

> If a value can't be serialized (circular references, `BigInt`), the extension fails **open**: it dispatches the push and drops the stored baseline, so a comparison it can't make never silences a real change.

`subscribe` must return an unsubscribe function

The extension calls it to disconnect from your store — on unregister, and before a re-registration. If your `subscribe` returns nothing, the subscription can never be torn down and it leaks; you'll get a console warning saying so. Return the teardown, or omit `subscribe` entirely to make the resource pull-only.

Re-registering a URI starts fresh

Registering the same `uri` again replaces the descriptor: the previous `subscribe`'s teardown runs first, and the comparison baseline is cleared. The first push after a re-registration therefore always dispatches, even if the value is unchanged — worth knowing when a hot reload re-runs your registrations.

Testing

Test your tools by hand and watch the whole surface through debug console logs — no agent required

Test your agent surface before any agent is connected — the browser is the inspector. Every tool can be called by hand from the console, Chrome's DevTools shows the surface natively, and debug mode narrates everything the toolkit does.

Call tools by hand

Every registered tool is callable from the DevTools console:

```
const tools = await document.modelContext.getTools();
const t = tools.find(x => x.name === 'cart.add');
await document.modelContext.executeTool(t, JSON.stringify({ productId: 'sku_123' }));
```

The real app should react — the cart drawer opens, the page navigates, the form submits. That's the whole point of the test: **the agent will use exactly the same path**, so if the app responds to your console call, it will respond to the agent.

Two wire-contract details when reading results by hand: `getTools()` returns each tool's `inputSchema` as a JSON **string** (parse before use), and `executeTool` resolves with a JSON **string** of the standard result envelope:

```
const text = JSON.parse(await document.modelContext.executeTool(t, JSON.stringify(args)))
  .content?.[0]?.text;
```

A good pass covers **one tool per safety level** you registered — a read, a reversible write, and (carefully) a `destructiveHint` tool — confirming for each that the returned `content` makes sense and the UI reacts.

Chrome DevTools

Chrome ships a dedicated WebMCP panel in DevTools: every registered tool is listed with its description and an invocation counter, every call is logged with its status, input, and output — and you can **invoke tools manually** with your own parameters, straight from the panel. This works because Edge MCP mirrors tools into the browser's native WebMCP registry.

Enabling the panel

The panel doesn't appear by default — WebMCP is an origin-trial feature (Chrome 149+), and for local development you enable it with flags:

1. Open `chrome://flags/#enable-webmcp-testing` and set **WebMCP for testing** to **Enabled**.
2. On the same page, search for **WebMCP support in DevTools** and set it to **Enabled** too.
3. Relaunch Chrome.
4. Open DevTools on your page → **Application** tab → **WebMCP** in the sidebar.

> The flags are only needed for the DevTools panel. Edge MCP itself works without them in every browser — the polyfill provides `document.modelContext` regardless; the native mirror simply stays dormant until a native WebMCP surface exists.

The DevTools panel doesn't know about resources — they're a Napster extension. For those, use debug mode below.

Debug mode

Turn on the toolkit's flow logs and it narrates the entire surface with a colored `[edge-mcp]` prefix:

```
import { setDebug } from '@napster-corp/edge-mcp';
setDebug(true);
```

Or at runtime, straight from the console — no rebuild needed:

```
globalThis.__EDGE_MCP_DEBUG__ = true;
```

What you'll see, per event:

Log line	When
`registered tool "…"` / `unregistered tool "…"`	Tool lifecycle
`tool "…" called (locally | via native registry) — args:`	Every invocation, with its arguments
`tool "…" responded:` / `tool "…" failed:`	Every result or error
`registered resource "…"`	Resource lifecycle
`resource "…" changed →`	Every push sent to consumers, with the value
`resource "…" push suppressed (value unchanged)`	A push dropped because the value matched the last one sent

Logging is off by default, so the published integration stays quiet in production. Warnings about real problems (a failed native-registry mirror, a throwing resource getter) always print, debug on or off.

Testing resources

Resources only earn their place if they update on **out-of-band** changes — things the user does by hand. So test them that way:

1. Enable debug mode.
2. Use the site yourself: navigate, add something to the cart, type into the form — whatever your resources track.
3. Watch for `resource "…" changed →` — each user action that should update state should log **exactly once**. Several echoes per action means your `subscribe` is wired to too broad a signal; none means either the subscription isn't firing, or the value didn't actually change — check for a `push suppressed (value unchanged)` line to tell those two apart.

You can also read a resource's current value directly:

```
await document.modelContext.readResource('state://cart');
```

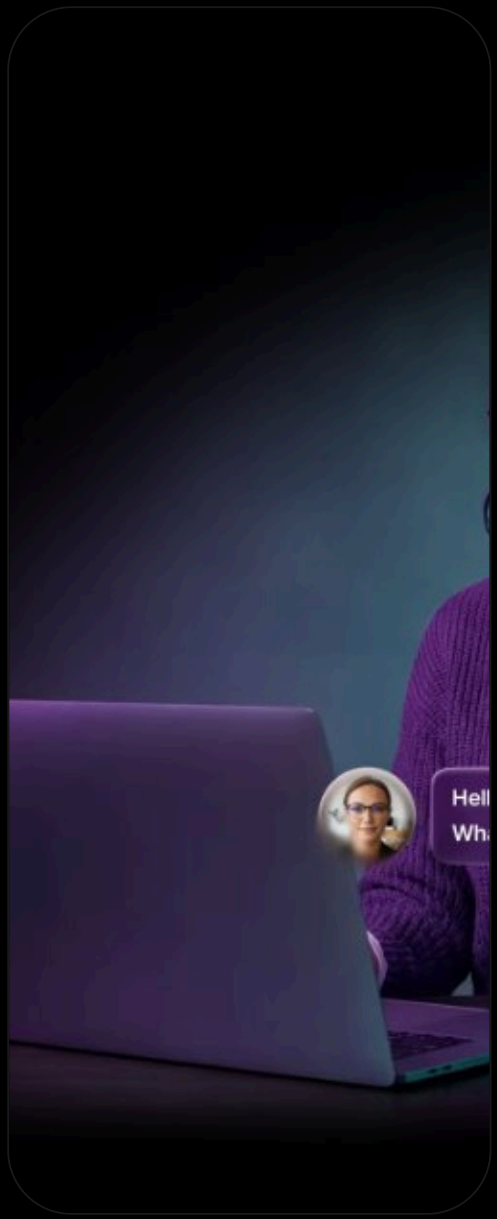
Napster

SECTION 06

Guides

Step-by-step walkthroughs for keys, Azure resources, digital twins, and video recordings.

Create a New API Key	257
Create a Resource in Azure Portal	264
Re-enable an Azure Resource	271
Enable Digital Twins	276
Video Recordings	278



Create a New API Key

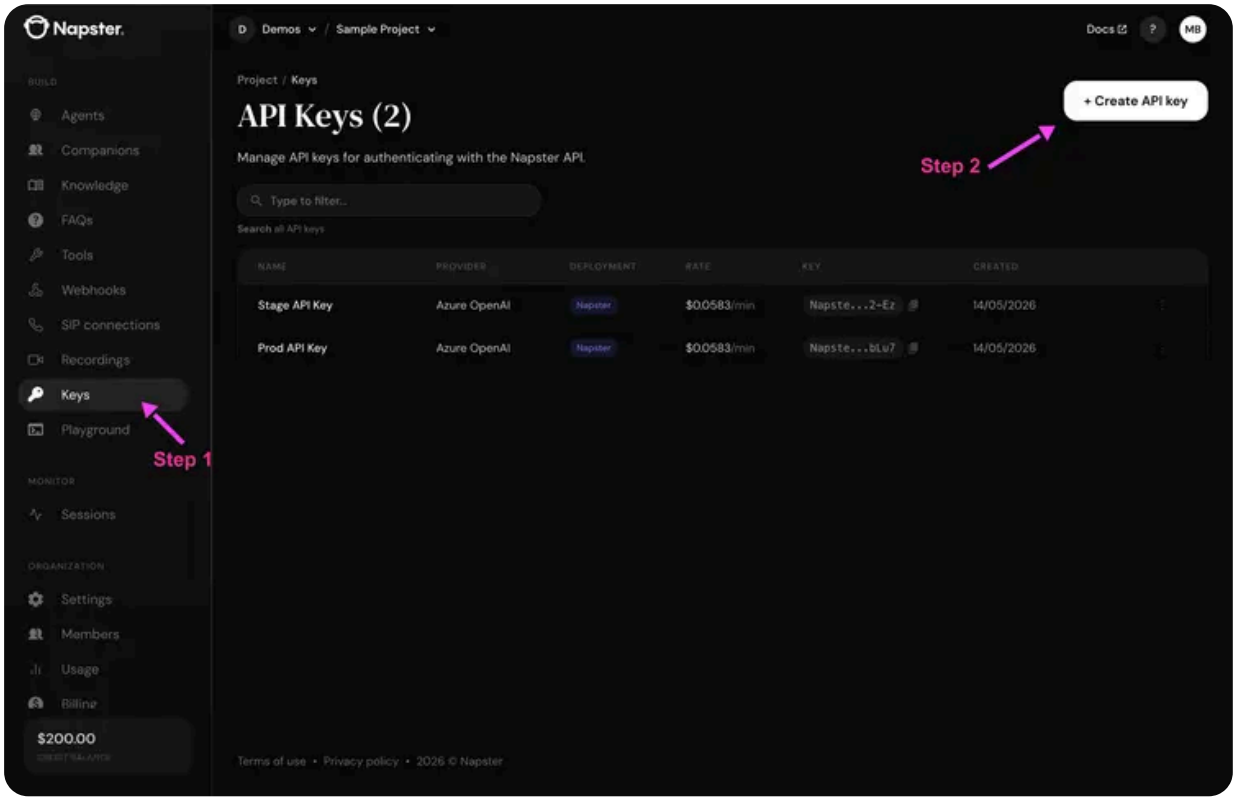
Generate an API key from the Napster dashboard to authenticate with the Napster API

API keys authenticate your requests to the Napster API. Each key is scoped to a project, so you can manage access separately for different environments (e.g. staging vs production). A key also sets the architecture your Omniagent runs on and the per-minute rate for its sessions.

> ⚠️ Your API key is a secret. Keep it private and only use it on the server side — never expose it in client-side code.

1 Navigate to Keys

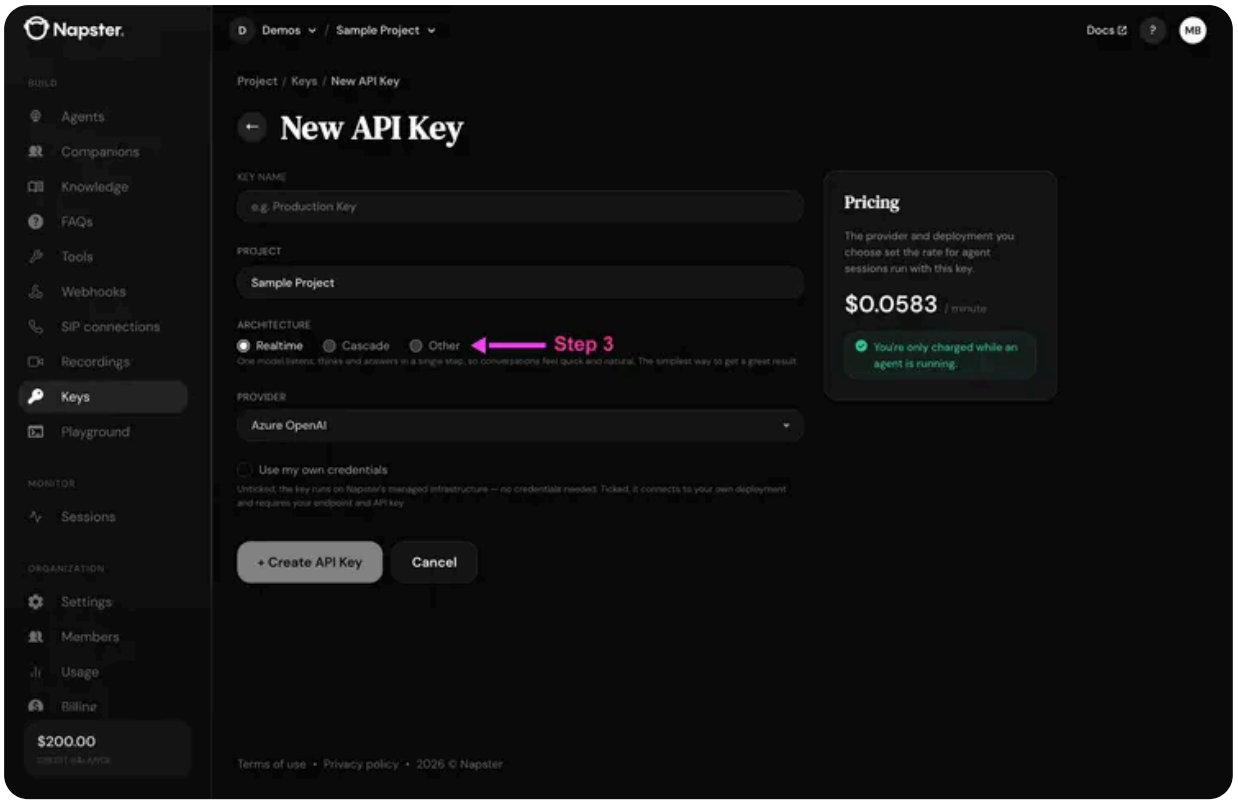
Open the Napster dashboard and select your organization and project from the top navigation. Then click **Keys** in the left sidebar under **Build**.



2 Create a new API key

Click the **+ Create API key** button in the top right corner.

3 Configure the key



Fill in the form:

Field	Description
Key Name	A descriptive label (e.g. `Stage API Key`, `Prod API Key`)
Project	Automatically set to your current project
Architecture	How your Omniagent listens, thinks, and speaks: **Realtime**, **Cascade**, or **Other** (see below). Each architecture has its own provider or platform options.

The **Pricing** panel beside the form shows the per-minute rate for your current selection. You're only charged while an agent is running.

Architectures

> REALTIME

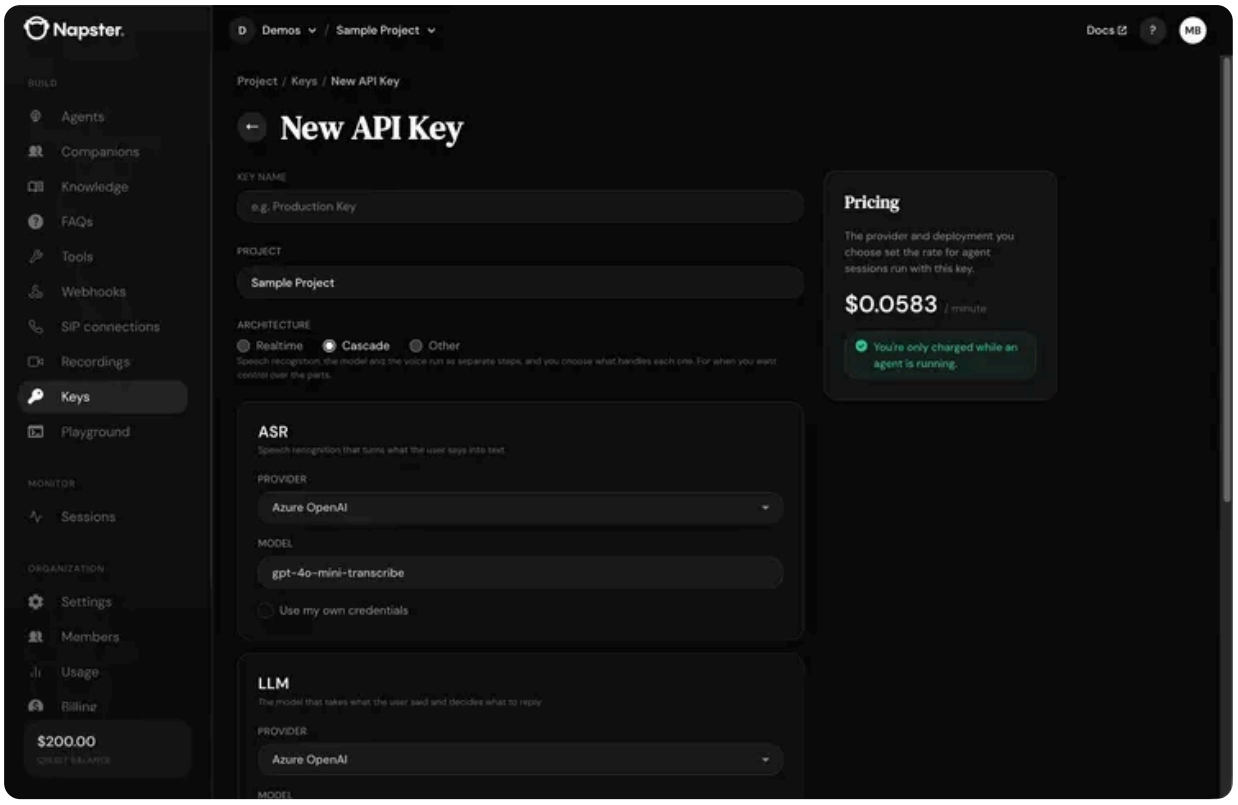
One model listens, thinks, and answers in a single step, so conversations feel quick and natural. Realtime is the default and the simplest way to get a good result.

Choose a **Provider**:

Provider	Description
Azure OpenAI	Default provider. Works with all channels including phone. Runs on Napster's managed infrastructure — no credentials needed. Tick **Use my own credentials** to connect your own Azure OpenAI deployment instead (deployment name, endpoint URL, API key, and optionally a model).
OpenAI	Connects directly to OpenAI. Always requires your own OpenAI API key (and optionally a model). Required for MCP connectors.

Realtime sessions cost `$0.0583` per minute on Napster's managed infrastructure, or `$0.01` per minute when you use your own credentials.

CASCADE



Speech recognition, the language model, and the voice run as separate steps instead of one realtime model, and you choose the provider for each one. Use Cascade when you want control over the individual parts.

Stage	What it does	Providers
ASR	Turns what the user says into text	Azure OpenAI
LLM	Takes what the user said and decides what to reply	Azure OpenAI, Anthropic
TTS	Speaks the reply out loud	Azure OpenAI, OpenAI

Azure OpenAI stages run on Napster's managed infrastructure with a preset model. Tick **Use my own credentials** on a stage to connect your own Azure OpenAI deployment for that stage. Anthropic and OpenAI stages always require your own API key and a model name.

Some features work differently on Cascade — see Cascade architecture.

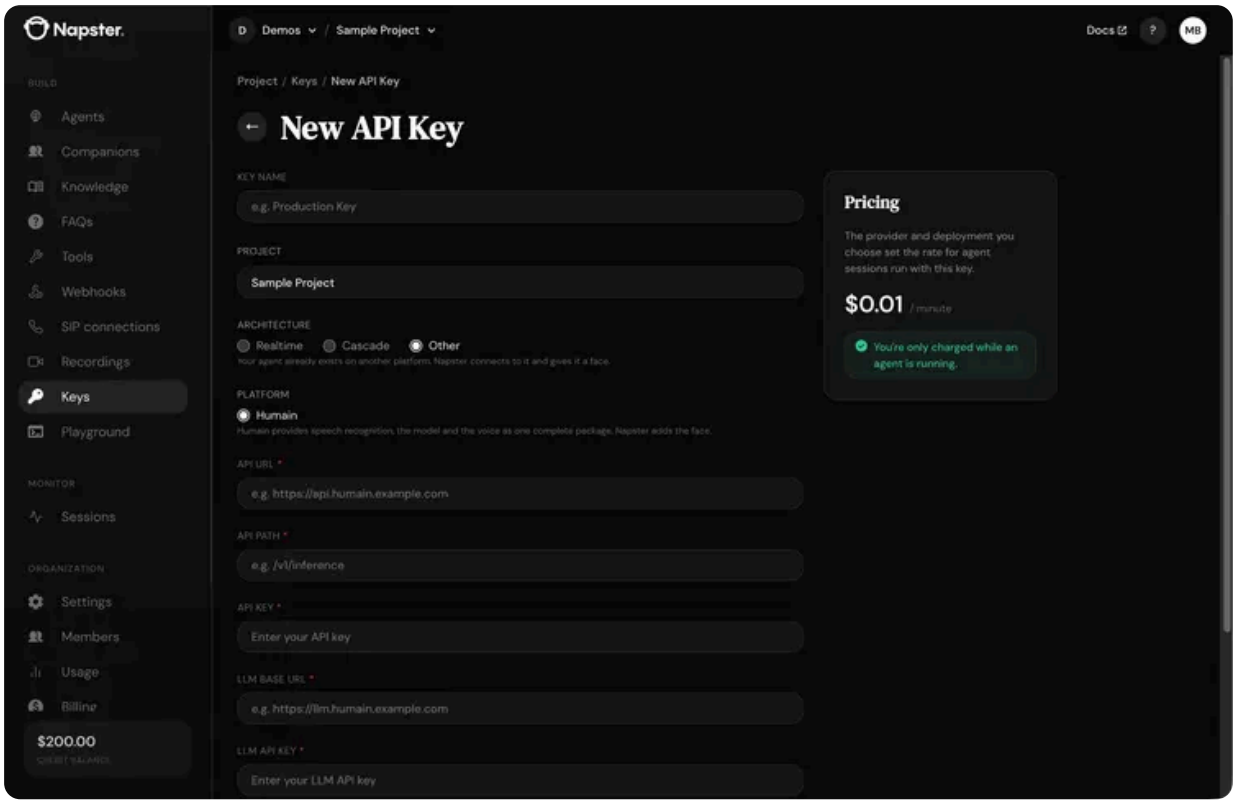
OTHER

Your agent already exists on another platform. The Napster API connects to it and gives it a face — a lifelike video avatar on top of your existing agent logic.

> Humain is not enabled by default. Contact us to request access for your organization.

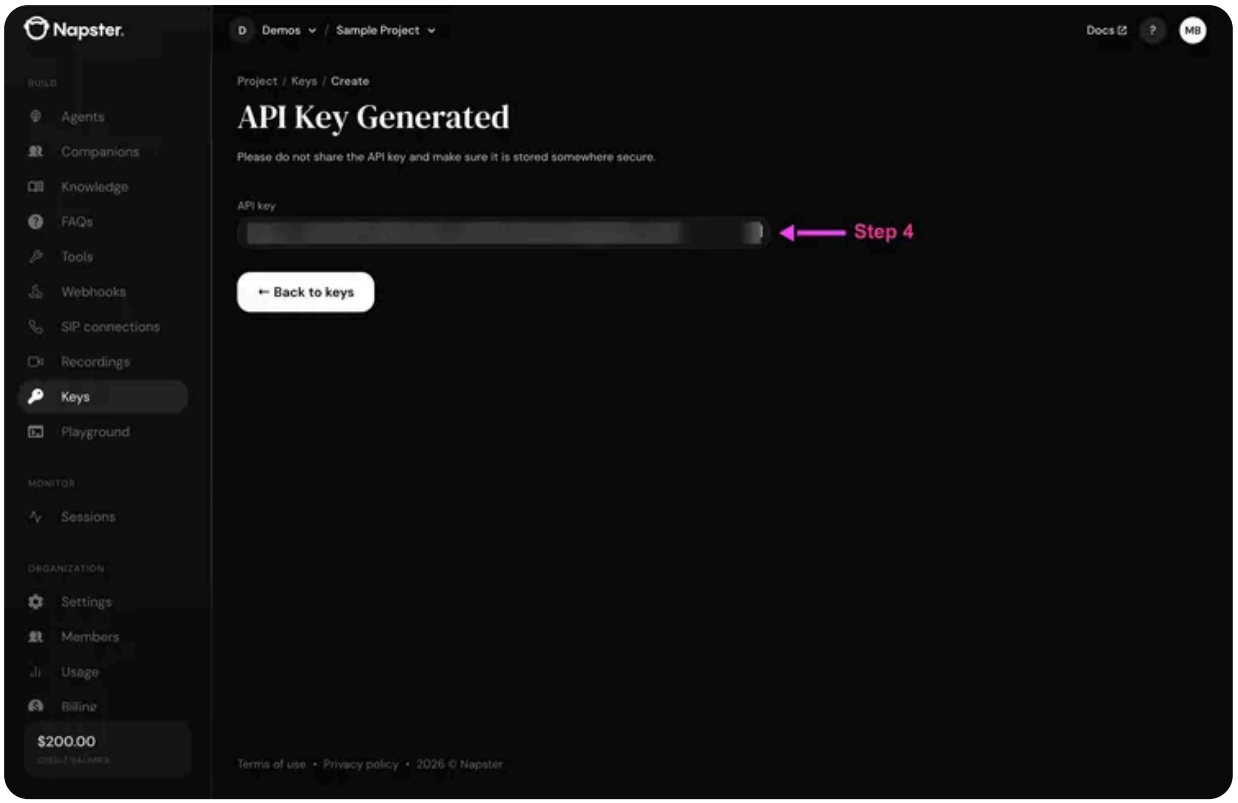
Choose a **Platform**:

Platform	Description	Required fields
Humain	Powered by Allam, an Arabic-first foundation model built by HUMAIN in Saudi Arabia with strong dialect coverage and cultural alignment. Humain provides speech recognition, the model, and the voice as one package. Best for Arabic-language agents.	API URL, API path, API key, LLM base URL, LLM API key, LLM model



Click **+ Create API Key** to generate the key.

4 Copy the key



The **API Key Generated** page shows your new key. Copy it from the **API key** field, then click ← **Back to keys**. The key now appears in your keys list with its provider, deployment, and rate.

5 Store the key

Save the key as an environment variable in a gitignored `.env` file:

```
NAPSTER_API_KEY=your-api-key-here
```

All API requests use the `X-Api-Key` header for authentication:

```
curl https://companion-api.napster.com/public/companions/napster-stock?pageSize=1 \
  -H "X-Api-Key: $NAPSTER_API_KEY"
```

A `200` response confirms your key is working.

Next steps

- Authentication — full details on how authentication works
- Cascade architecture — which features behave differently on a Cascade key
- Quickstart — build and deploy your first Omniagent

Create a Resource in Azure Portal

Set up the Napster API resource in your Azure subscription

This guide walks you through creating a Napster API resource in the Azure Portal. Once created, the resource links your Azure subscription to the Napster API platform.

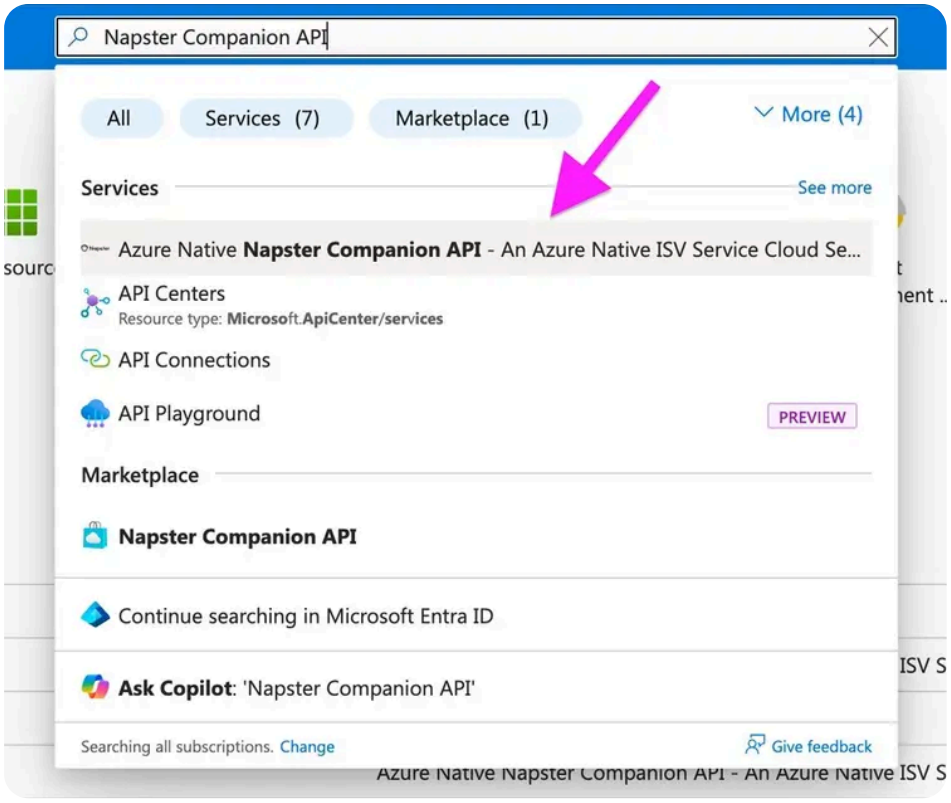
> This is the path for billing through your **Azure subscription**. If you don't need that, the fastest way to get access is signing up directly at napster.com/developer — no Azure required.

1 Open the Azure Portal

Go to portal.azure.com and sign in with your Azure account.

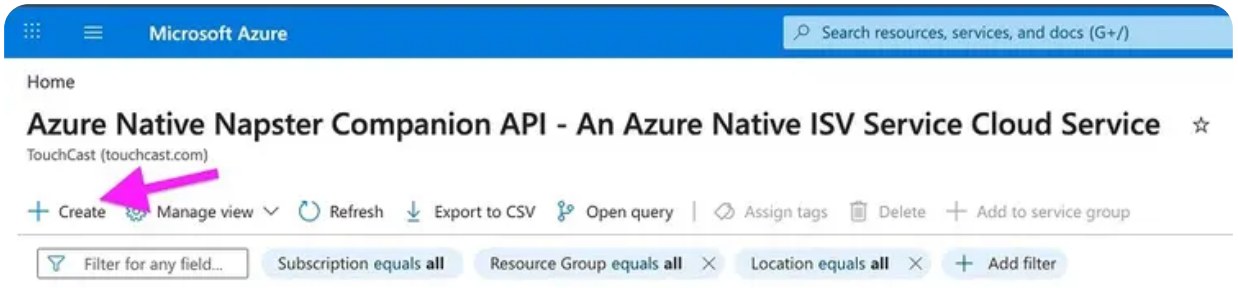
2 Search for Napster API

In the search bar at the top of the page, type **Napster API**. From the results, select **Azure Native Napster API - An Azure Native ISV Service Cloud Service** under Services.



3 Create a new resource

You'll land on the Napster API service page. Click the **+ Create** button to start setting up a new resource.

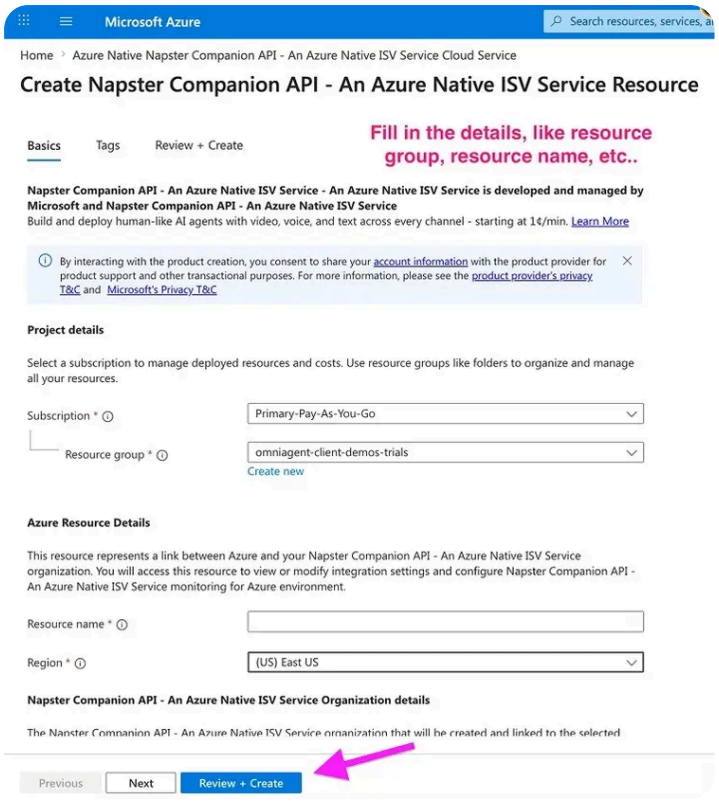


4 Configure the resource

Fill in the resource details:

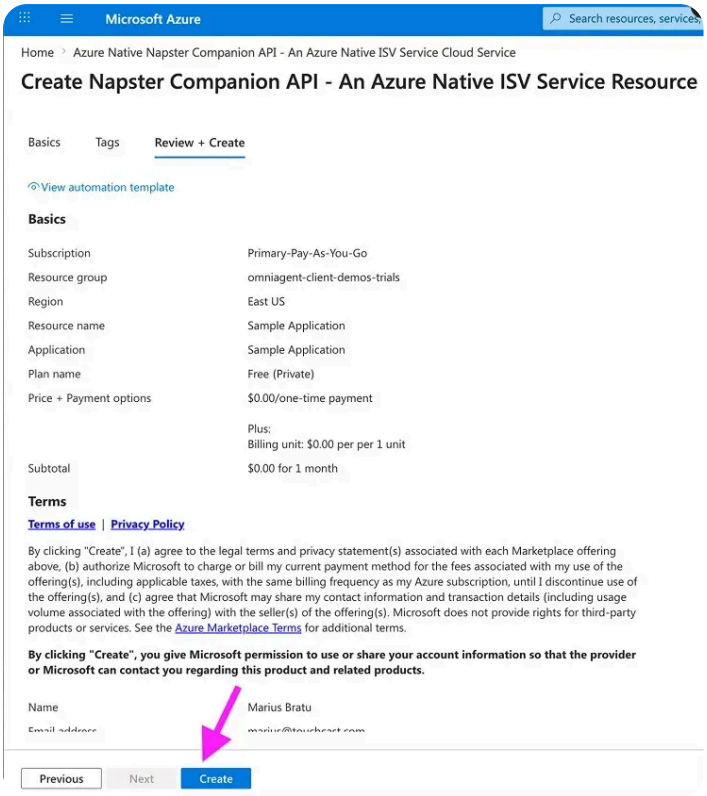
- **Subscription** — select your Azure subscription
- **Resource group** — choose an existing resource group or create a new one
- **Resource name** — give the application a name
- **Region** — select **(US) East US**

Once everything is filled in, click **Review + Create**.



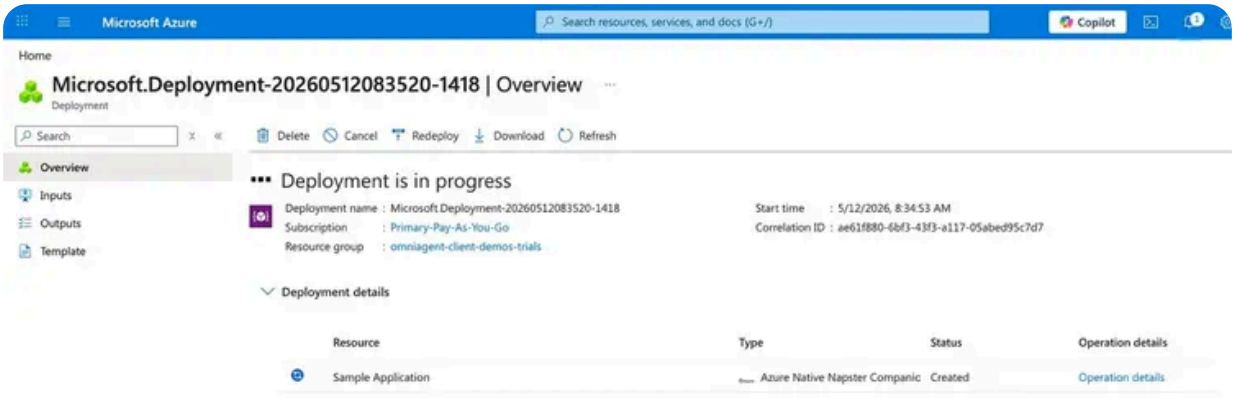
5 Review and create

Review your configuration. When everything looks correct, click **Create**.



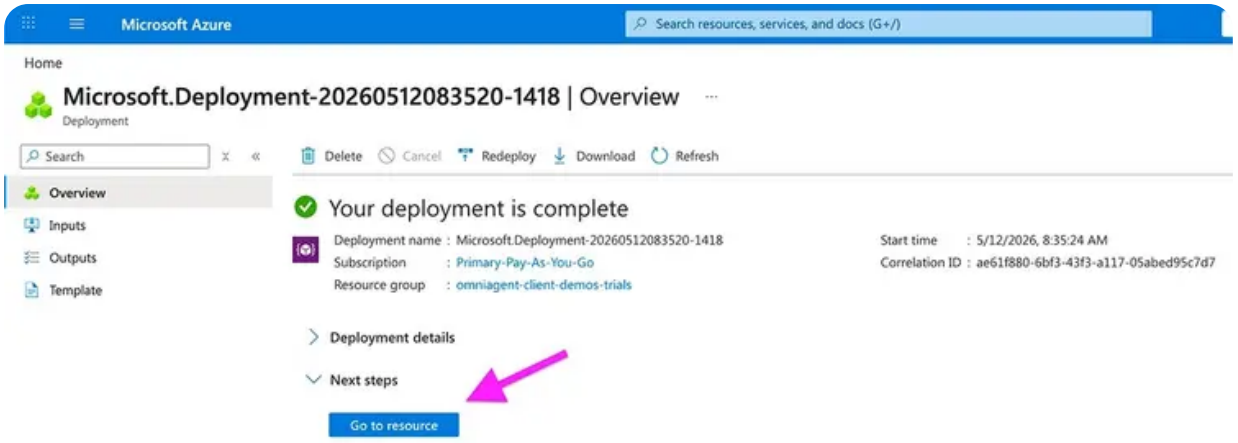
6 Wait for deployment

Azure will deploy the resource. This may take a few moments.



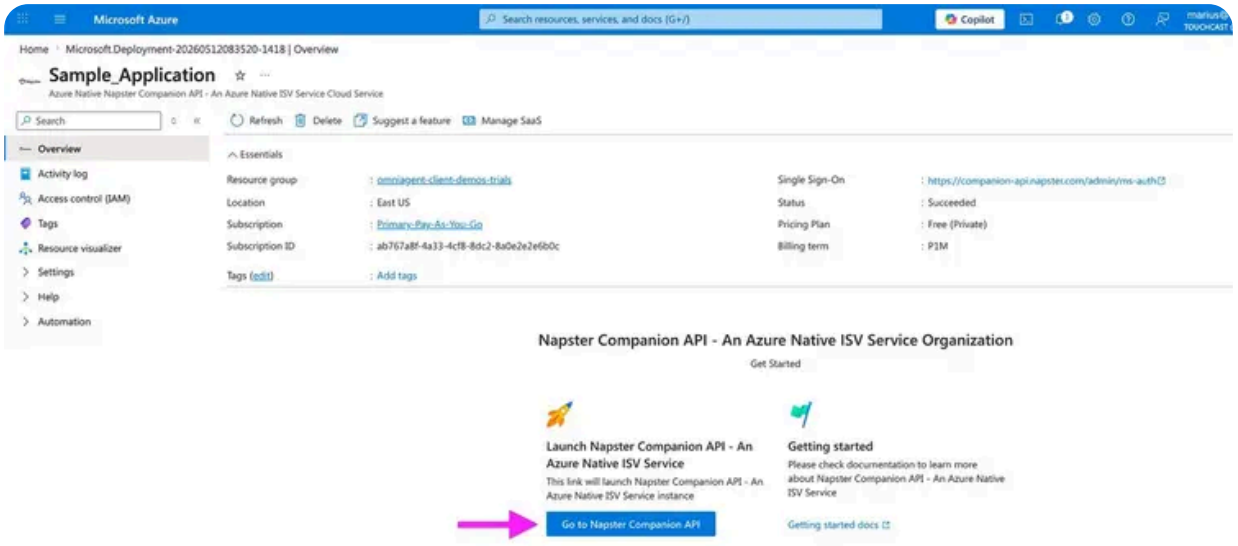
7 Go to the resource

Once the deployment completes, click **Go to resource** to open your new Napster API resource.



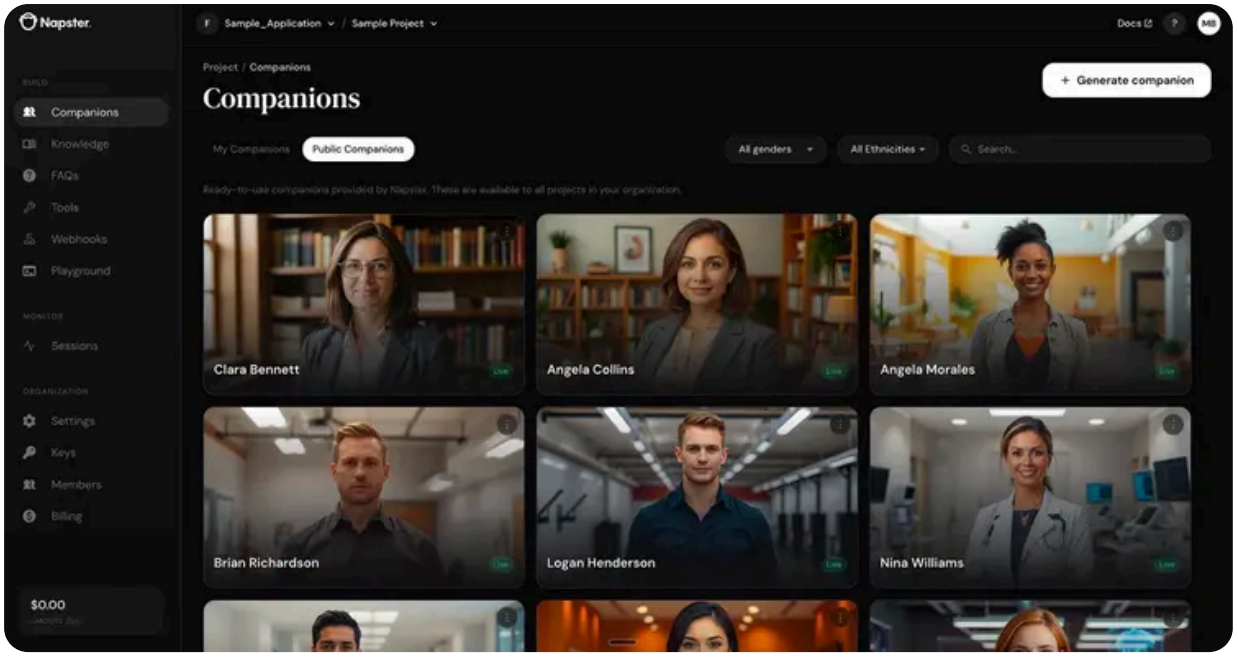
8 Go to the Napster API dashboard

From the resource page, click **Go to Napster API** to open the Napster API dashboard.



9 You're in

You'll land on the Napster API dashboard. From here, you can browse companions, set up knowledge and tools, create API keys, and test your agent in the Playground. Follow the Quickstart to build and deploy your first Omniagent.



10 Keep your subscription active

> ⚠ A subscription created this way is a 1-month subscription with **auto-renew off** by default. When auto-renew is off, the subscription — and your access to the Napster API — expires on the renewal date and the service is cancelled. Turn auto-renew on to keep the service running. You can only change this setting while the subscription is active; if it has already expired, follow Re-enable an Azure Resource to restore access to your existing resource.

The auto-renew setting lives on the **SaaS subscription**, which is separate from the resource you opened with **Go to resource**. To turn it on:

1. In the Azure Portal, type `SaaS` in the global search bar at the top and select **SaaS** under Services.
2. Find and select your Napster API subscription from the list.
3. On the **Overview** (Product and plan details) page, find the **Auto-renew** column — it shows `Auto-renew: Off` and the expiry date. Select the **Edit Auto-renew** link (or the **Edit Auto-renew** button in the top command bar).
4. In the side panel that appears, switch auto-renew **On**, then select **Update**.

The **Auto-renew** column now shows the renewal date instead of an expiry date.

> The **Edit Auto-renew** option only appears if you have **Owner** or **Contributor** permissions on the subscription. With read-only access, ask whoever holds the Owner role to make the change.

Common errors

The plan can't be purchased using a free subscription

If you select a free Azure subscription when configuring the resource, creation fails with:

```
The plan 'napster_companion_api_feb_2026' can't be purchased using a free
subscription. Please select a different subscription and try again.
```

There's no charge for the plan itself — you're only billed for usage, such as running agents — but Azure still requires a billable subscription to complete the purchase.

Re-enable an Azure Resource

Restore access to your Napster API resource after its Azure SaaS subscription expired or was canceled

A Napster API subscription created through the Azure Portal is a 1-month SaaS subscription with **auto-renew off** by default. If auto-renew was never turned on, the subscription expires on its renewal date and Azure cancels it — and with it, your access to the Napster API. See Keep your subscription active for turning auto-renew on before this happens.

A canceled subscription can't be reactivated. The fix is to create a **new** SaaS subscription and link it to your **existing** Napster API resource — your resource, and everything in it, is still there. This guide walks you through it.

1 Create a new SaaS subscription

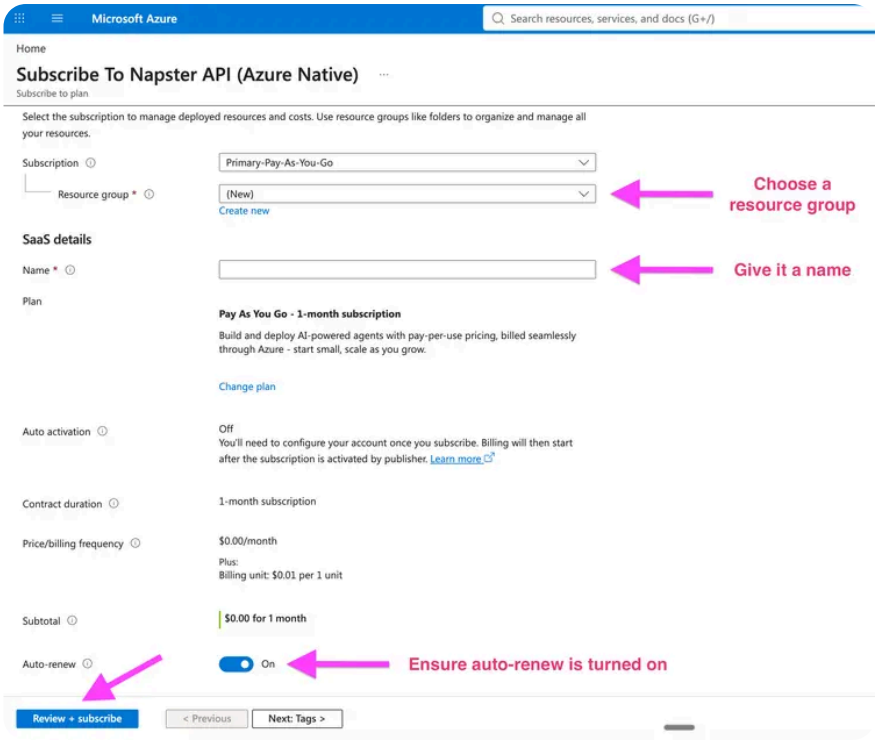
Open the Napster API subscribe page directly:

https://portal.azure.com/#create/touchcastinc1655995956899.napster_companion_api

Sign in with the same Azure account that owns your existing Napster API resource, then fill in the details:

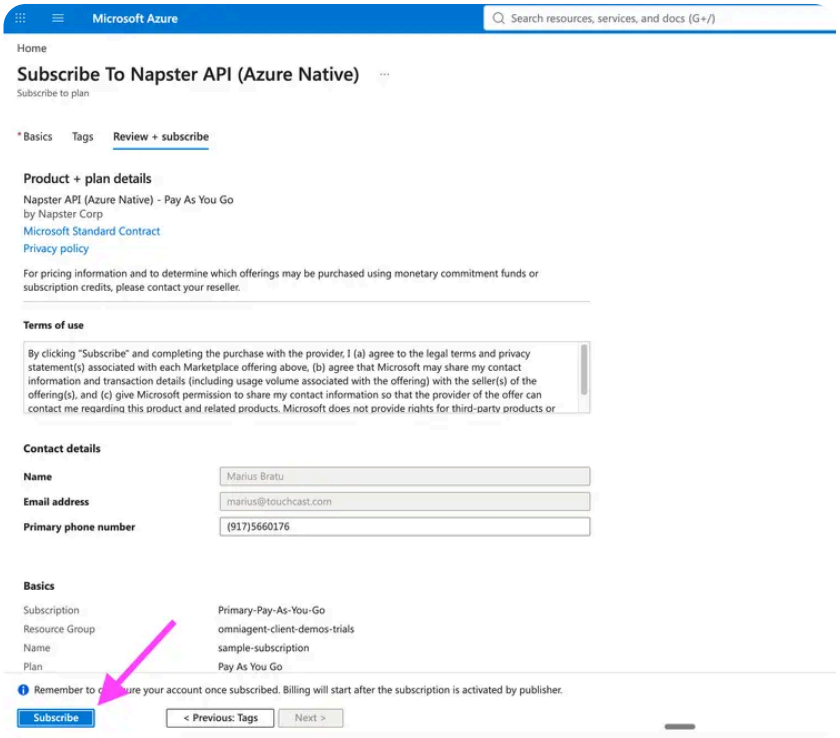
- **Subscription** — select your Azure subscription
- **Resource group** — choose a resource group
- **Name** — give the new SaaS subscription a name
- **Auto-renew** — switch it **On**, so this subscription doesn't expire the way the last one did

When everything is filled in, click **Review + subscribe**.



2 Review and subscribe

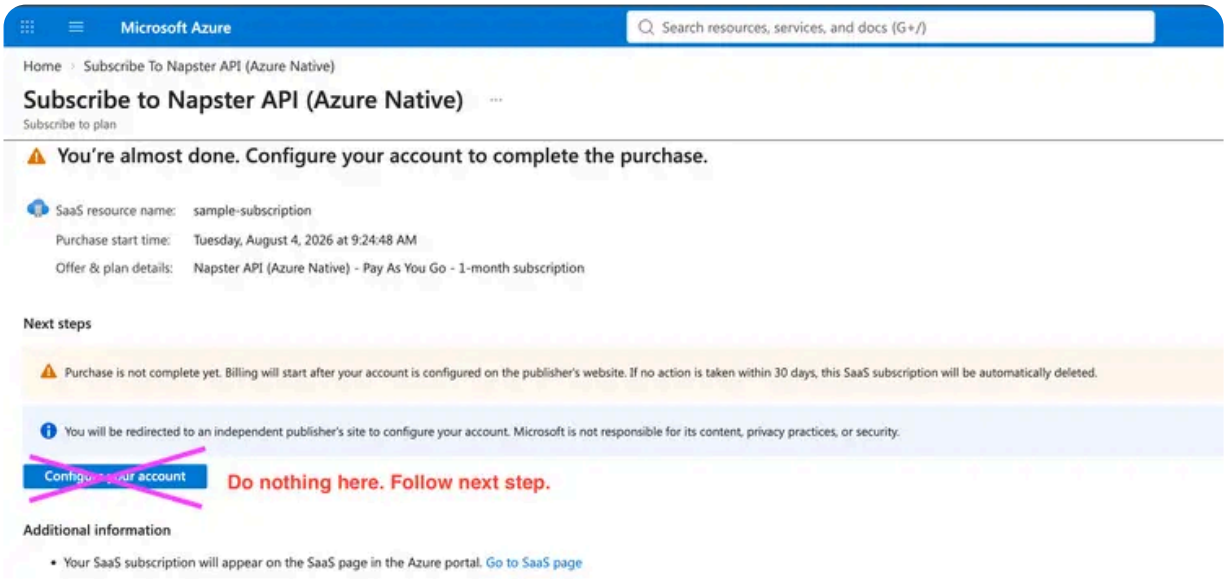
Review the plan, terms, and contact details, then click **Subscribe**. Azure creates the new SaaS subscription — this may take a few moments.



3 Do not configure the account

When the purchase completes, Azure shows *"You're almost done. Configure your account to complete the purchase."*

Do nothing here. Don't click **Configure your account**. You're going to link this subscription to your existing resource instead, so at the end of this step it stays deliberately **not activated** (its status shows as *Pending fulfillment*). That's correct.

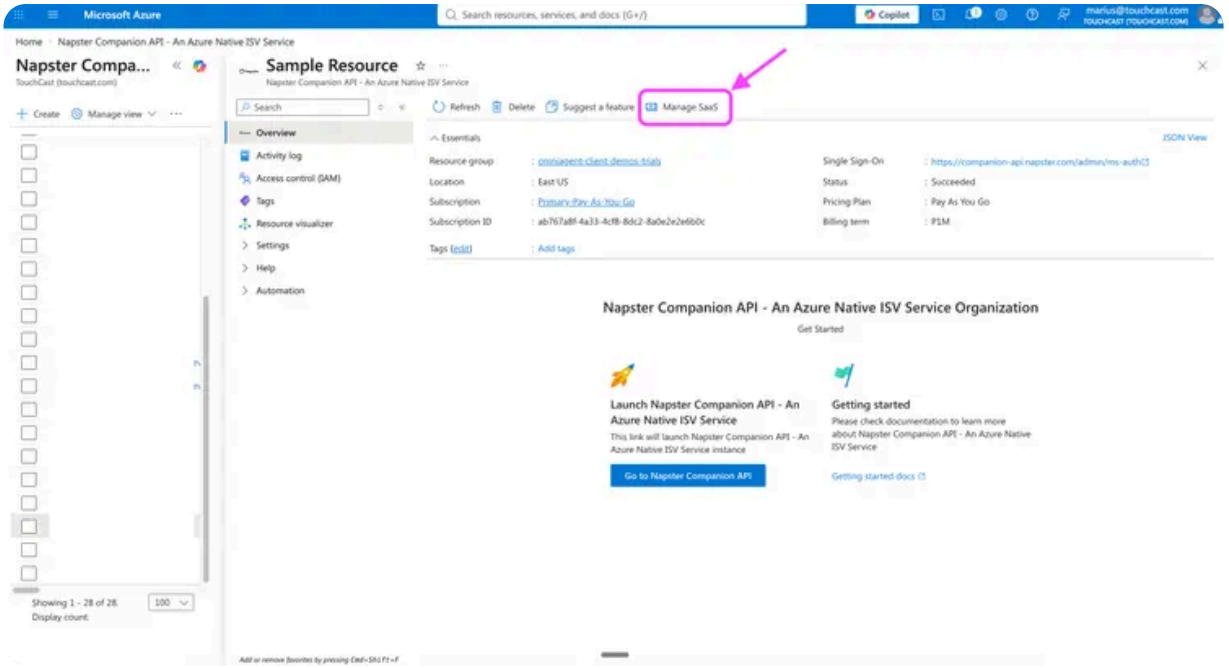


4 Open your existing resource and click Manage SaaS

Open the list of your Napster API resources:

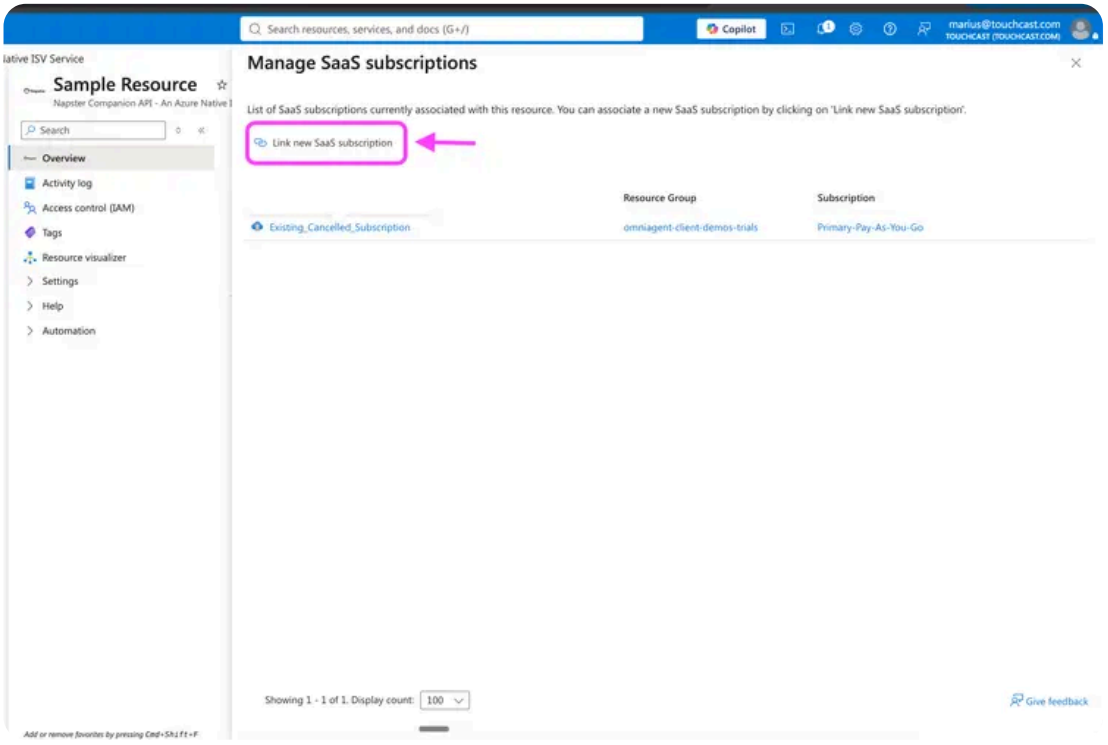
https://portal.azure.com/#browse/napster.companionapi%2Forganizations

Find the resource you originally created and open it. On the resource's **Overview** page, click **Manage SaaS** in the top command bar.



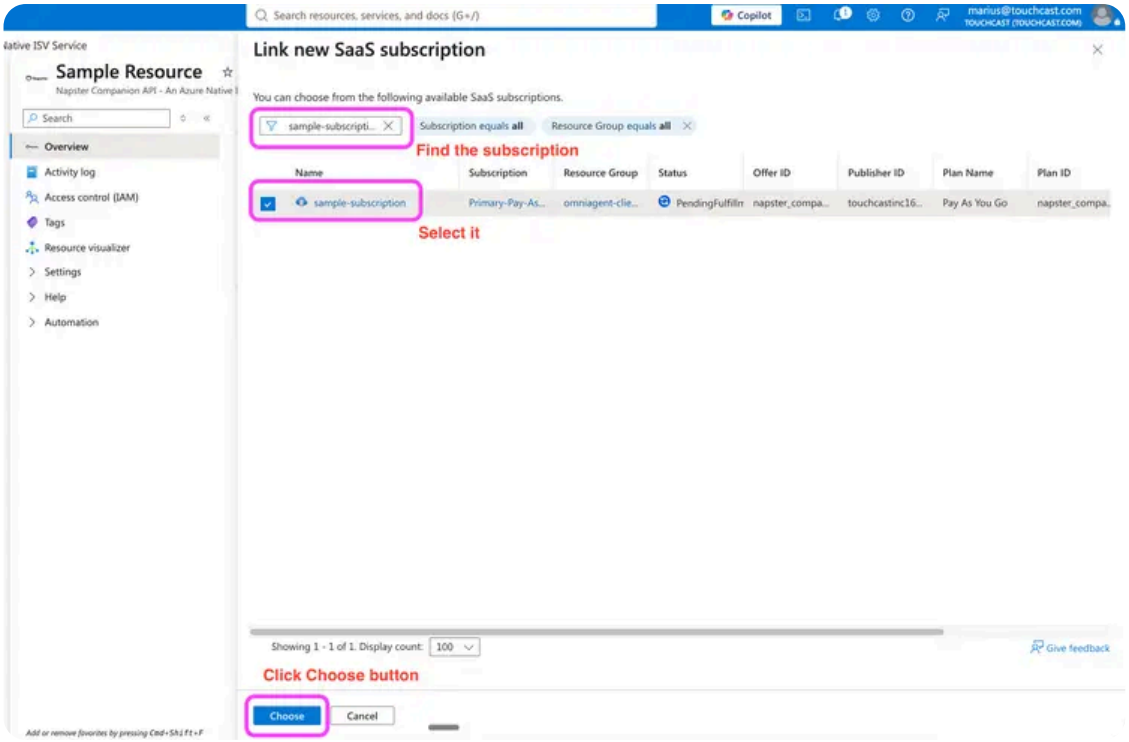
5 Link the new SaaS subscription

The **Manage SaaS subscriptions** panel lists the subscriptions associated with this resource — you'll see your old, canceled one. Click **Link new SaaS subscription**.



6 Select the new subscription and choose

Find the subscription you created in step 1 — use the filter box if you have many — select it (its status still shows *Pending fulfillment*), and click **Choose**.



The new subscription is now linked to your existing resource. Activation completes on the Napster side, and your resource — with all its companions, agents, and keys — is back in service. Billing starts once the subscription is activated.

Because you turned **auto-renew on** in step 1, this subscription renews automatically — you shouldn't end up back on this page next month.

Enable Digital Twins

Grant consent in the dashboard before creating digital twins through the API

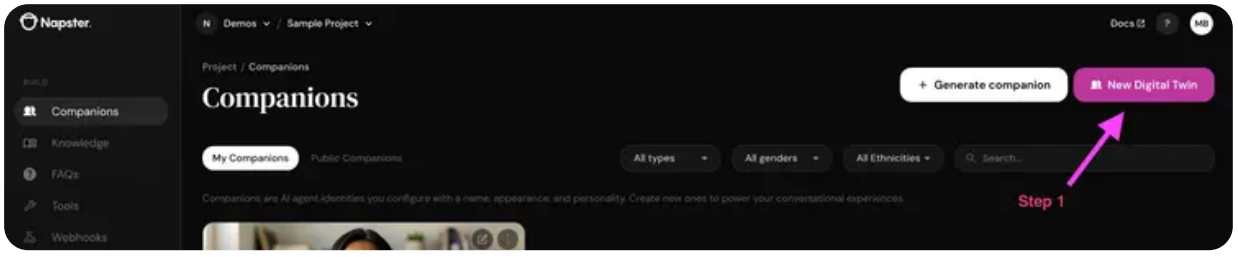
Digital twins replicate a real person's likeness and voice, so before you can create them through the API, an organization admin must grant consent in the dashboard. This is a one-time confirmation that applies across every project in your organization.

> ⚠ Until consent is granted, `POST /public/digital-twins` returns a `DigitalTwinConsentRequired` error. Only an organization admin can grant consent.

The consent dialog is reached by starting a new digital twin in the dashboard. You don't have to finish creating one — once you agree, consent is granted and you can close the flow.

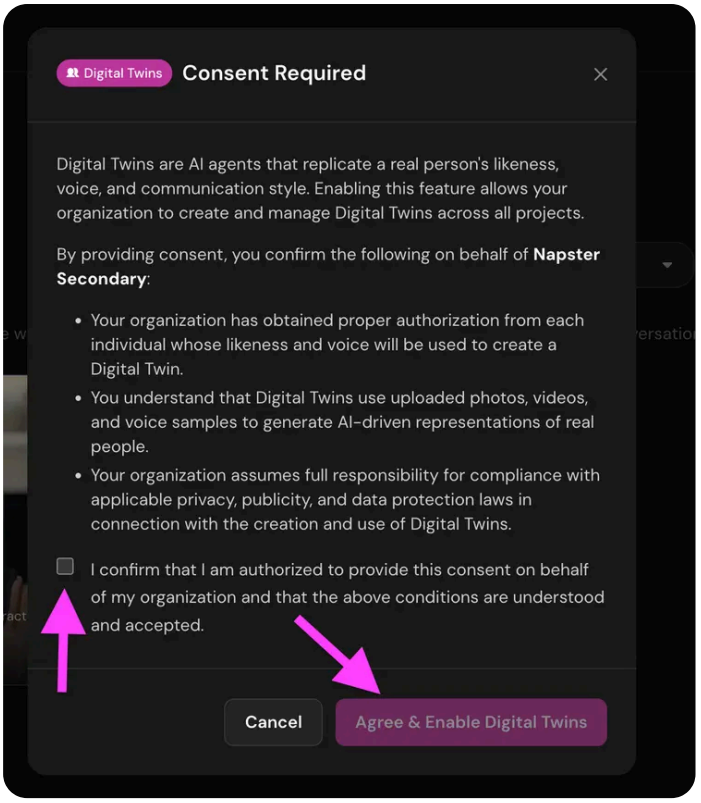
1 Start a new digital twin

In the dashboard, open **Companions** and click **New Digital Twin**.



2 Agree to the consent terms

In the **Consent Required** dialog, check the confirmation box and click **Agree & Enable Digital Twins**.



Once consent is granted, `POST /public/digital-twins` works for everyone in your organization.

Next steps

- Digital Twins — create a digital twin through the API

Video Recordings

Generate a pre-recorded video of your Omniagent speaking — video greetings, training clips, onboarding, and announcements

A recording is a pre-recorded video of your Omniagent's avatar speaking text you provide — a video greeting, a training or onboarding clip, or an announcement. Use it anywhere a fixed, pre-recorded clip works better than a live session, so you get the same polished delivery every time.

You supply either text and a voice, or an audio track; the API renders a lip-synced video of the companion that you can preview, download, and reuse. Recordings are generated asynchronously: each one starts as a job that moves through a few states before the video is ready.

Generating a recording

Call `POST /public/companions/{companionId}/recordings` with a voice and the text for the avatar to speak.

```
curl -X POST https://companion-api.napster.com/public/companions/comp_abc123/recordings \
  -H "X-Api-Key: $NAPSTER_API_KEY" \
  -H "Content-Type: application/json" \
  -d '{
    "voiceId": "alloy",
    "text": "Welcome to Acme. In this short clip, I will walk you through everything you
need to get started."
  }'
```

The response returns the recording with a `status` of `pending` — the video is rendered in the background.

```
{
  "id": "rec_xyz789",
  "provider": "openAi",
  "voiceId": "alloy",
  "text": "Welcome to Acme. In this short clip, I will walk you through everything you need
to get started.",
  "audioUrl": null,
  "fromFile": false,
  "previewUrl": null,
  "status": "pending",
  "createdAt": 1721433600,
  "updatedAt": 1721433600,
  "tags": null,
  "companion": {
    "id": "comp_abc123",
    "firstName": "Ava",
    "lastName": "Reed"
  }
}
```


Request parameters

Parameter	Type	Required	Description
`text`	string	Conditional	The text for the avatar to speak, 84–30,000 characters. Required unless you pass `audioUrl`.
`voiceId`	string	Conditional	The voice to speak with — for example `alloy`, `echo`, or `shimmer`. Required with `text`, except for digital twins, which use their cloned voice when you omit it.
`provider`	string	No	The voice provider: `openAi` (default) or `elevenLabs`. Use `elevenLabs` to pick a digital twin's cloned voice explicitly, passing its `externalId` from the companion's `customVoices` as `voiceId`.
`audioUrl`	string	Conditional	An `http` or `https` URL of an audio file for the avatar to lip-sync to. Use instead of `text` — see Generating from audio.
`tags`	object	No	Arbitrary key–value metadata to attach to the recording. Useful for filtering later.

Pass exactly one of `text` or `audioUrl`.

Generating from audio

Instead of text, the avatar can lip-sync to audio you provide — a voice-over or a recorded message. The recording uses the audio as is, so `voiceId` and `provider` don't apply.

Pass a link to the file as `audioUrl`:

```
curl -X POST https://companion-api.napster.com/public/companions/comp_abc123/recordings \
  -H "X-Api-Key: $NAPSTER_API_KEY" \
  -H "Content-Type: application/json" \
  -d '{
    "audioUrl": "https://assets.acme.com/audio/welcome.mp3"
  }'
```

Or upload the file directly with `POST /public/companions/{companionId}/recordings/from-file` as `multipart/form-data`:

```
curl -X POST https://companion-
api.napster.com/public/companions/comp_abc123/recordings/from-file \
  -H "X-Api-Key: $NAPSTER_API_KEY" \
  -F "File=@welcome.mp3" \
  -F "Tags[campaign]=onboarding"
```

Both return the recording with a `status` of `pending`. Uploaded recordings have `fromFile` set to `true`.

	`audioUrl`	**Upload (`from-file`)**
Maximum size	50 MB	2 MB
Formats	MP3, WAV, M4A, FLAC, OGG, WebM, MP4	MP3, WAV, M4A, FLAC, OGG, WebM, MP4

Recording lifecycle

A recording moves through these `status` values:

Status	Meaning
`pending`	The job is queued.
`processing`	The avatar video is being rendered.
`uploading`	Rendering finished; the video is being stored.
`completed`	The video is ready. `previewUrl` is populated and the download URL is available.
`failed`	Rendering did not succeed.

Poll the recording (see below) until `status` is `completed`, then use `previewUrl` to play it inline or request a download URL for the video file.

Retrieving a recording

Fetch a single recording by ID to check its status or read back its details.

```
curl https://companion-api.napster.com/public/companions/comp_abc123/recordings/rec_xyz789 \
  -H "X-Api-Key: $NAPSTER_API_KEY"
```

```
{
  "id": "rec_xyz789",
  "provider": "openAi",
  "voiceId": "alloy",
  "text": "Welcome to Acme. In this short clip, I will walk you through everything you need
to get started.",
  "audioUrl": null,
  "fromFile": false,
  "previewUrl": "https://cdn.napster.com/recordings/rec_xyz789.mp4",
  "status": "completed",
  "createdAt": 1721433600,
  "updatedAt": 1721433742,
  "tags": null,
  "companion": {
    "id": "comp_abc123",
    "firstName": "Ava",
    "lastName": "Reed"
  }
}
```


Listing recordings

List the recordings for a companion. The response is paginated and supports several filters.

```
curl "https://companion-api.napster.com/public/companions/comp_abc123/recordings?
pageSize=20" \
  -H "X-Api-Key: $NAPSTER_API_KEY"
```

```
{
  "items": [
    {
      "id": "rec_xyz789",
      "provider": "openAi",
      "voiceId": "alloy",
      "previewUrl": "https://cdn.napster.com/recordings/rec_xyz789.mp4",
      "status": "completed",
      "createdAt": 1721433600,
      "updatedAt": 1721433742,
      "tags": null,
      "companion": {
        "id": "comp_abc123",
        "firstName": "Ava",
        "lastName": "Reed"
      }
    }
  ],
  "filteredCount": 1,
  "totalCount": 1,
  "pageIndex": 0,
  "pageSize": 20,
  "search": null
}
```


To list recordings across every companion in your project, call `GET /public/recordings`. It returns the same shape and accepts the same filters, plus `companionId`:

```
curl "https://companion-api.napster.com/public/recordings?
companionId=comp_abc123&pageSize=20" \
  -H "X-Api-Key: $NAPSTER_API_KEY"
```

Query parameters

Parameter	Type	Description
`companionId`	string	`GET /public/recordings` only. Filter to one companion's recordings.
`provider`	string	Filter by voice provider: `openAi` or `elevenLabs`.
`voiceId`	string	Filter to recordings made with a specific voice.
`search`	string	Free-text search across the recording text.
`tags`	object	Filter by tag values — a map of tag name to a list of matching values, e.g. `tags[campaign]=onboarding`.
`pageIndex`	integer	Zero-based page number.
`pageSize`	integer	Number of results per page.

Downloading a recording

Once a recording is `completed`, request a download URL for the video file. The URL is time-limited, so fetch it when you need it rather than storing it.

```
curl https://companion-
api.napster.com/public/companions/comp_abc123/recordings/rec_xyz789/download-url \
  -H "X-Api-Key: $NAPSTER_API_KEY"
```

```
{
  "url": "https://cdn.napster.com/recordings/rec_xyz789.mp4?token=..."
}
```

Deleting a recording

Remove a recording you no longer need.

```
curl -X DELETE https://companion-
api.napster.com/public/companions/comp_abc123/recordings/rec_xyz789 \
  -H "X-Api-Key: $NAPSTER_API_KEY"
```

> ⚠️ Deleting a recording is permanent. Any content referencing its video URL stops resolving.

Errors

A rejected request returns `400` with one of these codes:

Code	When
`InvalidRecordingTextLength`	`text` is shorter than 84 or longer than 30,000 characters.
`VoiceIdRequired`	You sent `text` without a `voiceId` for a companion that isn't a digital twin.
`ClonedVoiceNotAvailable`	The digital twin's cloned voice isn't ready yet.
`AudioUnsupportedFormat`	The audio isn't in one of the supported formats.
`AudioTooLarge`	The audio is over 50 MB (`audioUrl`) or 2 MB (upload).
`AudioValidationFailed`	The audio couldn't be read, or has no audio track.
`NoAvailableConnections`	No rendering capacity is free right now. Retry later.

Sending both `text` and `audioUrl` , neither of them, or an `audioUrl` that isn't an `http` or `https` URL returns a `400` validation error.

Next steps

- Digital Twins — build a companion from a real person's voice and likeness
- Avatar Models — configure the visual avatar for your companion
- API reference: Recordings — full request and response schemas

Napster.

SECTION 07

FAQs

Answers to the questions developers ask most.

FAQs 287



FAQs

Frequently asked questions about the Napster API

Getting started

How do I get started?

Start by signing up at napster.com/developer — new accounts get $6 of free credits, roughly 100 minutes of conversation. (If you'd rather bill through your Azure subscription, create a Napster API resource in the Azure Portal instead.) Then sign in to the dashboard to create your organization and first project. From there, create a Companion, set up your knowledge and tools, generate an API key, and create an Omniagent that brings everything together. Deploy it to any channel using the Web SDK, WebSocket, VoIP, or SIP. You can test everything in the Playground before writing any integration code.

Do I need my own LLM deployment?

Not necessarily. The platform supports both customer-hosted and Napster-hosted model configurations. With customer-hosted (Azure OpenAI), you provide your own deployment and credentials. With Napster-hosted, Napster manages the model infrastructure and you don't need your own cloud account or deployment. You choose the option when creating an API key.

Can I use the Napster API without the dashboard?

Yes. Once your API keys, companions, knowledge, tools, and FAQs are set up, developers interact with the product through the REST API and the connectivity layer (Web SDK, WebSocket, or SIP). The dashboard is primarily a management and testing tool.

Can I test my agent without writing code?

Yes. The Playground in the dashboard lets you have a live video and voice conversation with any agent you assemble. No integration required. It's designed for rapid testing and iteration.

What's the difference between Admin and Member roles?

Admins have full access to all dashboard features including API Keys, Settings, billing, member management, and project management (create, edit, delete). Members can browse companions, use the Playground, and view existing projects, but do not have access to API Keys, Settings, or the ability to create, edit, or delete projects.

Companions and agents

What is an Omniagent?

An **Omniagent** is a persistent, channel-agnostic AI agent with a consistent voice, personality, knowledge, and memory. You define it once and deploy it to WebRTC, WebSocket, VoIP, or SIP — same agent, every surface. See Get Started to build your first one.

What's the difference between a Companion and an agent?

A **Companion** is the identity layer: the avatar, name, personality, and behavior. On its own, it doesn't do much. An **Omniagent** is what you get when you combine a Companion with a voice, tools, knowledge, FAQs, and model settings via `POST /public/agents`. Think of the Companion as the face, and the Omniagent as the complete package — ready to deploy to any channel.

Can I edit a companion after creating it?

You can modify the companion's behavior by updating the system instructions in the Playground. The avatar and name are defined at creation time. Coming soon, you will have the ability to fully manage the companion's appearance, update its properties, and configure all aspects of it through both the dashboard and the API.

What image should I upload for the companion avatar?

A 16:9 photo showing a person from the waist up, facing the camera with relaxed arms and a gentle smile. The image is used to generate the companion's video avatar. You can use a real person's photo (with their consent), a stock photo, or any suitable portrait. No public figures are allowed.

What avatar views are available?

Three options, rendered through the Web SDK: **Round** (circle, focused on face), **Rectangle** (full body with background), and **Silhouette** (full body, background removed, floating on page).

What voices are available?

Standard voices are available by default, sourced from your LLM provider (Azure OpenAI). Whatever voices the provider supports, the Napster API supports. You set the voice on the Omniagent with the `voiceId` field, or select one in the Playground. For custom voice cloning, create a digital twin with a `voiceUrl` recording of the person.

Knowledge and tools

How do I give my agent knowledge?

Create a Knowledge Collection and upload your documents, either through the dashboard or through the API. Then attach the collection to your Omniagent using the `knowledgeBaseId` field when creating the agent. The agent uses the uploaded content to ground its answers.

What's the difference between implicit and explicit tools?

Implicit tools are handled by your calling application. The platform delivers the tool name and arguments to your application, you handle the logic, and send the result back. **Explicit tools** are forwarded to a server-side endpoint (HTTP or WebSocket URL) that you configure. Your server executes the logic and returns the result. In both cases, the agent uses the returned result to continue the conversation. You choose the execution flow when you create the tool.

Connectivity

What are the different ways to connect to the Napster API?

There are two separate choices: the **channel** (the transport — how you connect) and the **modality** (whether the conversation is audio, text, or video).

Five channels are available: WebRTC via the Web SDK for browser-based audio and video; WebSocket for a persistent connection from any environment; VoIP to answer phone calls via Twilio; SIP to integrate your own SIP trunk; and Kiosk for in-person conversations on a Napster Station.

Modality is a separate axis: WebRTC delivers a video avatar with voice, WebSocket sessions can be audio or text-only, and the phone channels (VoIP, SIP) are voice. See Choosing a Channel for a full breakdown of which to use.

Can I have a text-only conversation with the agent (no voice)?

Yes. Text-only is a **modality** on the WebSocket channel — not a separate transport. Create a WebSocket connection with `modality: "text"` and the session becomes a typed, text-only chat: the user's messages in, the agent's text replies out, with no audio and no avatar. (WebRTC is for audio and video; a text-only chatbot runs over WebSocket.) See WebSockets — Text sessions for the full setup.

Can I use the same agent on multiple channels?

Yes — that's the core idea behind the Omniagent. Create the agent once and deploy it to WebRTC, WebSocket, VoIP, and SIP simultaneously. The same voice, personality, knowledge, and memory are available on every surface. A user can start a conversation on the web and continue it over the phone.

What's the difference between per-session connections and Omniagent?

With **per-session connections** (`POST /public/connections` or `POST /public/ws-connections`), you pass the full configuration — companion, voice, tools, knowledge, provider settings — on every request. Nothing persists between sessions. With an **Omniagent** (`POST /public/agents`), you define the configuration once and create sessions with a minimal call. The per-session endpoints are still available for prototyping. See the migration guide for details.

How do I migrate to Omniagent?

If you're using per-session connections (`POST /public/connections` or `POST /public/ws-connections`), see From Per-Session Connections to Omniagent. If you're using the old SIP Agent model (`/public/sip-agents`), see From SIP Agents to Omniagent.

Is there an inactive session timeout?

Yes. Inactivity timeout automatically disconnects the avatar after a period of user inactivity. Users receive a countdown notification before disconnection. You can try it in the Playground's Advanced Settings, and configure it programmatically through the Web SDK with control over the timeout duration and countdown length.

What does the Disclaimer setting do?

When enabled through the Web SDK, it displays a customizable message to the user (default: "This is an AI-generated avatar"). You can edit the text to suit your use case or compliance requirements.

What are the rate limits?

Connection endpoints (`POST /public/connections`, `POST /public/ws-connections`, `POST /public/agents/{agentId}/connections`, and `DELETE /public/connections/{connectionId}`) share a rate limit of 1 request per second per API key. Exceeding this returns a `429` status code. See Limits for details.

How many concurrent sessions can I run?

Each organization can run up to 5 concurrent sessions per transport — 5 WebRTC, 5 WebSocket, 5 VoIP, and 5 SIP, for a total of 20. Exceeding this returns a `409` status code. For higher limits, contact us.

What happens when a phone caller hits the concurrency limit?

On VoIP and SIP, the system plays a pre-recorded message informing the caller that no agents are available and asks them to try again later. The call is then dropped.

Billing and data

How is billing and pricing handled?

Billing works two ways: buy credits directly with a card (sign up at napster.com/developer; top up or auto-recharge from the dashboard), or bill through the Azure Marketplace. No setup fees, no commitments. You pay only for the minutes your agents are active. Pricing depends on your setup: if you bring your own LLM, you pay $0.01/min ($0.60/hour). If you use a Napster-hosted model, you pay approximately $0.058/min ($3.50/hour). You can view usage in the dashboard; manage your balance there too (direct billing), or manage your subscription from the Azure Portal (Marketplace billing).

Does Napster have access to my data or models?

With a customer-hosted deployment, Napster does not access your LLM or its inference data. AI inference runs entirely in your cloud environment. However, Napster stores the agent configuration (system prompts, personality, instructions) on its platform and handles video rendering and agent orchestration. With a Napster-hosted deployment, Napster manages the model infrastructure on your behalf.

Can I have multiple API keys?

Yes. You can create as many API keys as you need, each linked to a different LLM provider or deployment. API keys are scoped to a project, so you can use projects to separate environments (Production, Staging, Development) or applications.

Napster

SECTION 08

What's New

Release history for the platform, the Web SDK, and Edge MCP.

PLATFORM & API

September 29, 2026 294

August 19, 2026 300

August 12, 2026 303

July 13, 2026 306

June 26, 2026 309

June 1, 2026 311

May 7, 2026 314

WEB SDK

August 12, 2026 318

EDGE MCP

August 12, 2026 320



September 29, 2026

Cascade architecture, puppeteer mode, companion webhooks, route SIP calls to different teams, end live sessions from your backend, delete sessions, video recordings from audio, MCP token refresh, one FAQ collection per agent, and tool connection data on a session is now `functionStates`

Refresh MCP tokens mid-session

When a per-user MCP server's access token is about to expire, you can now send a fresh one without ending the session. Send `set_settings` with `mcp.authorizations` from your client — the agent uses the new token from its next call to that server.

```
{
  "type": "set_settings",
  "data": {
    "mcp": {
      "authorizations": [
        {
          "mcp_server_id": "crm",
          "token": "ya29.a0AfH6_refreshed..."
        }
      ]
    }
  }
}
```

See MCP Servers — Refreshing a token mid-session.

Cascade architecture: choose each model

Cascade is a new architecture for API keys. Instead of one realtime model, speech recognition, the language model, and the voice run as separate stages, and you choose the provider for each:

Stage	Providers
Speech recognition	Azure OpenAI
Language model	Azure OpenAI, Anthropic
Voice	Azure OpenAI, OpenAI

Azure OpenAI stages can run on Napster's managed infrastructure; Anthropic and OpenAI stages use your own API key. Cascade also unlocks puppeteer mode. Some features work differently on Cascade — for example, MCP servers aren't supported.

See Create an API key — Architectures and Cascade architecture.

Puppeteer mode: the agent speaks your lines

A new `mode` setting controls how the agent speaks. `conversation` is the default. In `puppeteer` mode, the agent speaks only the lines your client sends — use it for scripted presentations, guided tours, or when your own system decides every line.

Set `mode` on the agent, or per session on `POST /public/connections` and `POST /public/ws-connections`:

```
curl -X PATCH https://companion-api.napster.com/public/agents/agent_abc123 \
  -H "X-Api-Key: $NAPSTER_API_KEY" \
  -H "Content-Type: application/json" \
  -d '{
    "mode": "puppeteer"
  }'
```

Then send each line with the new `talk` command over the session's data channel, and follow it with `talk_state_changed` events, which carry your `talkId` back as `talk_id`. Puppeteer mode requires an API key that uses the Cascade architecture.

See Configuration — Mode and Client commands — `talk`.

Companion webhooks

Webhooks now also fire for companions: `companion.created`, `companion.updated`, and `companion.deleted`. Use them to react when a companion you created finishes generating, instead of polling its status. Each status change while it generates arrives as `companion.updated`, from `generationCompleted` through `readyToUse` to `completed`. Companion events cover the companions your project owns, not the Napster stock catalog.

Subscribe to the new events on your webhook in the dashboard under **Webhooks**. Session webhooks (`session.started` and `session.closed`) work as before.

Webhooks also have their own guide now, covering setup, payloads, signature verification, and retries. See Webhooks.

Route SIP calls to different teams

Human handoff on SIP telephony can now send callers to different extensions depending on what they need. Add `routes` to the SIP channel config — up to 10, each an `extension` with a `purpose` — and the agent transfers to the route whose purpose clearly matches the caller's need. Otherwise, the call goes to `transferExtension`, which is now required when handoff is enabled or routes are set.

```
curl -X PUT https://companion-api.napster.com/public/agents/agent_abc123/channels/sip \
  -H "X-Api-Key: $NAPSTER_API_KEY" \
  -H "Content-Type: application/json" \
  -d '{
    "humanHandoff": {
      "enabled": true,
      "transferExtension": "200",
      "routes": [
        {
          "extension": "210",
          "purpose": "Billing questions, invoices, and refunds"
        },
        {
          "extension": "220",
          "purpose": "Technical support for devices and connectivity"
        }
      ]
    }
  }'
```

See SIP — Routing to different teams.

End a live session from your backend

You can now close a running session with `DELETE /public/connections/{connectionId}`, using the `connection.id` you received when you opened it. The session ends right away and shows up as `closed` with `closeReason: "connection_aborted"`. It works for `webrtc`, `websocket`, and `kiosk` sessions.

```
curl -X DELETE https://companion-api.napster.com/public/connections/sess_xyz789 \
  -H "X-Api-Key: $NAPSTER_API_KEY"
```

See Session configuration — Ending a session.

Delete a session and its data

`DELETE /public/sessions/{sessionId}` permanently removes a finished session, together with its conversation transcript and the memory stored from it. Use it when an end user asks you to delete their data. Only `closed` or `failed` sessions can be deleted.

```
curl -X DELETE https://companion-api.napster.com/public/sessions/sess_abc123 \
  -H "X-Api-Key: $NAPSTER_API_KEY"
```

See Sessions — Deleting a session.

Video recordings from audio, and a project-wide list

Recordings can now be generated from an audio track instead of text: the avatar lip-syncs to a voice-over or message you provide. Pass `audioUrl` (up to 50 MB), or upload the file to the new `POST /public/companions/{companionId}/recordings/from-file` endpoint (up to 2 MB).

```
curl -X POST https://companion-api.napster.com/public/companions/comp_abc123/recordings \
  -H "X-Api-Key: $NAPSTER_API_KEY" \
  -H "Content-Type: application/json" \
  -d '{
    "audioUrl": "https://assets.acme.com/audio/welcome.mp3"
  }'
```

To list all recordings in your project, use the new `GET /public/recordings` endpoint.

See Video Recordings.

One FAQ collection per agent

An agent can now have one FAQ collection attached. Passing more than one ID in `faqCollections` returns a `400` validation error with the message "At most 1 FAQ collection can be attached." The limit applies to agents, channel configs, and connections.

```
curl -X PATCH https://companion-api.napster.com/public/agents/agent_abc123 \
  -H "X-Api-Key: $NAPSTER_API_KEY" \
  -H "Content-Type: application/json" \
  -d '{
    "faqCollections": ["faq_support"]
  }'
```

If you split your Q&A pairs across several collections, merge them into one. See Limits — FAQ collections per agent.

`functionMetrics` is now `functionStates`

The per-tool connection record on a session is renamed from `functionMetrics` to `functionStates`, and its shape is simpler. It used to be an object keyed by function library ID, with an array of tool entries under each ID. It is now one flat array covering every WebSocket-based tool in the session.

```
curl https://companion-api.napster.com/public/sessions/sess_abc123 \
  -H "X-Api-Key: $NAPSTER_API_KEY"
```

```
{
  "id": "sess_abc123",
  "functionStates": [
    {
      "name": "get_order_status",
      "state": "connected",
      "startedAt": "2026-09-29T10:15:02Z",
      "connectedAt": "2026-09-29T10:15:03Z"
    }
  ]
}
```

Each entry keeps the same fields — `name`, `state`, `startedAt`, `connectedAt`, `failedAt`, `canceledAt`, and `error` — so updating your code means reading `functionStates` and dropping the library-ID level. The field is returned by `GET /public/sessions/{sessionId}`.

See Sessions — Tool connection states.

August 19, 2026

Attach remote MCP servers to an Omniagent, use ready-made connectors for Gmail, Google Calendar and other services, and approve tool calls before they run

Your agent can use remote MCP servers

Point an Omniagent at a server that speaks the Model Context Protocol and it can call that server's tools mid-conversation. You write no tool definitions and handle no tool calls — the provider connects to the server, discovers what it offers, and runs the calls itself.

Register the server once:

```
curl -X POST https://companion-api.napster.com/public/mcp-servers \
  -H "X-Api-Key: $NAPSTER_API_KEY" \
  -H "Content-Type: application/json" \
  -d '{
    "id": "inventory",
    "url": "https://mcp.acme.com/sse",
    "headers": {
      "Authorization": "Bearer sk_live_abc123"
    },
    "allowedTools": ["check_stock", "get_price"],
    "requireApproval": "never"
  }'
```

Then attach it to an agent with the new `mcp` field:

```
curl -X PATCH https://companion-api.napster.com/public/agents/agent_abc123 \
  -H "X-Api-Key: $NAPSTER_API_KEY" \
  -H "Content-Type: application/json" \
  -d '{
    "mcp": {
      "servers": ["inventory"]
    }
  }'
```

`mcp` is available on agents, channel configurations, and connections, and the servers attached to a session are reported back on the session record.

See MCP — Servers.

Servers that act as the signed-in user

When the data belongs to individual people rather than to your application, register the server with `authorizationRequired: true` and pass that user's OAuth token when you open the session:

```
curl -X POST https://companion-api.napster.com/public/agents/agent_abc123/connections \
  -H "X-Api-Key: $NAPSTER_API_KEY" \
  -H "Content-Type: application/json" \
  -d '{
    "channelType": "webrtc",
    "mcp": {
      "authorizations": [
        {
          "mcpServerId": "crm",
          "token": "ya29.a0AfH6..."
        }
      ]
    }
  }'
```

The token is used for that session only and is never stored. See Per-user authorization.

Ready-made connectors for common services

Connectors are MCP servers the platform already maintains, so there is nothing to register — attach them with `mcp.connectors` and supply the user's token:

```
curl -X PATCH https://companion-api.napster.com/public/agents/agent_abc123 \
  -H "X-Api-Key: $NAPSTER_API_KEY" \
  -H "Content-Type: application/json" \
  -d '{
    "mcp": {
      "servers": ["inventory"],
      "connectors": ["googlecalendar"]
    }
  }'
```

Available today: Dropbox, Gmail, Google Calendar, Google Drive, Microsoft Teams, Outlook Calendar, Outlook Email and SharePoint.

Connectors require the OpenAI provider and are not available on Azure OpenAI. If your Omniagent runs on Azure OpenAI, register the service's own MCP server instead — see Connectors.

Approve tool calls before they run

Register a server with `requireApproval: "always"` and every call pauses for a decision from your client. You receive an `mcp_approval_request` event over the session's data channel naming the server, the tool and its arguments, and answer with the new `send_mcp_approval` command — passing the request's `item_id` and `approve: true` or `false`.

You have 15 seconds to answer before the request is rejected automatically. Two further message types report what the agent is doing: `mcp_tools` lists each server's tools at session start, and `mcp_call` tracks each invocation.

See Server events — MCP tool lifecycle and Client commands — `send_mcp_approval`.

August 12, 2026

Buy the Napster API directly with self-serve sign-up and credits, outbound SIP calls, session metadata always sent to WebSocket tools, tool connection behavior and metrics, a failed session status, and an agent limit

Buy the Napster API directly

You no longer need an Azure subscription to use the Napster API. Sign up at napster.com/developer, and you can create an API key and start building right away:

- New accounts start with **$6 of free credits** — roughly 100 minutes of conversation.
- When the credits run out, top up your balance with a card from the dashboard — or enable auto-recharge and never think about it.

The Azure Marketplace path remains available if you prefer billing through your Azure subscription.

Your agent can place outbound SIP calls

SIP is no longer answer-only. Start a call with `POST /public/sip-connections/{sipConnectionConfigurationId}/call` — the agent dials the number and, optionally, speaks first:

```
curl -X POST https://companion-api.napster.com/public/sip-
connections/$SIP_CONNECTION_ID/call \
  -H "X-Api-Key: $NAPSTER_API_KEY" \
  -H "Content-Type: application/json" \
  -d '{
    "destination": "15551234567",
    "initialSpeech": "Hi, this is Ava from Acme — I'\''m calling about your recent order."
  }'
```

See SIP — Outbound calls.

Session metadata is always sent to WebSocket tools

The `initialize` message your functions server receives at session start now always includes a `metadata` object carrying the connection's context — `tags`, `external_client_id`, and `external_client_profile`. Your server can tie the session to your own user, tenant, or profile without a separate lookup, and without any per-tool opt-in.

See WebSocket Tools — Session metadata for the message shape and field mapping.

Control what a session does when your tool server is slow or down

WebSocket-based tools have a new `connectionBehavior` setting with two flags:

- `connectionBehavior.waitBeforeStart` — the session waits for your tool's socket to fully connect before starting.
- `connectionBehavior.abortOnFailure` — the whole session is aborted if your tool's socket fails to connect.

Both default to `false` — and that is a behavior change worth checking your integration against: previously every session waited for tool sockets and died if one failed to connect. Sessions now start fast and survive a failed tool connection unless you opt back in.

See WebSocket Tools — Connection behavior.

See how each tool's connection went

Session details now include `functionMetrics` — a per-tool record of the WebSocket connection lifecycle: when it started connecting, when it connected, whether it failed (and why, as `{ code, message }`), or whether the user left before it finished connecting.

See Sessions — Tool connection states.

Sessions that never start are now `failed`

The session `status` field has a fourth value: `pending`, `started`, `closed` — and now `failed`, for sessions whose connection was never established. Failed sessions appear in the session list alongside the others; use the `status` field to separate them in your analytics.

See Sessions — Session fields.

Agent limit per organization

Organizations can now hold up to **100 agents**. Creating an agent past the limit returns a `409` with code `AgentLimitExceeded`. The limit can be raised per organization — contact us if you need more.

See Limits — Agents per organization.

July 13, 2026

Kiosk channel and Napster Station, text-only WebSocket sessions, pre-recorded video clips, v2 avatars from a still image, and clearer connection errors

Deploy to a Napster Station with the Kiosk channel

You can now deploy your Omniagent to a **Napster Station** — a dedicated device for in-person, walk-up conversations in lobbies, retail floors, showrooms, and events. The Station pairs a small screen showing the agent's avatar with a large screen that runs your custom **experience**, which the agent drives as it talks. You configure it through the `kiosk` channel.

- `PUT /public/agents/{agentId}/channels/kiosk` — configure the kiosk channel (the experience URL, opening instructions, tools, knowledge, FAQs)
- Kiosk conversations are recorded as sessions with `sessionType: "kiosk"`

Kiosk requires a Napster Station. See napster.com/station for more details, and contact us to get set up.

See the Kiosk guide for details.

Text-only conversations over WebSocket

WebSocket sessions can now run in a **text-only** modality — a typed chat with no voice and no avatar. Set `modality` to `"text"` when you create a WebSocket connection: the user's messages go in and the agent's text replies come back over the same connection, with no audio streaming.

```
curl -X POST https://companion-api.napster.com/public/agents/agent_abc123/connections \
  -H "X-Api-Key: $NAPSTER_API_KEY" \
  -H "Content-Type: application/json" \
  -d '{
    "channelType": "websocket",
    "modality": "text"
  }'
```

`modality` accepts `audio` (the default) or `text`, and is also available on `POST /public/ws-connections`. Text-only is a WebSocket modality, not a separate channel.

See WebSockets — Text sessions and Choosing a Channel for details.

Generate a v2 avatar from a still image

The v2 avatar model now accepts a still image, not only a video. If you don't have a reference video of the person, set `version` to `"v2"` and provide a `pictureUrl` — you get the v2 rendering model without needing to capture and upload video. A reference video (`videoUrl`) still produces the best lip sync and expressions.

```
curl -X POST https://companion-api.napster.com/public/digital-twins \
  -H "X-Api-Key: $NAPSTER_API_KEY" \
  -H "Content-Type: application/json" \
  -d '{
    "firstName": "Sarah",
    "lastName": "Johnson",
    "description": "VP of Customer Success at Acme Corp.",
    "gender": "female",
    "ethnicity": "Caucasian",
    "externalClientId": "sarah-johnson-001",
    "pictureUrl": "https://example.com/images/sarah.png",
    "version": "v2"
  }'
```

See Avatar Models for the full comparison.

Pre-recorded video clips

You can now generate a **pre-recorded video** of your Omniagent's avatar speaking text you provide — a video greeting, a training clip, an onboarding walkthrough, or an announcement. Give it the text and a voice; the API renders a lip-synced, voiced video you can preview, download, and reuse.

```
curl -X POST https://companion-api.napster.com/public/companions/comp_abc123/recordings \
  -H "X-Api-Key: $NAPSTER_API_KEY" \
  -H "Content-Type: application/json" \
  -d '{
    "provider": "openAi",
    "voiceId": "alloy",
    "text": "Welcome to Acme. Here is everything you need to get started."
  }'
```

Recordings are generated asynchronously — poll the recording until its `status` is `completed`, then request a time-limited download URL for the video.

- `POST /public/companions/{companionId}/recordings` — start a recording
- `GET /public/companions/{companionId}/recordings` and `…/{recordingId}` — list and check status
- `GET …/{recordingId}/download-url` — get the video file
- `DELETE …/{recordingId}` — remove it

See the Video Recordings guide for details.

Clearer errors on invalid connections

Creating a connection with invalid settings — an unsupported `voiceId`, malformed provider credentials, or a companion that isn't ready — now returns a `400` response with a descriptive error explaining what to fix. Previously such a request could connect and then fail silently; now it is aborted immediately and the reason comes back on the request.

Handle the `400` on your backend before handing a token to the client. See Session Configuration — Handling invalid configuration.

June 26, 2026

Session cost, opening instructions, and v2 digital twins that take voice from the reference video

Session cost

Session records now include a `cost` field — the billed cost of the session in USD, calculated from the minutes the agent was active and your API key's model configuration.

`cost` is returned on both `GET /public/sessions` and `GET /public/sessions/{sessionId}`. It is `null` while a session is `pending` or `started`, and is populated once the session closes and billing is finalized.

Use it to attribute spend per session, per companion, or per end user — list sessions and sum `cost`, filtering by `companionId` or `externalClientId`.

See the Sessions guide — Session cost for details.

Give the agent opening instructions

You can now instruct the agent on how to start a session, before the user speaks. Pass `initialSpeech` with guidance for how it should open *that* conversation — it's an instruction the agent follows in its own voice, not a verbatim line to read.

```
curl -X POST https://companion-api.napster.com/public/agents/agent_abc123/connections \
  -H "X-Api-Key: $NAPSTER_API_KEY" \
  -H "Content-Type: application/json" \
  -d '{
    "channelType": "webrtc",
    "initialSpeech": "Greet the customer warmly, introduce yourself as Acme Support, and ask
what they need help with."
  }'
```

`initialSpeech` is available on every connection endpoint (`POST /public/agents/{agentId}/connections` , `POST /public/connections` , `POST /public/ws-connections`) and on the channel config (`PUT /public/agents/{agentId}/channels/{channelType}`) to set a default for every session on a channel.

See Session Configuration — Opening instructions for details.

v2 digital twins can take voice from the video

When you create a **v2** digital twin from a video that includes audio, the voice is now cloned from the video's own audio track — you no longer need to provide a separate voice recording. `voiceUrl` becomes optional for `v2` : pass a `videoUrl` with clear speech and it supplies both the appearance and the voice.

```
curl -X POST https://companion-api.napster.com/public/digital-twins \
  -H "X-Api-Key: $NAPSTER_API_KEY" \
  -H "Content-Type: application/json" \
  -d '{
    "firstName": "Sarah",
    "lastName": "Johnson",
    "description": "VP of Customer Success at Acme Corp.",
    "gender": "female",
    "ethnicity": "Caucasian",
    "externalClientId": "sarah-johnson-001",
    "videoUrl": "https://example.com/videos/sarah-speaking.mp4",
    "version": "v2"
  }'
```

Providing a separate `voiceUrl` is still supported — use it when you want to clone the voice from a different or higher-quality recording than the video's audio.

See the Digital Twins guide for details.

June 1, 2026

Web search, digital twins, the HUMAIN provider, and the v2 avatar model

Agents can now search the web

Agents can now search the web during conversations. Web search is enabled by default — set `useWebSearch` to `false` when creating or updating an agent if you want the agent to rely only on its provided knowledge.

The field is available on agent creation, agent updates, channel overrides, and connection requests.

See Agent Configuration — Web search for details.

Create agents from real people

You can now create **digital twins** — companions based on a real person's likeness and voice. Unlike custom companions, a digital twin uses the person's actual voice recording and appearance to produce an avatar that looks and sounds like them.

New endpoint

- `POST /public/digital-twins` — create a digital twin from a voice recording, reference image or video, and identity details

Key fields

- `voiceUrl` (required) — a recording of the person's voice, used to clone it for the avatar
- `externalClientId` (required) — ties the digital twin to a real individual in your system
- `videoUrl` (optional) — reference video for higher quality lip sync and expressions

Once created, a digital twin returns a companion ID that works the same way as any other companion — attach it to an agent and deploy to any channel.

See the Digital Twins guide for details.

Arabic-first LLM with HUMAIN's Allam

The Napster API now supports **HUMAIN** as an LLM provider alongside Azure OpenAI. HUMAIN is powered by Allam, an Arabic-first foundation model built by HUMAIN in Saudi Arabia with strong dialect coverage and cultural alignment — best for Arabic-language agents.

HUMAIN is not enabled by default. Contact us to request access.

To use HUMAIN, select it as the **Provider** when creating a new API key.

Higher quality avatars with video input

You can now choose between two avatar models when creating a custom companion or digital twin. Both produce real-time video avatars — the difference is the reference input you provide and the resulting quality.

v1 (default)

Uses a still image (`pictureUrl`) as input. Fast generation, good quality, and strong photorealism.

v2

Uses a video of the person speaking (`videoUrl`) as input. Highest quality with the best lip sync accuracy, natural facial expressions, and better results with facial hair. Generation can take up to several hours.

New fields

- `version` — set to `"v1"` or `"v2"` when creating a companion. Defaults to `"v1"` if omitted.
- `videoUrl` — URL of a reference video for v2 generation.

Example

```
curl -X POST https://companion-api.napster.com/public/companions \
  -H "X-Api-Key: $NAPSTER_API_KEY" \
  -H "Content-Type: application/json" \
  -d '{
    "firstName": "Alex",
    "description": "A friendly customer support specialist.",
    "videoUrl": "https://example.com/videos/alex-speaking.mp4",
    "version": "v2"
  }'
```

See Avatar Models for the full comparison.

May 7, 2026

The Omniagent, richer session data and transcripts, knowledge file summaries, and companion headlines

Introducing the Omniagent

We've introduced the **Omniagent** — a single AI agent with a consistent voice, personality, knowledge, and memory that works across every channel. Define it once, deploy it to WebRTC, WebSocket, or SIP. Update it once, and every channel picks up the change.

What changed

Previously, every session was assembled from scratch — you passed the companion, voice, tools, knowledge, and provider settings on every `POST /public/connections` request. There was no persistent agent configuration, and SIP had its own separate agent model (`/public/sip-agents`).

Now, the **Agents API** (`/public/agents`) gives you a single resource that holds all your agent's configuration. You create it once, then spin up sessions on any channel with a minimal request.

New endpoints

- `POST /public/agents` — create an Omniagent
- `GET /public/agents` — list all agents
- `GET /public/agents/{agentId}` — get a single agent
- `PATCH /public/agents/{agentId}` — update an agent
- `DELETE /public/agents/{agentId}` — delete an agent
- `POST /public/agents/{agentId}/connections` — create a session from an agent (WebRTC or WebSocket)
- `PUT /public/agents/{agentId}/channels/{channelType}` — set up per-channel overrides
- `GET /public/agents/{agentId}/channels/{channelType}` — get channel config
- `DELETE /public/agents/{agentId}/channels/{channelType}` — remove channel config

Removed endpoints

- `/public/sip-agents/*` — replaced by `/public/agents` + `/public/agents/{agentId}/channels/sip`

Migration guides

- From Per-Session Connections to Omniagent — if you're using `POST /public/connections` or `POST /public/ws-connections`
- From SIP Agents to Omniagent — if you're using `/public/sip-agents`

Getting started

See Get Started to build your first Omniagent.

Richer session data and transcripts

The Sessions API now returns richer data for both listing and retrieving sessions.

New fields

- `sessionType` — the channel used for the session (`webrtc`, `websocket`, `voip`, `sip`)
- `status` — current session state (`pending`, `started`, `closed`)
- `closeReason` — why the session ended (e.g. `idle_timeout`)
- `conversation` — full transcript included when retrieving a single session via `GET /public/sessions/{sessionId}`

Expanded resource objects

Session responses now include full objects with additional details:

- `companion` — returns `id`, `firstName`, `lastName` alongside `companionId`
- `knowledgeBase` — returns `id`, `name` alongside `knowledgeBaseId`
- `faqCollections` — returns array of `id`, `name` objects alongside `faqIds`

The original fields (`companionId`, `knowledgeBaseId`, `faqIds`) are still returned for backward compatibility. No changes to your existing parsing code are required.

See the Sessions guide for updated examples.

Add summaries to knowledge files

You can now add a `summary` to any file in a knowledge collection. Summaries help the agent understand what each file is about and improve the chances of retrieving the right content when answering questions.

New endpoint

- `PATCH /public/knowledge-bases/{knowledgeBaseId}/files/{fileId}/summary` — update a file's summary

New field

- `summary` — returned on file responses (`GET /public/knowledge-bases/{knowledgeBaseId}/files/{fileId}`)

See the Files guide for usage.

Add a headline to companions

Companions now support a `headline` field — a short tagline or subtitle (e.g. "Senior Tech Support Specialist"). The field can be set when updating a companion (`PATCH /public/companions/{companionId}`) and is returned when retrieving companions.

See the Companion guide for details.

August 12, 2026

Persistence mode for multi-page websites, faster session starts, a full Web SDK docs section, and the Add an AI Agent to Your Website guide

Persistence mode

For **multi-page websites**: on a traditional multi-page site, clicking a link triggers a full page reload — which tears down the running session, so the agent restarts on every page. The Web SDK's new **persistence mode** keeps the agent, its state, and the live connection alive as the visitor moves through the site — the conversation continues mid-sentence across page loads. (Single-page apps don't need it: client-side routing never unloads the page, so the session already survives route changes.)

```
NapsterCompanionApiSdk.init(token, {
  persistence: { enabled: true },
});
```

By default the site is wrapped as soon as the session connects; with `iframeOnNavigate: true` the landing page stays fully native and the wrap happens on the first click. Exclusion rules, history syncing, and an opt-in navigation progress bar are all configurable.

See the Persistence guide.

Faster session starts

Sessions now start noticeably faster. Ephemeral connection tokens are issued in under a second (down from a couple of seconds) — time that came straight off the gap between a user hitting start and the agent being live. And with `videoPolicy: "deferred"`, the conversation starts the moment audio is ready, with the video avatar appearing a moment later — the agent feels instant even on slower connections.

A full Web SDK docs section

The Web SDK now has a dedicated **docs section**: installation, initialization, configuration, methods, events, messaging, built-in features, framework examples, persistence, and troubleshooting.

New guide: Add an AI Agent to Your Website

An end-to-end walkthrough for the most common goal: putting an Omniagent on your website — from token endpoint to embedded agent, including making the agent able to *operate* your site, not just talk about it.

See Add an AI Agent to Your Website.

August 12, 2026

Introducing Edge MCP — let an AI agent operate your website through your app's own functions

Introducing Edge MCP — agentify your website

An agent embedded on your website can talk — but when the user says "add the OLED to my cart," the best it can do is point at the button. Edge MCP lets the agent **operate the app itself**: your site declares which of its real operations an agent may run (`products.search`, `cart.add`, `checkout.placeOrder`), and the agent calls them — the cart opens, the page navigates, the order gets placed.

`@napster-corp/edge-mcp` is a thin layer on the WebMCP standard: it polyfills `document.modelContext` in every browser and adds live-state resources and console debugging on top — your registrations stay pure standard WebMCP, so any compatible agent can drive them, the Napster Omniagent included.

What Edge MCP is — and isn't. The rule that explains everything: the agent operates your website through the website's own functions. If the website can do something, the agent can do it. If the website can't, neither can the agent — Edge MCP adds no capabilities your app doesn't already have; it makes the ones you choose callable by an agent.

Start here: the Edge MCP docs — or go end-to-end with Add an AI Agent to Your Website, which combines Edge MCP with the Web SDK to put an agent on your site that can also operate it.