Docs

Architecture

Architecture

Cookie Munch is a monorepo of small, framework-agnostic TypeScript packages. This page maps the pieces so you know where to look when integrating: what runs in the visitor's browser, what runs on the server, and which surface (dashboard API vs. developer API) you should be calling.

The three runtime surfaces

Surface	Who calls it	Auth	Purpose
Embed (consent.js + lazy consent-tcf.js)	Visitor browsers	None (public)	Renders the banner, blocks trackers, records consent
Dashboard API (/api/v1/*)	The Cookie Munch web app	Session cookie	Everything the dashboard UI does — property/banner CRUD, org settings
Developer API (/v1/*)	Your servers, the SDK, the MCP server	API key (fck_…)	Stable, versioned automation surface — the subject of this reference

This documentation section covers the embed (this page, embed scripts, JavaScript API, data-attributes) and the Developer API (authentication onward). The dashboard API is internal and not covered here — if you need to automate something the dashboard can do, use /v1 instead; it is the versioned, supported surface for external integrations.

> Note: The dashboard API and the Developer API are deliberately separate implementations (control-plane.ts vs. dev-api.ts in the server package) even where they overlap (e.g. banner-library CRUD). This means the Developer API can evolve on its own version/compat guarantees without being coupled to dashboard UI changes.

Request flow, end to end

- A visitor loads your page. The <script> tag you installed (see embed scripts) pulls consent.js from the embed origin (https://cdn.cookiemunch.net by default, or your own domain if you're self-hosting).
- consent.js installs the auto-blocker synchronously, before your page's own scripts run, then fetches the site's SiteConfig from GET /config/:cbid on the API origin (https://api.cookiemunch.net).
- Once configured, it renders the banner (or applies implied consent per your region rules), exposes window.CookieMunch (see JavaScript API), and beacons impressions/consent

decisions back to the API — all unauthenticated, since this is public client-side telemetry, not account data.

- If IAB TCF, GPP, or US Privacy is enabled for the site, a small synchronous stub (__tcfapi/__gpp/__uspapi) is installed immediately, and the heavier consent-tcf.js bundle (which encodes/decodes the IAB TC string) loads lazily in the background so the base embed stays small for sites that don't need it.
- Your backend — or a script you run in CI, a cron job, or an internal tool — authenticates with an API key and talks to /v1 (this reference) to read consent stats, export audit logs, manage sites, or drive DSAR/vendor/RoPA workflows.

Packages

The functionality behind these surfaces is split into focused workspace packages (packages/*), published as @cookiemunch/*:

Package	Responsibility
@cookiemunch/core	The embed runtime: consent engine, banner UI, auto-blocker, TCF/GPP/USP bridge, i18n
@cookiemunch/server	HTTP surfaces: the dashboard API, the /v1 Developer API, OpenAPI document, config/consent-log/stats services
@cookiemunch/sdk	createCookieMunch() — a typed /v1 REST client (see JavaScript API for the embed SDK; this is the server-side SDK)
@cookiemunch/mcp	An MCP server exposing /v1 as tools for AI agents
@cookiemunch/saas	Multi-tenant primitives: orgs, sites, members, API keys + scopes, webhooks
@cookiemunch/tcf	IAB TCF v2.2 GVL cache, purpose mapping, TC-string encode/decode
@cookiemunch/dsar	Data-subject-access-request lifecycle
@cookiemunch/governance	Vendor risk register and RoPA (Records of Processing Activities)
@cookiemunch/receipts	Signed ISO/IEC-27560-style consent receipts (HMAC-SHA256)
@cookiemunch/react / @cookiemunch/react-native	Framework bindings over the core embed

Deployment model

Cookie Munch is single-tenant self-hosted: one deployment, one Postgres database, serving all your organizations. There is no cross-account data plane to reason about — an API key is scoped to exactly one org, and every cbid path on /v1 is checked for org ownership server-side (a key from org A gets a 404, not a 403, reading org B's site — the API never confirms another org's resource even exists).

Next: Embed scripts for exact installation, or Authentication & scopes to start calling /v1.

Developer overview

Developer overview

The Cookie Munch API is a plain HTTP/JSON surface — no SDK is required to integrate, though a thin client makes the common calls (fetch config, log consent, read stats) more convenient. Every site is identified by its cbid (Cookie Munch ID), issued when you register the site in the dashboard.

Two credential types cover the whole surface: a session Bearer token for the dashboard and account-management routes, and a per-org API key for the public /v1 developer endpoints (config reads, consent stats, exports). Public embed routes (/config, /consent.js, consent ingest) are unauthenticated by design — the embed itself is public JS running on your visitors' browsers.

A minimal server-side read of a site's current banner config looks like this:

```
curl -H "Authorization: Bearer $COOKIEMUNCH_API_KEY" \
  https://api.cookiemunch.net/v1/config/YOUR_SITE_ID
```

From here, the rest of the developer docs walk through consent logging, webhooks, the banner-library API for reusable designs, and the granular TCF endpoints.

Embed scripts (consent.js)

Embed scripts (consent.js)

Every site is protected by a single <script> tag. It loads consent.js — a small, dependency-free IIFE — which installs prior-blocking, fetches your site's config, and renders the banner.

The install tag

```
<script id="CookieMunch"
  src="https://cdn.cookiemunch.net/consent.js"
  data-api="https://api.cookiemunch.net"
  data-cbid="your-site-id"
  data-blockingmode="auto"></script>
```

Place it as early as possible in <head> — before any analytics/marketing tags — so prior-blocking can intercept them. The generated snippet for a given site is also available programmatically:

```
curl https://api.cookiemunch.net/v1/sites/YOUR_CBID/snippet \
  -H "Authorization: Bearer fck_…"
```

```
{
  "snippet": "<script id=\"CookieMunch\"\n  src=\"https://cdn.cookiemunch.net/
consent.js\"\n  data-cbid=\"your-site-id\"\n  data-blockingmode=\"auto\"></
script>",
  "src": "https://cdn.cookiemunch.net/consent.js",
  "api": "https://api.cookiemunch.net",
  "cbid": "your-site-id",
  "blockingMode": "auto"
}
```

> Note: The element id may be CookieMunch or Cookiebot — the embed looks for either (in that order), then falls back to document.currentScript. This is what makes drop-in migration from Cookiebot possible: existing id="Cookiebot" tags keep working unmodified.

Attributes

Attribute	Required	Values	Description
src	yes	URL	The consent.js embed origin. https://cdn.cookiemunch.net/consent.js by default; your own domain if self-hosting the static asset.
data-cbid	yes	string	The site identifier issued when you create the site. Without it the embed refuses to start.
data-blockingmode	no	auto | manual	Auto-blocking strategy. auto (default) installs the DOM-interception blocker; any other value (including checklist, accepted by the snippet builder for forward-compat) is treated as manual — you mark tags inert yourself.
data-api			

no

URL

Overrides the API origin used for config fetch + beacons + the lazy consent-tcf.js bundle. Only needed when consent.js is served from a separate static CDN than the API — otherwise the embed derives the API origin from its own src.

data-culture

no

BCP-47 tag, e.g. en, fr

Force the initial banner language, overriding auto-detection from navigator.language.

All three functional settings (cbid, blockingmode, culture) can also be supplied via window.CookieMunchConfig = { cbid, blockingmode, culture } (useful when a tag manager can't set data-* attributes) or as query parameters on the script src (?cbid=…&blockingmode=…). Precedence is data-* attribute > window.CookieMunchConfig

src query string.

What consent.js does on load

- Installs prior-blocking synchronously, before fetching anything — so no tracker script that loads later in the page can slip through the window between page-load and config-fetch. See script blocking & data-attributes.
- Fetches the live config from GET <api-origin>/config/:cbid and re-normalizes the blocker/banner/framework settings against it. If the fetch fails (offline, CORS, outage), it falls back to the attribute-only base config rather than leaving the page unprotected.
- Detects the visitor's region (server-provided) and resolves the banner mode (opt-in / opt-out / off) per your geo rules, honoring Global Privacy Control (GPC) as a forced decline where applicable.
- Renders the banner (or applies implied consent silently, depending on mode) and exposes window.CookieMunch — see the JavaScript API.
- Beacons anonymous impression + consent-decision events back to the API origin. These calls are unauthenticated by design (the embed runs in an untrusted browser); authenticated read access to the resulting data goes through consent log, stats & export.

consent-tcf.js (lazy IAB bundle)

When a site has framework: "iab" (TCF), GPP, or US-Privacy enabled, consent.js installs synchronous __tcfapi / __gpp / __uspapi stubs immediately (so any ad tech checking for them on page-load sees a valid, if pending, API), then lazily loads consent-tcf.js in the background. That bundle carries the IAB TC-string encoder/GVL lookup logic (@iabtcf/*-derived), which would otherwise bloat the base embed for the majority of sites that don't need IAB compliance. Once loaded, it re-encodes the TC string on every explicit accept/decline and dispatches it through the stub to any listener (addEventListener('addEventListener', …)) already registered by ad tags.

You never load consent-tcf.js yourself — it is fetched automatically, from the same API origin as the beacons, only when needed.

Cookiebot compatibility

Cookie Munch's embed is designed as a drop-in replacement for Cookiebot's script:

- The install tag accepts id="Cookiebot" in addition to id="CookieMunch".
- window.Cookiebot is exposed as an alias of window.CookieMunch (same object).
- Lifecycle events are dispatched under both CookieMunch<Event> and Cookiebot<Event> names, and both CookieMunchCallback_<Event> and CookiebotCallback_<Event> globals are invoked if defined.
- The legacy data-cookieconsent script-blocking attribute is recognized (see script blocking & data-attributes).

This means most Cookiebot integrations (ad-network wiring, custom onConsentReady handlers) work by swapping the <script> tag with no other code changes.

Next: JavaScript API for window.CookieMunch, or script blocking & data-attributes for prior-blocking markup.

Quick start

Quick start

This page gets a working consent banner live on a site in under five minutes: create an organization, add a property, install the script, and confirm the banner renders.

1. Create your organization

When you first sign in, Cookie Munch creates a default organization for you. You can create more from the organization switcher at the top of the sidebar — click it, then New org. Every property, banner design, and team member belongs to exactly one organization, and switching orgs re-scopes the whole dashboard.

Note: Free-tier accounts may own only 1 organization. See plans & billing for tier limits.

2. Add a property

A property is one domain you protect with a banner. Go to Properties in the sidebar and click Add property:

- Enter the Domain (e.g. acme.com).
- Pick a Platform — Web, iOS, Android, AMP, or Other — so the install instructions match your stack.
- Click Add.

Cookie Munch generates a unique site ID (cbid) for the property, derived from the domain plus a random suffix (e.g. acme-com-9f3k). See properties for verification and management details.

3. Install the script

Click Install on the property card to open the snippet dialog, then copy the <script> tag. See installing the script for the full tag reference and platform-specific placement notes.

4. Design your banner

Open Banner Studio from the sidebar. A new organization starts with a polished default banner — a bottom bar with Accept all / Reject all / Customize, plus a preferences modal — so you have something presentable immediately. Pick a different starting point from Templates, or start editing directly. See Banner Studio for the full editor tour.

5. Publish

Click Publish in the Studio toolbar. This pushes your current draft live to every property the banner design is assigned to — visitors on those domains will see the new banner within seconds (the runtime script polls the config on load).

6. Verify it works

Open your site in an incognito window. You should see the banner on first load. Reject or accept, reload, and confirm the choice is remembered. Every decision is written to the consent log, viewable per-visitor from the dashboard.

Where to go next

- Properties — manage multiple domains and verify ownership.
- Banner library — reuse one design across many sites.
- Prior blocking — stop trackers from firing before consent.
- REST API — automate site and banner management.

JavaScript API

JavaScript API

Once consent.js has loaded (see embed scripts), it exposes a global window.CookieMunch object (aliased as window.Cookiebot for migration compatibility — both point at the exact same object, so either name works everywhere below).

```
// Read the current consent state
const state = CookieMunch.consent;
// !’ { necessary: true, preferences: false, statistics: true,
//     marketing: false, stamp: 'a1b2c3…' }

// React to changes
CookieMunch.onConsentChange((state) => {
  if (state.statistics) loadAnalytics();
});

// Open the banner or the granular preferences view
CookieMunch.show();
CookieMunch.showSettings();

// Withdraw consent and re-block non-essential tags
CookieMunch.withdraw();
```

Note: consent is a getter property, not a method — read it as CookieMunch.consent, not CookieMunch.consent(). This matches the Cookiebot API shape.

Properties

Property	Type	Description
consent	object	Current per-category consent snapshot: { necessary, preferences, statistics, marketing, stamp, ...customCategories }. stamp is the consent-receipt id — see consent log, stats & export.
consented	boolean	true once the visitor has an active, valid consent decision (stored or just made).
declined	boolean	true when the visitor explicitly declined everything.
hasResponse	boolean	true once any decision — explicit or implied — has been recorded for this visitor.
doNotTrack	boolean	Mirrors navigator.doNotTrack === '1' at load time.
regulations	object	The resolved regulation set applicable to the visitor (region-dependent).

Methods

Method	Description
show()	Re-open the banner (first layer).
hide()	Hide the banner without changing consent.
renew()	Re-prompt the visitor — identical to show(), used semantically when re-consent is required (e.g. after a policy version bump).
showSettings()	Open the granular preference center (second layer / modal). Wire this to your footer "Cookie settings" link.
withdraw()	Revoke all non-necessary consent and re-block previously activated tags.
submitCustomConsent(preferences, statistics, marketing)	Programmatically submit a specific per-category decision, bypassing the UI.
runScripts()	Force-run any scripts already unblocked by the current consent state (rarely needed — the engine does this automatically on every decision).
getScript(url, async, callback?)	Inject a <script src="url"> tag directly (helper for code that doesn't want to manage the DOM itself). Does not check consent — call it only after you've confirmed the relevant category yourself.
onConsentChange(callback)	Subscribe to consent changes. Fires on explicit accept/decline and on returning-visitor hydration of stored consent. Returns an unsubscribe function.
goToView(id)	Jump to a named view inside a flow-based (Banner Studio v2) banner. No-op on v1 banners.
getActiveView()	Returns the current view id in a v2 flow banner, or null (v1 banners, or before the banner is mounted).
consentId()	Convenience for CookieMunch.consent.stamp — the receipt id, useful for self-serve erasure/export flows run from the visitor's own page.

onConsentChange semantics

```
const unsubscribe = CookieMunch.onConsentChange((state) => {
  console.log('marketing granted:', state.marketing);
});

// later, if needed:
unsubscribe();
```

The callback fires once per explicit user decision (accept-all, decline-all, or a granular save), and once more on page load for returning visitors whose stored consent is being restored. It does not fire on every internal lifecycle tick — only on genuine state changes — so it's safe to wire directly to tag-loading logic without de-duplicating yourself.

Lifecycle events

For code that prefers DOM events over the callback API (or migrating from Cookiebot), every lifecycle transition is also dispatched as a CustomEvent on window, under both the CookieMunch* and Cookiebot* names, with the consent state in event.detail:

Event	Fires when
CookieMunchOnLoad / CookiebotOnLoad	The embed has finished bootstrapping.
CookieMunchOnDialogInit / CookiebotOnDialogInit	The banner is about to be shown for the first time.
CookieMunchOnDialogDisplay / CookiebotOnDialogDisplay	The banner (or preferences view) becomes visible — including re-opens via show()/ showSettings().
CookieMunchOnAccept / CookiebotOnAccept	The visitor accepted (all or some categories).
CookieMunchOnDecline / CookiebotOnDecline	The visitor declined.
CookieMunchOnConsentReady / CookiebotOnConsentReady	A valid consent state exists — fires for fresh decisions and for returning visitors with stored consent.
CookieMunchOnTagsExecuted / CookiebotOnTagsExecuted	Previously blocked tags have finished being reactivated.

```
window.addEventListener('CookieMunchOnAccept', (e) => {
  console.log('consent state:', e.detail);
});
```

Equivalently, you can define a global callback function named CookieMunchCallback_OnAccept (or the Cookiebot-prefixed variant) and it will be invoked directly — no addEventListener needed. This mirrors Cookiebot's callback convention exactly, for drop-in migration.

Declarative triggers (no JS required)

Any element with data-fc-open="banner" or data-fc-open="preferences" opens the corresponding layer on click, with no wiring needed:

```
<a href="#" data-fc-open="preferences">Cookie settings</a>
<a href="#" data-fc-open="banner">Manage cookies</a>
```

data-cc="show-settings" is also recognized (Cookiebot-compatible alias for data-fc-open="preferences").

Next: script blocking & data-attributes for prior-blocking markup, or back to embed scripts for install-tag details.

Installing the script

Installing the script

Every property is protected by a single <script> tag. This page covers the tag itself, where to place it, and the attributes that control blocking mode and language.

Get your snippet

From Properties, find your site's card and click Install. The dialog shows a tag generated specifically for that property's cbid (site ID). Click Copy snippet to copy it to your clipboard.

A generated tag looks like this:

```
<script id="CookieMunch"
  src="https://cdn.cookiemunch.net/consent.js"
  data-cbid="acme-com-9f3k"
  data-blockingmode="auto"></script>
```

> Note: If your dashboard's embed CDN and API run on different origins, the tag also includes a data-api="https://api.yourdomain.com" attribute — copy the snippet as generated rather than retyping it by hand.

Where to put it

Paste the tag as high as possible in <head>, before any analytics, ads, or embed scripts. Placement matters for two reasons:

- Consent banner timing. The script needs to run early to show the banner before visitors interact with the page.
- Auto-blocking. In auto mode, the script intercepts DOM insertions from the moment it runs — any tracking script that loads before it is invisible to the blocker. See prior blocking for how blocking works.

Tag attributes

Attribute	Purpose
id="CookieMunch"	Fixed ID the runtime and other page scripts use to find the tag. Keep it as generated.
src	The embed script URL, served from your CDN/embed origin.
data-cbid	Your property's unique site ID. Identifies which config/banner to load.
data-blockingmode	auto (default) or manual — see prior blocking.
data-api	Only present when the API origin differs from the embed origin. Leave as generated.
data-culture	Optional. Forces a starting language (e.g. de, fr) instead of auto-detecting from the browser.

Platform notes

The Platform you chose when adding the property (Web, iOS, Android, AMP, or Other) doesn't change the tag — it only tailors the install guidance shown in the dashboard. For standard web platforms (custom HTML, most site builders and CMSs), paste the tag directly into the site's global

<head> include. For tag-manager-driven sites, add it as a custom HTML tag that fires on all pages, set to fire as early as possible (ideally outside of the tag manager's own async loader).

Confirming installation

Reload your site in an incognito/private window. You should see the banner. If you don't see it:

- Open your browser's dev tools and confirm the <script> tag is present in the rendered HTML <head> — not injected later by JavaScript.
- Check the console for network errors loading consent.js — a stale data-cbid or wrong src origin is the most common cause.
- Confirm the property is verified — verification unlocks full dashboard control. See properties.

Next steps

- Prior blocking — configure auto vs manual mode.
- Banner Studio — design what visitors see.
- Data attributes reference — manual-mode markup for scripts and iframes.

Properties (sites & domains)

Properties (sites & domains)

A property is one domain protected by a Cookie Munch banner. The Properties page (sidebar ’ Properties) is where you add, verify, and manage every domain in your organization.

Adding a property

Click Add property and fill in:

- Domain — e.g. acme.com. Cookie Munch normalizes this (strips https://, lower-cases it) and derives a unique site ID (cbid) from it, such as acme-com-9f3k.
- Platform — Web, iOS, Android, AMP, or Other. This only tailors the install guidance shown to you; it doesn't change the embed tag.

Each property card shows its domain, platform, cbid (click to copy), verification status, and quick links to the banner Studio and the cookie declaration table.

> Note: Free-tier organizations can have up to 50 properties — sites are effectively unlimited on every plan. Paid tiers differ on monthly consent events, seats, and advanced features, not domain count. See plans & billing.

Verifying ownership

A newly-added property is Pending until you prove you control the domain. Click Verify on the card to open the verification dialog, which offers three methods:

Method	How it works
DNS (TXT record)	Add a TXT record at @ with the value cookiemunch-verify=<token>, then click Verify now.
Meta tag	Paste a <meta name="cookiemunch-site-verification" content="<token>"> tag into your homepage <head>, then click Verify now.
File upload	Upload a token file to a well-known path on your domain, then click Verify now.

Pick whichever method fits your workflow — DNS if you manage your zone file directly, meta tag or file upload if you only have access to the site's HTML. Verification is checked on demand, not continuously, so click Verify now again after you've made the change.

The active property

The sidebar's property switcher (below the organization switcher) shows which property is currently "active" — the one Banner Studio, Cookies, and other per-site pages operate on. Switch it from the dropdown, or click Set active on a property card. Switching is instant (no page reload) and is remembered across sessions on this device.

Installing and managing a property

Each card has an Install button that opens the copy-pasteable <script> snippet for that property — see installing the script for the full tag reference. From a property you can also jump straight to:

- Banner Studio — design or assign the consent banner for this site.
- Cookies — the auto-discovered cookie declaration table for this domain.

Next steps

- Organization & members — invite teammates and manage roles.

- Banner library — assign one reusable banner design to many properties at once.
- Prior blocking — configure how scripts are held back before consent.

Script blocking & data-attributes

Script blocking & data-attributes

Cookie Munch blocks trackers from firing before consent in two complementary ways: auto-blocking (a heuristic DOM interceptor that needs no markup changes) and manual markup (you mark a tag inert yourself, for precise control). Both feed the same reactivation mechanism once consent is granted.

Auto-blocking (data-blockingmode="auto", the default)

With auto-blocking, Cookie Munch patches Node.prototype.appendChild / insertBefore and installs a MutationObserver before your page's own scripts run, so it sees every <script>, <iframe>, <img>, <link>, <embed>, <object>, <audio>, <video>, and <source> element the instant it's inserted — whether by inline HTML, a tag manager, or your own JS.

Each element is checked against a bundled checklist of known tracker domains/patterns (Google Analytics, Meta Pixel, common ad networks, etc.). If it matches a category that isn't yet granted, the element is neutralized in place:

- Scripts get type="text/plain" — the browser won't execute them.
- Other resources get their src moved to data-cookieblock-src and removed — the browser won't fetch them.
- Either way, data-cookieconsent="<category>" is written onto the element, recording why it was blocked.

You don't write any markup for this path — it's fully automatic. Two escape hatches:

Mechanism	Effect
data-cookieconsent="ignore" on an element	Never auto-blocked — useful for your own first-party scripts that happen to match a checklist pattern.
ignoreSelectors in the site's blocking config	CSS selectors that are never inspected at all (config-level allowlist, set via PUT /v1/sites/:cbid/config).

Manual markup (data-blockingmode="manual")

With manual mode, you take full control: mark each tracking tag inert yourself using the same attribute vocabulary the auto-blocker writes, and Cookie Munch reactivates it once the declared categories are granted.

```
<!-- Cookiebot-compatible: comma-separated standard categories -->
<script type="text/plain" data-cookieconsent="statistics"
  src="https://www.google-analytics.com/analytics.js"></script>

<!-- Requires BOTH categories before it runs -->
<script type="text/plain" data-cookieconsent="statistics,marketing"
  src="https://example.com/combined-pixel.js"></script>

<!-- Non-script resource: src goes in data-cookieblock-src -->
<iframe data-cookieblock-src="https://www.youtube.com/embed/xyz"
  data-cookieconsent="marketing"></iframe>
```

Attribute	Applies to	Values	Description

type="text/plain"

<script>

—

Marks the tag inert. Cookie Munch swaps it back to type="text/javascript" (re-creating the node, so the browser actually executes it) once granted.

data-cookieconsent

any blockable element

preferences, statistics, marketing (comma-separated), or ignore

The categories required before this element activates. All listed categories must be granted. ignore opts the element out of auto-blocking entirely (see above).

data-cookieblock-src

non-script resources (<iframe>, <img>, …)

a URL

Holds the real src while blocked; Cookie Munch restores it to src on activation.

data-fc-category

any blockable element

any category id, standard or custom (comma-separated)

Cookie Munch-native alternative to data-cookieconsent — use this when you need a custom category (Banner Studio v2 custom categories aren't valid data-cookieconsent values, which are limited to the four standard categories).

Note: data-cookieconsent only accepts the three user-toggleable standard categories (preferences, statistics, marketing) plus ignore — necessary and custom category ids are rejected there. If your site defines custom categories in Banner Studio, tag those scripts with data-fc-category="my-custom-id" instead.

Placeholder elements

For consent-gated embeds (e.g. "Enable this video" instead of just silently not loading), toggle visibility with these classes — no JS required:

```
<div class="cookieconsent-optin-marketing">
  <!-- shown once marketing is granted -->
</div>
<div class="cookieconsent-optout-marketing">
  Please accept marketing cookies to view this content.
  <!-- shown until marketing is granted -->
</div>
```

Cookie Munch toggles the hidden attribute on any element matching .cookieconsent-optin-<category> / .cookieconsent-optout-<category> every time consent is granted, withdrawn, or re-evaluated (categories: preferences, statistics, marketing).

Migrating from Cookiebot

data-cookieconsent, type="text/plain", and data-cookieblock-src are exact Cookiebot-compatible attribute names — existing manually-marked-up pages work unmodified after swapping the <script> tag. The only Cookie Munch-native addition is data-fc-category, needed solely for custom (non-standard) categories.

Next: JavaScript API to react to consent changes in your own code, or site config to set the blocking mode and ignoreSelectors via the REST API.

Organization & members

Organization & members

Every property, banner design, and consent log belongs to an organization — the top-level container your team collaborates in. This page covers switching between organizations and managing who has access.

Organizations

The organization switcher sits at the top of the sidebar, showing the active org's name and your role badge. Click it to:

- Switch to a different organization you belong to — this reloads the dashboard scoped to that org (properties, banners, members, everything is per-org).
- New org — create another organization. You'll be its Owner.

Note: How many organizations you may own is capped by your account's plan (Free: 1, Starter: 3, Pro: 10, Scale: unlimited). Being invited as a member of someone else's org doesn't count against this limit. See plans & billing.

Team & roles

Go to Account !’ Team to see everyone with access to the current organization. Each row shows their email and role. Click Invite member to add someone:

- Enter their email.
- Choose a role: Admin, Member, or Viewer.
- Click Send invite.

Roles

Role	Badge	What they can do
Owner	Owner	Full control, including billing and member removal. Every org has exactly one at creation; ownership isn't reassigned from this screen.
Admin	Admin	Manage properties, banners, and settings; invite and manage other members (cannot remove the Owner).
Member	Member	Manage properties and banners day-to-day; no access to billing or member management.
Viewer	Viewer	Read-only access — can view the dashboard, consent log, and analytics but not change configuration.

You can change anyone's role (except the Owner's) from the dropdown next to their name in the member list — the change takes effect immediately. To revoke access entirely, click Remove on their row and confirm; they lose access immediately.

Note: Seats are plan-limited (Free: 1, Starter: 3, Pro: 10, Scale: unlimited). The Team page shows a used / limit counter, and invites past the limit prompt an upgrade.

What roles gate

Role checks apply per-organization and cover configuration changes (properties, banner designs, settings, billing, member management). They're enforced on the server, not just hidden in the UI — so a Viewer's API calls are rejected the same way the buttons are disabled for them.

Next steps

- Plans & billing — seat limits, org limits, and feature gating per tier.
- Properties — the domains this organization protects.
- REST API — authenticate as a member/service account to automate org management.

Authentication & scopes

Authentication & scopes

Every request to the Developer API (https://api.cookiemunch.net/v1/*) is authenticated with an API key. There is no separate "organization id" parameter anywhere on this surface — the org is derived from the key itself, so a key can never be pointed at the wrong org, accidentally or otherwise.

Getting a key

Issue a key from Settings !' API keys in the dashboard, or via the API itself once you have one full-access key to bootstrap with:

```
curl -X POST https://api.cookiemunch.net/v1/keys \
  -H "Authorization: Bearer fck_existingkey…"
```

```
{
  "key": "fck_8f2a91c0b3d4e5f6...",
  "prefix": "fck_8f2a91c0"
}
```

> Note: The full key (key) is returned exactly once, at creation. Store it immediately — the server only ever persists a hash. GET /v1/keys later shows you the prefix for identification, never the secret.

Sending the key

Two equivalent forms are accepted on every request:

```
# Bearer token (recommended)
curl https://api.cookiemunch.net/v1/me \
  -H "Authorization: Bearer fck_…"

# ...or the X-API-Key header
curl https://api.cookiemunch.net/v1/me \
  -H "X-API-Key: fck_…"
```

```
{ "orgId": "org_9k2m", "plan": "pro", "keyPrefix": "fck_8f2a91c0" }
```

GET /v1/me is a cheap, no-scope-required identity check — useful for verifying a key works before wiring up real calls.

Scopes

Keys can be issued unscoped (full access to everything below) or scoped to a specific allow-list of resource:action pairs. Scoping follows least-privilege: a CI job that only needs to read consent stats should hold a consent:read-only key, not a full-access one.

Scope	Grants
sites:read / sites:write	List/get sites, read config, read cookies/scan-status/A/B results, get the install snippet, list/create brand kits, read a signed receipt, and export a subject's records (read/write respectively). sites:write also covers create/delete site, verify domain, save config, delete brand kit, trigger a scan, and erase a subject's consent records.
consent:read / consent:write	Consent log, stats, and CSV export (read); preference-center records (read via consent:read, write via consent:write). Receipts, subject-export, and erase-consent are not covered by this scope — despite living under /consent/...-adjacent paths, they're checked against sites:read/sites:write instead (see consent log, stats & export).
dsar:read / dsar:write	

List DSARs (read); create and advance DSARs (write).

vendors:read / vendors:write

List vendors (read); create a vendor (write).

ropa:read / ropa:write

List RoPA entries (read); create a RoPA entry (write).

banners:read / banners:write

List/get account-level banner library entries (read); create/update/delete banner library entries (write).

```
curl -X POST https://api.cookiemunch.net/v1/keys \
  -H "Authorization: Bearer fck_fullaccesskey…" \
  -H "Content-Type: application/json" \
  -d '{ "name": "nightly-export-job", "scopes": ["consent:read"] }'
```

A request whose key lacks the required scope gets a 403:

```
{ "error": "insufficient_scope", "message": "missing required scope:
consent:write" }
```

Note: GET /v1/me and GET /v1/usage need no scope — every key, scoped or not, can call them. Everything else on /v1/members, /v1/keys, and /v1/webhooks — the org-admin surfaces — requires an unscoped (legacy full-access) key; there is no members/keys/webhooks scope to grant, by design. The account-level banner library (/v1/banners/*) has its own banners:read/banners:write scope, so a narrowly-scoped automation key can reach it without also granting sites access.

CORS and public routes

/v1/* responses always include permissive CORS headers (Access-Control-Allow-Origin: *), since these are server-to-server credentials, not cookie-based — there's no session to leak cross-origin. The one unauthenticated route on this surface is GET /v1/openapi.json, which serves the machine-readable OpenAPI 3.1 document for this API (useful for generating your own client, or feeding an API explorer).

```
curl https://api.cookiemunch.net/v1/openapi.json
```

Next: conventions & errors for the request/response shape every endpoint shares, or jump straight to sites.

Conventions & errors

Conventions & errors

Base URL

`https://api.cookiemunch.net/v1`

Self-hosted deployments use their own domain instead — the /v1 prefix is the same. The SDK takes this as baseUrl and appends /v1 for you.

Requests

- JSON in, JSON out. Send Content-Type: application/json with a JSON body on POST/PUT/PATCH requests that take one.
- Path parameters (:cbid, :id) are always the last-known-good identifier — a cbid that doesn't belong to your org behaves exactly like one that doesn't exist (a plain 404), never a 403. This is intentional: the API never confirms the existence of another org's resource.
- There is no pagination on any list endpoint today — GET /v1/sites, GET /v1/dsar, GET /v1/vendors, etc. return the full collection. Time-ranged endpoints (consent log/stats/export) use from/to query parameters instead of pages.
- OPTIONS is handled generically for CORS preflight on every route and returns 204.

```
curl -X PUT https://api.cookiemunch.net/v1/sites/acme-com-9f3k/config \
  -H "Authorization: Bearer fck_…" \
  -H "Content-Type: application/json" \
  -d '{ "blocking": { "mode": "auto" } }'
```

Responses & status codes

Status	Meaning
200	Success (read, or a write that returns the updated resource).
201	Created (site, DSAR, vendor, RoPA entry, webhook, banner, API key).
202	Accepted — an async job (cookie scan) was started; poll for status.
204	Success, no body (delete, OPTIONS preflight, consent-profile write).
400	Malformed request — missing/invalid field. Body includes error.
401	Missing or invalid API key.
403	Authenticated, but forbidden — insufficient scope, or a plan/verification gate.
404	Not found, or it belongs to another org (see above — these are indistinguishable by design).
409	Conflict — e.g. a cbid already claimed, a scan already running, an illegal DSAR status transition, or a banner still assigned to sites.

422

Semantically invalid — e.g. assigning a cbid that isn't in your org to a banner.

501

The endpoint exists in this API version but isn't configured on this deployment (e.g. self-host without cookie scanning wired up).

Every error body has the same minimal shape:

```
{ "error": "site not found" }
```

Scope errors add a machine-readable error code plus a human message:

```
{ "error": "insufficient_scope", "message": "missing required scope:
consent:write" }
```

Using the SDK, every non-2xx response is thrown as a CookieMunchApiError with .status, .message, and an optional .code:

```
import { createCookieMunch, CookieMunchApiError } from '@cookiemunch/sdk';

const fc = createCookieMunch({ apiKey: process.env.FC_KEY, baseUrl: 'https://
api.cookiemunch.net' });

try {
  await fc.sites.get('unknown-cbid');
} catch (e) {
  if (e instanceof CookieMunchApiError && e.status === 404) {
    console.log('no such site (or not yours)');
  }
}
```

Identity & usage

Two meta endpoints work with any key, scoped or not — useful for health checks and usage dashboards without needing a broad grant.

GET /v1/me

```
curl https://api.cookiemunch.net/v1/me \
  -H "Authorization: Bearer fck_…"
```

```
{ "orgId": "org_9k2m", "plan": "pro", "keyPrefix": "fck_8f2a91c0" }
```

GET /v1/usage

Current resource usage against your plan's limits.

```
curl https://api.cookiemunch.net/v1/usage \
  -H "Authorization: Bearer fck_…"
```

```
{ "domains": 12, "seats": 4, "monthlyEvents": 812304 }
```

Field

Description

domains

Number of sites (properties) currently registered in the org.

seats

Number of members in the org.

monthlyEvents

Consent events recorded this billing period.

Note: GET /v1/usage returns 501 on deployments that haven't wired up usage reporting (rare — only relevant for minimal self-host configurations).

Next: sites for the first real resource, or authentication & scopes if you haven't set up a key yet.

Plans & billing

Plans & billing

Cookie Munch prices per organization, not per domain — properties are effectively unlimited on every tier (up to a generous 50/org cap). What differs between tiers is monthly consent events, team seats, how many organizations you may own, and a handful of advanced features. Manage your plan from Account !’ Plan & usage.

Tiers

Plan	Price	Domains	Monthly events	Seats	Orgs you can own
Free	$0/mo	50	50,000	1	1
Starter	$29/mo	50	500,000	3	3
Pro	$99/mo	50	5,000,000	10	10
Scale	$399/mo	50	Unlimited	Unlimited	Unlimited

"Monthly events" counts consent decisions recorded across every property in the organization; "seats" counts team members (see organization & members).

Feature gating

Some features are only available from Pro upward:

Feature
Free
Starter
Pro
Scale
Auto-blocking, Consent Mode, geo-targeting
'
'
'
'
IAB TCF v2.2

'
'
Remove Cookie Munch branding (white-label)

'
'
Enterprise SSO (SAML/OIDC)

'
'
A/B testing

'

Where a gated control appears in the dashboard (e.g. the TCF toggle or the A/B testing switch in Banner Studio settings), it's shown disabled with a badge naming the tier that unlocks it, rather than hidden — so you always know what upgrading buys you. Gating is also enforced server-side when a config is saved, not just in the UI.

Viewing usage

Account ⁉ Plan & usage shows your current plan and a live usage panel for domains, monthly events, and seats, each with a progress indicator. If you're within limits it reads "within limits"; if any dimension is over its cap it flags "over a limit" so you know before you hit a hard wall.

Upgrading or switching plans

In the Plans picker (same page), each tier shows its price and a button:

- Upgrade — for a higher tier than your current one.
- Switch — for a lateral move.
- Free tiers show no button when already active; downgrading below your current tier is done via Manage billing, not the picker.

Clicking a paid tier starts a secure Stripe Checkout session. After payment, you're returned to the dashboard with a confirmation toast and your new plan applied immediately.

Managing billing

Click Manage billing to open the Stripe customer portal, where you can update your payment method, view invoices, downgrade, or cancel. If your organization has no Stripe customer yet (e.g. still on Free), this shows a "No billing set up yet" message instead.

Next steps

- Organization & members — seat limits and roles.
- Banner Studio — where TCF, white-label, and A/B testing are configured once unlocked.

Banner Studio

Banner Studio

Banner Studio (sidebar !’ Banner Studio) is a visual, canvas-based editor for your consent banner — every screen, button, and category, positioned exactly where you want it, with a live preview and one-click publish.

Layout

- Left rail — three tabs: Screens, Elements, Layers.
- Canvas (center) — a freeform drag-and-resize surface showing the selected screen.
- Right rail (Inspector) — properties for whatever is selected: text, colors, actions, spacing.
- Toolbar — a Desktop/Mobile device toggle, Undo/Redo, and Preview / Publish on the right.

Screens & flow

A banner is a small flow of screens (called views), each with a surface type:

Surface	Typical use
bar	A slim strip anchored to an edge — the first thing visitors see.
modal	A centered dialog — usually the detailed preferences screen.
popup	A smaller floating panel — often an intermediate "more options" step.

The Screens tab lists every screen, lets you add/duplicate/reorder/delete them, set the starting screen, and — for multi-region setups — set a different starting screen per region class (EU, US, BR, CA, other). It also overlays each screen's outgoing goto transitions and flags flow problems (orphaned screens with no way in, dead-ends with no way out) inline so you can fix them before publishing.

Buttons drive transitions via an action: goto (to another screen), acceptAll, rejectAll, savePrefs, back, or close.

Elements

The Elements tab is your palette — drag any of these onto the canvas:

Element	Purpose
heading, text	Banner copy.
logo, image	Brand assets.
divider, spacer	Visual structure.
button, link, policyLink	Actions and links (e.g. to your cookie policy).
categoryToggle	A single on/off switch for one category.

categoryGroup

The full list of categories with toggles, auto-generated.

cookieTable

The live cookie declaration table for this property.

customHtml

Escape hatch for anything else.

tcfPanel, tcfVendorCount

IAB TCF v2.2 granular consent UI (Pro).

Every element has independent desktop and mobile positions/sizes, so you can fine-tune the mobile layout without disturbing desktop. The Layers tab lists all elements on the current screen in stacking order for quick selection and reordering.

Editing

Click an element on the canvas to select it — its properties open in the Inspector on the right (text content, action, variant, colors, spacing). Drag to move, drag a handle to resize; the canvas snaps to alignment guides as you go. Use the Desktop/Mobile toggle in the toolbar to switch which breakpoint you're positioning.

Settings drawer

Click Settings to open configuration that isn't drawn on the canvas, organized into tabs:

- Behaviour — blocking mode, re-ask/expiry, bot handling. See prior blocking.
- Consent — Google Consent Mode, IAB TCF v2.2, US state privacy (GPP/US Privacy String).
- Targeting — language auto-detect/default, geo consent rules, A/B testing (Scale).
- Appearance — white-label branding, custom CSS, mobile breakpoint. See theming & appearance.
- Categories — define custom cookie categories beyond the standard three. See templates & presets.

Preview & publish

Preview opens an overlay simulating the live banner without affecting your site. Publish pushes the current draft live — to every property the currently-loaded banner design is assigned to. Visitors see the updated banner on their next page load.

> Note: Publish is disabled until the active site's stored config has finished loading, so publishing can never accidentally overwrite settings (blocking mode, Consent Mode, geo rules) with blank defaults.

Next steps

- Banner library — save a design once, reuse it across sites.
- Templates & presets — fast starting points.
- Theming & appearance — colors, fonts, and brand matching.

Sites

Sites

A site (identified by its cbid, or "Cookie Munch ID") is one domain or app property protected by a banner. This page covers creating, listing, and managing sites themselves. Related sub-resources have their own pages: site config, consent log, stats & export, and cookies, scan & verify.

Requires the sites scope (authentication & scopes): sites:read for the GETs below, sites:write for POST/DELETE.

GET /v1/sites

List every site in your org.

```
curl https://api.cookiemunch.net/v1/sites \
  -H "Authorization: Bearer fck_…"
```

```
[
  {
    "cbid": "acme-com-9f3k",
    "orgId": "org_9k2m",
    "domain": "acme.com",
    "verified": true,
    "verifyToken": "fcv_7a2e9c1b4d8f0e2a1c3b5d7f",
    "verifyMethod": "dns",
    "verifiedAt": 1719858421000
  }
]
```

POST /v1/sites

Create a site. cbid is optional — omit it and the server generates one.

Field	Type	Required	Description
domain	string	yes	The domain to protect, e.g. "acme.com".
cbid	string	no	Custom site identifier. Auto-generated when omitted.
platform	web | ios | android | amp | other	no	The property type. Defaults to "web" when omitted or invalid.

```
curl -X POST https://api.cookiemunch.net/v1/sites \
  -H "Authorization: Bearer fck_…" \
  -H "Content-Type: application/json" \
  -d '{ "domain": "acme.com" }'
```

```
{ "cbid": "dv-m3x8k2q1-a9f2b3c8", "orgId": "org_9k2m", "domain": "acme.com",
"platform": "web" }
```

201 on success. A 409 means the requested cbid is already claimed (the error message is

intentionally generic — it never reveals whether a cbid belongs to your org or someone else's).

GET /v1/sites/:cbid

```
curl https://api.cookiemunch.net/v1/sites/acme-com-9f3k \
  -H "Authorization: Bearer fck_…"
```

```
{
  "cbid": "acme-com-9f3k",
  "orgId": "org_9k2m",
  "domain": "acme.com",
  "verified": true,
  "verifyToken": "fcv_7a2e9c1b4d8f0e2a1c3b5d7f",
  "verifyMethod": "dns",
  "verifiedAt": 1719858421000
}
```

Field	Description
cbid	Site identifier — use this in every sub-resource path.
domain	The registered domain.
verified	Whether domain ownership has been proven. Some endpoints (consent export, receipts) require verified: true — see cookies, scan & verify.
verifyToken	The token to place per your chosen verification method.
verifyMethod	Which method last verified the domain (dns, meta, file, or embed — unset until verified).
verifiedAt	Epoch-ms of verification.

404 if the cbid doesn't exist, or belongs to another org.

DELETE /v1/sites/:cbid

```
curl -X DELETE https://api.cookiemunch.net/v1/sites/acme-com-9f3k \
  -H "Authorization: Bearer fck_…"
```

204 on success. This is permanent — the site's config, consent log, and cookie declaration are removed with it.

GET /v1/sites/:cbid/snippet

Returns the exact <script> install tag for a site (see embed scripts for what each attribute means), so you can generate installation instructions programmatically instead of hardcoding the tag format.

Query param	Values	Description
blockingmode	auto | manual | checklist	Overrides the emitted data-blockingmode. Default auto.
culture	BCP-47 tag, e.g. fr	Emits data-culture when set.

```
curl "https://api.cookiemunch.net/v1/sites/acme-com-9f3k/snippet?
```

```
blockingmode=auto" \
  -H "Authorization: Bearer fck_…"
```

```
{
  "snippet": "<script id=\"CookieMunch\"\n  src=\"https://cdn.cookiemunch.net/
consent.js\"\n  data-cbid=\"acme-com-9f3k\"\n  data-blockingmode=\"auto\"></
script>",
  "src": "https://cdn.cookiemunch.net/consent.js",
  "api": "https://api.cookiemunch.net",
  "cbid": "acme-com-9f3k",
  "blockingMode": "auto"
}
```

POST /v1/sites/:cbid/brand

"Match my site": fetches the site's homepage and extracts theme tokens (colors, font) for pre-filling a banner design. Best-effort — returns an empty suggestion object if the homepage can't be fetched.

```
curl -X POST https://api.cookiemunch.net/v1/sites/acme-com-9f3k/brand \
  -H "Authorization: Bearer fck_…"
```

```
{
  "suggestion": {
    "background": "#0b0f1a",
    "text": "#f5f5f5",
    "highlight": "#5b8def",
    "fontFamily": "Inter, sans-serif",
    "palette": ["#0b0f1a", "#5b8def", "#f5f5f5"]
  }
}
```

501 if brand extraction isn't configured on this deployment.

Next: site config to actually configure the banner for one of these sites.

Site config

Site config

A site's SiteConfig controls everything about how the banner behaves for that property: theme, copy, categories, blocking mode, geo rules, IAB framework, consent persistence policy, and more. This is the same object consent.js fetches at runtime from GET /config/:cbid (the public, unauthenticated embed endpoint) — the /v1 routes below are the authenticated read/write path for managing it programmatically.

Requires the sites scope: sites:read to read, sites:write to save.

GET /v1/sites/:cbid/config

Returns the current config, or the default config if the site has never been configured (this endpoint never 404s for a config that doesn't exist yet — only for an unknown/foreign cbid).

```
curl https://api.cookiemunch.net/v1/sites/acme-com-9f3k/config \
  -H "Authorization: Bearer fck_…"
```

```
{
  "cbid": "acme-com-9f3k",
  "ver": 1,
  "banner": {
    "type": "multilevel",
    "layout": "bottom",
    "theme": { "background": "#0b0f1a", "text": "#f5f5f5", "highlight":
"#5b8def" }
  },
  "categories": {},
  "blocking": { "mode": "auto" },
  "i18n": { "defaultCulture": "en", "autoDetect": true },
  "framework": "none",
  "consentMode": { "enabled": true, "mode": "advanced", "waitForUpdate": 500 },
  "defaultMode": "opt-in",
  "bots": { "suppress": true },
  "consent": { "version": 1 }
}
```

PUT /v1/sites/:cbid/config

Upserts the config. Send a full object or a partial patch — omitted top-level keys are left untouched (deep-merged over the existing config).

```
curl -X PUT https://api.cookiemunch.net/v1/sites/acme-com-9f3k/config \
  -H "Authorization: Bearer fck_…" \
  -H "Content-Type: application/json" \
  -d '{
    "blocking": { "mode": "auto", "ignoreSelectors": [".fc-first-party"] },
    "defaultMode": "opt-in",
    "framework": "iab"
  }'
```

```
{ "cbid": "acme-com-9f3k", "ver": 1, "framework": "iab", "blocking": { "mode":
"auto", "ignoreSelectors": [".fc-first-party"] }, "...": "..." }
```

Key fields

Field	Type	Description
banner	object	type, layout, theme, content — or a v2 flow document from Banner Studio.

categories

object

Per-category label/description overrides for the four standard categories.

blocking

object

{ mode: "auto" | "manual", ignoreSelectors?: string[] } — see script blocking & data-attributes.

i18n

object

{ defaultCulture, autoDetect } — default language and whether to auto-detect from the visitor's browser.

framework

string

"none" or "iab" (enables TCF v2.2).

consentMode

object

Google Consent Mode v2 wiring: { enabled, mode: "basic" | "advanced", waitForUpdate }.

geoRules

array

Region !’ banner-mode rules (first match wins).

defaultMode

string

Banner mode when no geoRule matches: "opt-in", "opt-out", or "off".

experiment

object

A/B test config: { enabled, splitB, variantB }.

consent

object

{ version } — bump to force re-consent from everyone.

Errors

400 with structured issues when the config fails v2 flow validation:

```
{
  "error": "invalid_config",
  "issues": [
    { "code": "unknown_start_view", "message": "flow.start references a view id
that doesn't exist", "path": "flow.start" }
  ]
}
```

Note: Plan gating is enforced here too, not just in the dashboard UI. If your org is on the Free or Starter plan and you attempt to set framework: "iab", hide the "Powered by" branding (whiteLabel), or enable an experiment, the server silently strips those fields before saving rather than erroring — so a scoped automation key can never bypass plan limits, but it also won't tell you it happened. Check the response body against what you sent if a premium field doesn't seem to take effect.

Next: cookies, scan & verify to populate the cookie declaration this config's cookieTable element (if any) renders, or consent log, stats & export to read the decisions this config produces.

Banner library

Banner library

The banner library is where your reusable banner designs live at the organization level. Instead of configuring a banner per-domain, you design it once and assign it to as many properties as you like — publishing pushes the update to all of them at once.

Designs vs. sites

A design is a saved BannerConfigV2 — theme, categories, and the full screen flow — identified by a name, independent of any single property. A property can have zero or one design assigned to it at a time. If a property has no assigned design, Banner Studio falls back to that property's own stored config (or the polished default banner for brand-new properties).

Switching and creating designs

At the top of Banner Studio, the banner switcher dropdown lists every saved design in the organization. Selecting one loads it into the editor. Choose '• New banner to open a dialog, name your design, and click Create — this starts from the default template and saves immediately.

> Note: If the currently active property has no design assigned yet, creating a new design from that empty state automatically assigns it to that property, so it's remembered on reload.

Click the rename icon next to the switcher to rename the active design in place — the name updates immediately, no separate save step.

Assigning a design to properties

With a design loaded, click Active on these sites (n) to open the assignment popover — a checklist of every property in the organization. Check or uncheck a property to assign or unassign the current design; each toggle writes straight to the server immediately (there's no separate "save assignments" step), and the change reverts automatically if the request fails.

A single design can be assigned to any number of properties — that's the point: build one on-brand banner and roll it out to your entire fleet in one place.

Publishing

Publish in the Studio toolbar sends the current draft live to every property this design is currently assigned to, in one action. There's no per-property publish step — assignment is what determines the blast radius of a publish.

> Note: Reassigning a design to a new property doesn't retroactively publish to it — click Publish again after changing assignments so the new property picks up the current draft.

When to use one design vs. many

Use one shared design for a fleet of similar sites (same brand, same legal posture) — you get consistency and one place to update copy or categories. Create separate designs when properties need materially different flows: e.g. an EU-facing opt-in flow (GDPR Opt-In template) versus a US CCPA "Do Not Sell" modal for a different property.

Next steps

- Banner Studio — the editor itself.
- Templates & presets — starting points for a new design.
- Properties — the domains a design can be assigned to.

Templates & presets

Templates & presets

Templates are ready-made starting points for a new banner library design — a full theme, category set, and screen flow you can load in one click and then customize in Banner Studio.

Using a template

Click Templates in the Studio toolbar to open the picker, then choose one and it loads straight into the canvas, replacing the current draft. From there you can rename elements, restyle, or add screens exactly as if you'd built it from scratch.

> Note: Loading a template overwrites your current unsaved draft. Save or publish anything you want to keep first.

Built-in templates

Template	Screens	Best for
Default	Bar + preferences modal	The recommended starting point for most sites — Accept all / Reject all / Customize on a bottom bar.
Slim Bar	Single compact bar	Minimal footprint: text, Reject, Accept all, and a "Cookie settings" link, all in one row.
Classic 2-Tier	Bar + preferences modal	A traditional layout with per-category toggles (Preferences, Statistics, Marketing) in the modal.
3-Tier Progressive	Bar !’ options popup !’ detailed preferences modal	Progressive disclosure — quick accept/reject up front, a "Detailed settings" screen for visitors who want it.
GDPR Opt-In	Bar + preferences modal	EU-posture: every non-essential category defaults off until the visitor explicitly opts in.
CCPA Do Not Sell	Single modal	A standalone "Do Not Sell or Share My Personal Information" toggle + confirm, for US/California opt-out flows.

Each template ships with its own accent color and corner radius so it doesn't look like a generic default — expect to still apply your brand via theming & appearance.

Standard categories

Most templates use the three standard categories: Preferences, Statistics, and Marketing (Necessary/essential cookies are implicit and always on — they're not a toggle). The GDPR and CCPA templates instead define custom categories with their own default state (e.g. default: false

for GDPR opt-in) and a mapsTo mapping back to a standard category for reporting.

Custom categories

Beyond the three standard ones, you can define your own categories from the Settings drawer's Categories tab in Banner Studio — set an ID, label, description, and default state, and optionally map it to a standard category. This is how the built-in GDPR and CCPA templates express their non-standard toggles, and you can do the same for things like "Functional" or region-specific labels.

Choosing a template for your compliance posture

- Serving EU visitors under GDPR !’ start from GDPR Opt-In (opt-in by default) or add a geo rule in Settings !’ Targeting so EU traffic gets opt-in while other regions get a lighter default.
- Serving California/US visitors under CCPA !’ start from CCPA Do Not Sell, or add it as an additional screen in a broader flow.
- Everyone else, or a first banner !’ Default or Classic 2-Tier.

Next steps

- Banner Studio — screens, elements, and the settings drawer.
- Theming & appearance — make a template match your brand.
- Banner library — save your customized template as a reusable design.

Consent log, stats & export

Consent log, stats & export

Every decision a visitor makes — accept, decline, or a granular save — is written to a tamper-evident, hash-chained consent log per site. These endpoints read that log in three shapes (aggregated stats, raw rows, CSV audit export), plus signed per-decision receipts and subject-level erasure for GDPR/CCPA requests.

Requires the consent scope for /consent/* sub-routes: consent:read for all GETs below. erase-consent, subject-export, and receipt are exceptions — see the note at the end of this page.

GET /v1/sites/:cbid/consent/stats

Aggregated per-day totals — the fastest way to build a dashboard chart.

Query param	Type	Description
from	integer	Epoch-ms lower bound (inclusive). Defaults to 0.
to	integer	Epoch-ms upper bound (inclusive). Defaults to "now".

```
curl "https://api.cookiemunch.net/v1/sites/acme-com-9f3k/consent/stats?
from=1719792000000&to=1719878400000" \
  -H "Authorization: Bearer fck_…"
```

```
[
  {
    "Date": "2024-07-01",
    "OptIn": 842,
    "OptOut": 219,
    "OptInImplied": 0,
    "OptInStrict": 842,
    "TypeOptInPref": 601,
    "TypeOptInStat": 780,
    "TypeOptInMark": 512,
    "Impressions": 1150,
    "Countries": { "US": 640, "DE": 201, "GB": 220 }
  }
]
```

GET /v1/sites/:cbid/consent/log

Recent anonymized consent records, most recent first.

Query param	Type	Description
from / to	integer	Epoch-ms range, same as stats.
limit	integer	Max rows to return. Default 200, capped at 1000.

```
curl "https://api.cookiemunch.net/v1/sites/acme-com-9f3k/consent/log?limit=50" \
  -H "Authorization: Bearer fck_…"
```

```
[
  {
    "stamp": "a1b2c3d4e5f6",
    "receivedAt": 1719858421000,
    "region": "de",
    "method": "explicit",
    "choices": { "preferences": true, "statistics": true, "marketing": false },
    "anonIp": "203.0.113.0",
    "url": "https://acme.com/pricing"
  }
]
```

stamp is the visitor's consent-receipt id — the same value CookieMunch.consentId() returns client-side (see JavaScript API), and the key you pass to the receipt/erasure/export-by-subject endpoints below.

GET /v1/sites/:cbid/consent/export

The full audit trail as CSV — the artifact regulators ask for.

```
curl "https://api.cookiemunch.net/v1/sites/acme-com-9f3k/consent/export?
from=1719792000000" \
  -H "Authorization: Bearer fck_…" \
  -o consent-log.csv
```

```
Consent ID,Date/Time (UTC),Anonymized IP,User
Agent,URL,Region,Necessary,Preferences,Statistics,Marketing,Method,Version
a1b2c3d4e5f6,2024-07-01T18:07:01.000Z,203.0.113.0,Mozilla/5.0...,https://
acme.com/pricing,de,true,true,true,false,explicit,1
```

> Note: export additionally requires the site to be domain-verified (verified: true — see sites) and returns 403 with { "error": "domain not verified" } otherwise. stats and log have no such requirement.

GET /v1/sites/:cbid/receipt/:stamp

A signed, ISO/IEC-27560-style consent receipt for one decision — portable proof of what was consented to, when, and under what policy version. Requires the site to be verified.

```
curl "https://api.cookiemunch.net/v1/sites/acme-com-9f3k/receipt/a1b2c3d4e5f6?
format=json" \
  -H "Authorization: Bearer fck_…"
```

```
{
  "receipt": {
    "version": "1.0",
    "collector": "acme-com-9f3k",
    "consentTimestamp": "2024-07-01T18:07:01.000Z",
    "pii": false,
    "proofId": "a1b2c3d4e5f6",
    "region": "de",
    "purposes": [
      { "category": "necessary", "purpose": "Strictly necessary cookies and
storage", "consentType": "explicit" },
      { "category": "statistics", "purpose": "Statistics and analytics
measurement", "consentType": "explicit" }
    ]
  },
  "signature": "sha256:9f2a8b1e..."
}
```

The format query parameter also accepts html (a printable document) and pdf (a downloadable file, also negotiable via Accept: application/pdf) — useful for handing a visitor a document rather than raw JSON.

Subject-level erasure & export

Two endpoints operate on all records for one stamp, for handling a GDPR/CCPA data subject request without going through the DSAR workflow manually:

```
# Export a subject's own records (portability/access)
curl "https://api.cookiemunch.net/v1/sites/acme-com-9f3k/subject-export?
stamp=a1b2c3d4e5f6" \
  -H "Authorization: Bearer fck_…"
{ "cbid": "acme-com-9f3k", "stamp": "a1b2c3d4e5f6", "records": [ /* ... */ ],
"count": 3 }
# Crypto-erase a subject's records (irreversible)
curl -X POST https://api.cookiemunch.net/v1/sites/acme-com-9f3k/erase-consent \
  -H "Authorization: Bearer fck_…" \
  -H "Content-Type: application/json" \
  -d '{ "stamp": "a1b2c3d4e5f6" }'
{ "erased": 3, "encryptionEnabled": true }
```

> Note: Unlike stats/log/export/receipt, these two endpoints — erase-consent and subject-export — are checked against the sites scope (sites:write and sites:read respectively), not consent. This is because their path doesn't match the /consent/... sub-route the scope check pattern-matches on. A key scoped to consent:read/consent:write alone cannot call them — grant sites scope too if your integration needs subject-level erasure/export.

Next: cookies, scan & verify, or back to site config.

Cookies, scan & verify

Cookies, scan & verify

Three related capabilities live here: reading a site's categorized cookie declaration, triggering the crawler that builds it, and proving domain ownership. All three are checked against the sites scope (sites:read for the GETs, sites:write for the POSTs) — none of them fall under the consent scope, since they aren't part of the /consent/... sub-route.

GET /v1/sites/:cbid/cookies

The latest categorized cookie declaration — the same data a cookieTable element in the banner renders, and what visitors see if you publish a public cookie-policy page.

```
curl https://api.cookiemunch.net/v1/sites/acme-com-9f3k/cookies \
  -H "Authorization: Bearer fck_…"
```

```
{
  "updatedAt": 1719858421000,
  "cookies": [
    {
      "name": "_ga",
      "domain": ".acme.com",
      "category": "statistics",
      "purpose": "Registers a unique ID for statistical purposes.",
      "provider": "Google Analytics",
      "expiry": "2 years"
    },
    {
      "name": "session_id",
      "category": "necessary"
    }
  ]
}
```

category is one of necessary, preferences, statistics, marketing, or unclassified (seen but not yet matched to a known tracker). updatedAt is 0 if no scan has ever run and nothing has been observed live yet. 501 if cookie declarations aren't configured on this deployment.

> Note: This same data also accumulates passively — the embed reports cookie names (never values) and external script hosts it observes on real visits (see embed scripts) and merges them into this snapshot. Running an explicit scan (below) is the fastest way to seed it before any real traffic arrives.

POST /v1/sites/:cbid/scan

Kicks off an async crawl of the site to (re)build its cookie declaration.

```
curl -X POST https://api.cookiemunch.net/v1/sites/acme-com-9f3k/scan \
  -H "Authorization: Bearer fck_…"
```

```
{ "status": "scanning" }
```

202 Accepted — the crawl runs in the background; poll GET on the same path for status. 409 if a scan is already running for this site:

```
{ "status": "already-scanning" }
```

GET /v1/sites/:cbid/scan

```
curl https://api.cookiemunch.net/v1/sites/acme-com-9f3k/scan \
  -H "Authorization: Bearer fck_…"
```

```
{ "status": "idle", "lastScannedAt": 1719858421000 }
```

Field

Values

Description

status	idle | scanning
	Whether a crawl is currently running.
lastScannedAt	epoch-ms or null
	When the last completed scan finished.

Both scan routes return 501 if cookie scanning isn't configured on this deployment (the crawler is an optional component).

POST /v1/sites/:cbid/verify

Proves you own the site's domain, unlocking domain-gated features (consent export, signed receipts — see consent log, stats & export). Three challenge methods are supported; the verifyToken to place is on the site resource (GET /v1/sites/:cbid).

method	How it's checked
dns	A TXT record containing verifyToken at the domain.
meta	A <meta name="cookiemunch-verify" content="verifyToken"> tag on the homepage.
file	A file at /.well-known/cookiemunch-verify.txt containing verifyToken, served from the domain.

```
curl -X POST https://api.cookiemunch.net/v1/sites/acme-com-9f3k/verify \
  -H "Authorization: Bearer fck_…" \
  -H "Content-Type: application/json" \
  -d '{ "method": "dns" }'
```

```
{ "verified": true, "method": "dns" }
```

On failure, verified is false and the server does not update the site's status:

```
{ "verified": false, "reason": "TXT record not found or did not match" }
```

400 if method isn't one of dns/meta/file. 501 if domain verification isn't configured on this deployment.

Next: back to sites, or site config to tune what the crawler and blocker treat as first-party via blocking.ignoreSelectors.

Theming & appearance

Theming & appearance

Every banner design carries a set of theme tokens — colors, typography, and shape — plus optional custom CSS and a mobile layout breakpoint. These live in the Settings drawer's Appearance tab in Banner Studio.

Theme tokens

Token	Controls
background	Banner surface color.
text	Primary text color.
mutedText	Secondary text (descriptions, "powered by").
highlight	Accent color — the primary button and focal elements.
highlightText	Text color on the primary (highlight) button. Defaults to white.
buttonBg / buttonText / buttonBorder	Secondary button styling. Default to background / text / shade.
shade	Borders and subtle dividers.
link	Link color. Defaults to highlight.
overlay	Backdrop behind modal/popup screens. Defaults to a semi-transparent black.
scheme	Light or dark color scheme hint.
fontFamily	CSS font-family stack applied to the banner.
fontUrl	An external stylesheet (e.g. Google Fonts) to @import — must be https.
buttonRadius	Corner radius on buttons.
radius	Corner radius on the banner surface itself.
maxWidth	Maximum width of bar/modal surfaces on desktop.

Edit colors directly on elements via the Inspector, or set global tokens from the Appearance tab; per-element overrides on the canvas take precedence over the theme defaults.

Match my site

Click Match my site in the Appearance tab to fetch a brand suggestion for the active property — background, text, and accent colors plus font family/URL, inferred from the live site — and apply it to the theme in one click. This requires an active property to be selected; it's a starting point, not a pixel-perfect brand import, so review the result before publishing.

Brand kits

Save a theme + custom CSS combination as a brand kit so you can reapply it to other banner designs without redoing the work — useful when you manage several distinct brands or sub-brands within one organization. From the Appearance tab:

- Save as brand kit — name the current theme/CSS and store it.
- Apply a saved kit to instantly overwrite the current draft's theme and custom CSS.
- Delete kits you no longer need.

Custom CSS

The Custom CSS field in the Appearance tab accepts raw CSS scoped to the banner, for anything the theme tokens don't cover — custom animations, unusual spacing, or overriding a specific element's style. It applies on top of the theme tokens and any per-element styling.

Mobile breakpoint

The Layout section's breakpoint slider (360–1024px) sets the viewport width at which the banner switches from its desktop layout to its mobile layout. Since every element carries independent desktop and mobile position/size (set via the Desktop/Mobile toggle in the Studio toolbar), this slider only controls when the switch happens — design each breakpoint's layout separately on the canvas.

White-label (Pro)

On Pro and Scale plans, the Remove Cookie Munch branding toggle in the Appearance tab hides the "powered by" attribution from the banner. On lower tiers the toggle is shown disabled with a badge — see plans & billing.

Next steps

- Banner Studio — where these settings live, and how the canvas works.
- Templates & presets — pre-themed starting points.

Banner library

Banner library

The banner library is an account-level store of reusable banner designs. A design is a self-contained json payload (a v2 { v: 2, flow, categories, ... } banner config, or any future shape) that isn't tied to a single site — you build it once, assign it to any number of sites, and publish it to push the design live everywhere it's assigned.

All routes below live under /v1/banners, require an API key (see authentication), and are scoped to the calling key's org — a key can never read or write another org's designs. Keys with a scoped grant need banners:read for GET requests and banners:write for everything else (see API keys, members, webhooks & usage for how scopes work).

> Note: If the server hasn't been configured with a banner-library store, every route in this section returns 501 { "error": "banner library not configured" }. This can't happen against the hosted Cookie Munch API — it only applies to self-hosted deployments that omitted the dependency.

List designs

`GET /v1/banners`

Returns lightweight summaries — no json payload — so listing stays cheap even with a large library.

```
curl -H "Authorization: Bearer $COOKIEMUNCH_API_KEY" \
  https://api.cookiemunch.net/v1/banners
```

```
[
  {
    "id": "bnr_8f2c1a",
    "name": "EU cookie banner — dark",
    "updatedAt": 1730332800000,
    "assignedCbids": ["site_abc123", "site_def456"]
  }
]
```

Create a design

`POST /v1/banners`

Field	Type	Required	Description
name	string	yes	Display name for the design.
json	object	yes	A v2 banner config ({ v: 2, flow, categories, ... }). Rejected with 400 if it fails validation.

```
curl -X POST https://api.cookiemunch.net/v1/banners \
  -H "Authorization: Bearer $COOKIEMUNCH_API_KEY" \
  -H "Content-Type: application/json" \
  -d '{
    "name": "EU cookie banner — dark",
    "json": { "v": 2, "flow": { "views": [] }, "categories": [] }
  }'
```

Response 201:

```
{
  "id": "bnr_8f2c1a",
  "orgId": "org_9f01",
  "name": "EU cookie banner — dark",
  "json": { "v": 2, "flow": { "views": [] }, "categories": [] },
  "createdAt": 1730332800000,
  "updatedAt": 1730332800000
}
```

400 { "error": "invalid_config", "issues": [...] } when json fails the same lint/validation gate the Studio editor enforces.

Get / update / delete a design

```
GET    /v1/banners/{id}
PUT    /v1/banners/{id}
DELETE /v1/banners/{id}
```

PUT accepts a partial patch — { name?, json? } — and re-validates json when supplied.

```
curl -X PUT https://api.cookiemunch.net/v1/banners/bnr_8f2c1a \
  -H "Authorization: Bearer $COOKIEMUNCH_API_KEY" \
  -H "Content-Type: application/json" \
  -d '{ "name": "EU cookie banner — dark v2" }'
```

DELETE fails while the design is still assigned to any site:

```
{ "error": "banner still assigned to 2 site(s)", "code": "banner_in_use",
"cbids": ["site_abc123", "site_def456"] }
```

That response is a 409. Unassign every site first (see below), then delete.

Assignments

```
GET /v1/banners/{id}/assignments
PUT /v1/banners/{id}/assignments
```

GET returns the cbids currently pointed at this design. PUT replaces the full assignment set — any cbid you remove reverts to being unassigned from this design (it does not touch that site's already-published config), and any cbid you add is moved off whichever other design it previously belonged to.

Field	Type	Required	Description
cbids	string[]	yes	The complete list of site ids this design should be assigned to.

```
curl -X PUT https://api.cookiemunch.net/v1/banners/bnr_8f2c1a/assignments \
  -H "Authorization: Bearer $COOKIEMUNCH_API_KEY" \
  -H "Content-Type: application/json" \
  -d '{ "cbids": ["site_abc123", "site_def456"] }'
```

```
{ "cbids": ["site_abc123", "site_def456"] }
```

422 { "error": "cbid_not_in_org", "code": "cbid_not_in_org" } if any cbid doesn't belong to the calling key's org.

Publish

```
POST /v1/banners/{id}/publish
```

Compiles the design and merges its overlay keys into the live SiteConfig of every site currently assigned to it — this is what actually pushes the design to visitors. Sites keep their own config for everything the design doesn't own (e.g. geo rules, blocking mode); only the banner-presentation

keys are overlaid.

```
curl -X POST https://api.cookiemunch.net/v1/banners/bnr_8f2c1a/publish \
  -H "Authorization: Bearer $COOKIEMUNCH_API_KEY"
```

```
{ "publishedCbids": ["site_abc123", "site_def456"] }
```

400 { "error": "invalid_config", "issues": [...] } if the design fails validation at publish time (e.g. it was edited into an invalid state).

> Note: Publishing one banner across sites is the fastest way to keep a multi-brand or multi-property fleet visually consistent — build once in Studio, then call this endpoint (or the SDK's client.banners.publish(id)) from your deploy pipeline whenever the design changes.

SDK equivalent

```
await client.banners.list();
await client.banners.create({ name, json });
await client.banners.get(id);
await client.banners.update(id, { name, json });
await client.banners.delete(id);
await client.banners.assignments(id);
await client.banners.setAssignments(id, cbids);
await client.banners.publish(id);
```

See the full client reference for signatures.

Prior blocking

Prior blocking

"Prior blocking" means holding tracking scripts (analytics, ads, embeds) inert until a visitor consents — a GDPR/ePrivacy requirement in most jurisdictions, since cookies can't legally be set before consent for non-essential categories. Cookie Munch supports two modes, configured per-property from Banner Studio !’ Settings !’ Behaviour.

Auto mode (default)

data-blockingmode="auto" on your install script tag — the default — makes Cookie Munch intercept scripts and other tracking resources as they're inserted into the page, without you having to touch your existing markup. As soon as the script runs (which is why early placement in <head> matters), it watches for elements being added to the DOM, neutralizes anything that matches a known-tracker checklist so it can't load, and tags it for re-activation. Once the visitor consents to the matching category, tagged elements are re-enabled and load normally.

Auto mode is the fastest way to get compliant blocking on an existing site with no code changes. It's matched against a maintained list of known trackers, so brand-new or highly custom tracking snippets may occasionally be missed — always spot-check with your browser's network tab after installing.

Ignore selectors

Some elements should never be touched by the auto-blocker (a live chat widget, a first-party script you've already vetted). In the Behaviour tab, add CSS selectors to Ignore selectors (comma-separated, e.g. #chat, .no-block) to exclude them from interception entirely.

> Note: If you find yourself excluding more than a handful of elements, switch that site to manual mode instead — heavy use of ignore selectors can change script execution order in ways auto mode isn't designed to handle.

Manual mode

data-blockingmode="manual" turns off automatic interception and instead relies on markup you add yourself, giving you exact, explicit control over what's blocked. See the full data attributes reference for syntax; in short:

Scripts — set type="text/plain" (so the browser won't execute it) and add data-cookieconsent naming the category:

```
<script type="text/plain" data-cookieconsent="statistics,marketing"
        src="https://example.com/tracker.js"></script>
```

Valid category values are preferences, statistics, marketing, or ignore (combinable with commas, except ignore). On consent, Cookie Munch flips matching scripts to type="text/javascript" and re-injects them so the browser executes them.

Non-script resources (iframes, images) — since they can't use type, rename src to data-cookieblock-src:

```
<iframe data-cookieblock-src="https://www.youtube.com/embed/xxxx"
        data-cookieconsent="marketing" frameborder="0" allowfullscreen></iframe>
```

Choosing a mode

Auto	
Manual	
Setup effort	
None — works on existing markup	

	Requires editing every tracking snippet
Precision	
	Checklist-based, may miss unusual scripts
	Exact — you decide what's blocked
Best for	
	Fast rollout, general sites
	Sites with custom/unusual tracking, or teams wanting full control

Both modes can coexist across your organization — set the mode per property in that property's install script tag and Banner Studio settings; there's no requirement that every property use the same mode.

Re-ask and expiry

Also on the Behaviour tab, the re-ask slider controls how long a visitor's consent choice is remembered (0–730 days) before Cookie Munch shows the banner again — set to "never" to keep a choice indefinitely. Suppress for bots (on by default) skips showing the banner to detected crawlers so they aren't blocked from indexing your site.

Next steps

- Data attributes reference — full manual-mode markup syntax.
- Banner Studio — where blocking mode is configured.
- Installing the script — the data-blockingmode tag attribute.

Cookie scanner

Cookie scanner

The cookie scanner crawls your site with a real headless browser (Playwright) and records every cookie it observes, so your cookie declaration table stays accurate without you maintaining it by hand.

How it works

- The scanner loads your site's homepage (https://yourdomain.com/) in a headless Chromium instance.
- Every cookie set during that visit is captured and run through the categorization engine, then saved as a snapshot for your property.
- The declaration table and the public /api/v1/:cbid/cookies feed always serve the latest snapshot.

Note: Scanning only sees cookies that are set on the pages it visits. Cookies that only fire after a specific user interaction (e.g. clicking a video embed) may need to be added manually or picked up by live observation (below).

Categorization

Each cookie is matched against a curated provider database in three tiers, in order of confidence:

Tier	Match type	Confidence
1	Exact cookie name (e.g. _ga)	exact
2	Wildcard name prefix (e.g. _ga_*)	pattern
3	Provider-domain heuristic	pattern
—	No match	unclassified (unknown)

Each match resolves a category (Necessary, Preferences, Statistics, Marketing), a provider, a purpose description, and an expiry. Cookies that don't match anything land in the Unclassified category, flagged so you can review and reclassify them.

Running a scan

From Dashboard !' Cookies, click Scan now. The scan runs asynchronously:

- The button shows a scanning state while a crawl is in flight for that property.
- A second scan request for the same site while one is already running returns "already scanning" rather than starting a duplicate crawl.
- Once complete, the table refreshes automatically and a Scanned on badge shows the last-crawl date.

Note: Rescanning is safe to run as often as you like — each crawl fully replaces the previous

snapshot for that property.

Live observation

Independent of scheduled/manual crawls, the embed script can report cookie names (never values) it sees on real visitor page loads back to the API. These observations are merged into the latest snapshot — known classifications are preserved, and newly seen cookie names are categorized and added. This catches cookies that only appear under real traffic patterns the crawler didn't trigger.

Automatic rescanning

Operators can enable a scheduled daily crawl across all verified properties, plus a weekly diff report that flags newly appeared or disappeared cookies since the last report — useful for catching a marketing team quietly adding a new ad pixel.

Needs review

The page header shows a Needs review badge whenever any cookies are still Unclassified. Search and filter the table by category or provider to find them, then correct the category from the cookie declaration view.

See also: Cookie declaration table · Vendors & ROPA for tracking the vendors behind these cookies.

DSAR, preferences, ROPA & vendors

DSAR, preferences, ROPA & vendors

The privacy/governance surface covers everything a DPO needs outside of consent logging itself: Data Subject Access Requests (DSAR), a preference-center record store, Records of Processing Activities (RoPA), and third-party vendor risk tracking. Every route requires an API key (see authentication) and is scoped to the calling key's org.

Resource	Read scope	Write scope
/v1/dsar	dsar:read	dsar:write
/v1/vendors	vendors:read	vendors:write
/v1/ropa	ropa:read	ropa:write
/v1/preferences	consent:read	consent:write

Note: legacy keys with no scope list (scopes: null) have full access to every resource; scoping is opt-in. See API keys, members, webhooks & usage.

DSAR — Data Subject Access Requests

List all DSARs

`GET /v1/dsar`

Returns every DSAR in the org, across all statuses.

```
curl -H "Authorization: Bearer $COOKIEMUNCH_API_KEY" \
  https://api.cookiemunch.net/v1/dsar
```

```
[
  {
    "id": "dsar_7c1b",
    "type": "access",
    "subjectEmail": "jane@example.com",
    "regulation": "gdpr",
    "status": "received",
    "createdAt": 1730332800000,
    "dueAt": 1732924800000
  }
]
```

Create a DSAR

`POST /v1/dsar`

Field	Type	Required

Description			
type	access | deletion | rectification | portability | opt-out	yes	The right being exercised.
subjectEmail	string	yes	The data subject's email.
regulation	gdpr | ccpa	yes	Governing regulation (sets the statutory due date).
note	string	no	Free-text note.

```
curl -X POST https://api.cookiemunch.net/v1/dsar \
  -H "Authorization: Bearer $COOKIEMUNCH_API_KEY" \
  -H "Content-Type: application/json" \
  -d '{ "type": "access", "subjectEmail": "jane@example.com", "regulation":
"gdpr" }'
```

Response 201: { "request": { "id": "dsar_7c1b", "type": "access", "subjectEmail": "jane@example.com", "regulation": "gdpr", "status": "received", "createdAt": ..., "dueAt": ... } }.

Creating a DSAR fires a dsar.created webhook.

Advance a DSAR

`POST /v1/dsar/{id}/advance`

Field	Type	Required	Description
toStatus	received | verifying | in_progress | completed | rejected	yes	Target status.

Statuses move forward through a fixed pipeline; skipping or reversing an illegal transition returns 409.

```
curl -X POST https://api.cookiemunch.net/v1/dsar/dsar_7c1b/advance \
  -H "Authorization: Bearer $COOKIEMUNCH_API_KEY" \
  -H "Content-Type: application/json" \
  -d '{ "toStatus": "verifying" }'
```

409 { "error": "illegal transition: received -> completed" } if you try to skip a step. Advancing fires a dsar.updated webhook.

SDK equivalent

```
await client.dsar.list();
await client.dsar.create({ type: 'access', subjectEmail: 'jane@example.com',
regulation: 'gdpr' });
await client.dsar.advance('dsar_7c1b', 'verifying');
```

Preference center

A lightweight key-value store of per-subject consent preferences, separate from the cookie-consent log — useful for marketing opt-ins (newsletter, SMS, etc.) tracked outside the banner.

List preference records

```
GET /v1/preferences
```

```
curl -H "Authorization: Bearer $COOKIEMUNCH_API_KEY" \
  https://api.cookiemunch.net/v1/preferences
```

```
[
  {
    "subjectId": "jane@example.com",
    "purposes": { "newsletter": true, "sms": false },
    "method": "single-opt-in",
    "confirmed": false,
    "updatedAt": 1719858421000,
    "version": 1
  }
]
```

501 { "error": "preference center not configured" } on a self-host that hasn't wired a preference store.

Save a preference record

```
POST /v1/preferences
```

Field	Type	Required	Description
subjectId	string	yes	End-user identifier (email, internal id, etc.).
purposes	object	yes	Map of purpose name !’ boolean.

```
curl -X POST https://api.cookiemunch.net/v1/preferences \
  -H "Authorization: Bearer $COOKIEMUNCH_API_KEY" \
  -H "Content-Type: application/json" \
  -d '{ "subjectId": "jane@example.com", "purposes": { "newsletter": true,
"sms": false } }'
```

```
{
  "record": {
    "subjectId": "jane@example.com",
    "purposes": { "newsletter": true, "sms": false },
    "method": "single-opt-in",
    "confirmed": false,
    "updatedAt": 1719858421000,
    "version": 1
  }
}
```

SDK equivalent

```
await client.preferences.list();
await client.preferences.save('jane@example.com', { newsletter: true, sms:
false });
```

Vendors (processors)

Registering a vendor computes a risk score/band from the fields you supply — useful for a vendor-risk register or DPA tracker.

List vendors

```
GET /v1/vendors
```

```
curl -H "Authorization: Bearer $COOKIEMUNCH_API_KEY" \
  https://api.cookiemunch.net/v1/vendors
```

```
[
  {
    "id": "vnd_1",
    "name": "Segment",
    "category": "analytics",
    "dataShared": ["device_id", "email_hash"],
    "dpaSigned": true,
    "subprocessors": 3,
    "certifications": ["SOC2"],
    "region": "US",
    "risk": { "score": 34, "band": "medium" }
  }
]
```

Create a vendor

```
POST /v1/vendors
```

Field	Type	Required	Description
name	string	yes	Vendor name.
category	string	yes	e.g. analytics, advertising.
dataShared	string[]	yes	Categories of data shared with the vendor.
dpaSigned	boolean	yes	Whether a Data Processing Agreement is signed.
subprocessors	number	yes	Count of the vendor's own subprocessors.
certifications	string[]		

yes

e.g. ["SOC2", "ISO27001"].

region

string

yes

Primary processing region, e.g. EU.

```
curl -X POST https://api.cookiemunch.net/v1/vendors \
  -H "Authorization: Bearer $COOKIEMUNCH_API_KEY" \
  -H "Content-Type: application/json" \
  -d '{
    "name": "Segment", "category": "analytics", "dataShared": ["device_id"],
    "dpaSigned": true, "subprocessors": 3, "certifications": ["SOC2"],
"region": "US"
  }'
```

Response 201: { "vendor": { ... }, "risk": { "score": 34, "band": "medium" } }.

SDK equivalent

```
await client.vendors.list();
await client.vendors.create({ name, category, dataShared, dpaSigned,
subprocessors, certifications, region });
```

Records of Processing Activities (RoPA)

List RoPA entries

```
GET /v1/ropa
```

```
curl -H "Authorization: Bearer $COOKIEMUNCH_API_KEY" \
  https://api.cookiemunch.net/v1/ropa
```

Create a RoPA entry

```
POST /v1/ropa
```

Field

Type

Required

Description

name

string

yes

Activity name.

purpose

string

yes

Purpose of processing.

legalBasis

consent | contract | legal-obligation | vital-interests | public-task | legitimate-interests

yes

GDPR Art. 6 legal basis.

dataCategories

string[]

yes

Categories of personal data processed.

recipients

string[]

yes

Who receives the data.

retentionDays

number

yes

Retention period, in days.

crossBorderTransfer

boolean

yes

Whether data crosses borders.

```
curl -X POST https://api.cookiemunch.net/v1/ropa \
  -H "Authorization: Bearer $COOKIEMUNCH_API_KEY" \
  -H "Content-Type: application/json" \
  -d '{
    "name": "Email marketing", "purpose": "Send product updates",
    "legalBasis": "consent", "dataCategories": ["email"],
    "recipients": ["Mailchimp"], "retentionDays": 730, "crossBorderTransfer":
true
  }'
```

Response 201: { "entry": { "id": "ropa_1", "name": "Email marketing", ... } }.

SDK equivalent

```
await client.ropa.list();
await client.ropa.create({ name, purpose, legalBasis, dataCategories,
recipients, retentionDays, crossBorderTransfer });
```

Related: consent logging, subject export/erasure, and signed receipts live under /v1/sites/{cbid}/consent/... — see consent & sites.

API keys, members, webhooks & usage

API keys, members, webhooks & usage

Org-administration endpoints: invite/manage members, issue/list API keys, subscribe to webhooks, and read usage. All require an API key (see authentication); the org is always derived from the key you authenticate with — no route in this section accepts an orgId parameter, so a key can only ever act on its own org.

> Note: these are admin-only surfaces. A scoped API key (one issued with an explicit scopes list) can never call /v1/members, /v1/keys, or /v1/webhooks — every request gets 403 { "error": "insufficient_scope", "message": "this key may not use admin endpoints; manage them in the dashboard" } regardless of which scopes it holds. Only a legacy, unscoped key (scopes: null — the default for keys issued before scoped keys existed, and for keys issued via POST /v1/keys itself) can reach this section. Manage scoped keys' permissions from the dashboard instead.

Members

List members

```
GET /v1/members
```

```
curl -H "Authorization: Bearer $COOKIEMUNCH_API_KEY" \
  https://api.cookiemunch.net/v1/members
```

```
[
  { "userId": "usr_1", "email": "owner@example.com", "role": "owner" },
  { "userId": "usr_2", "email": "teammate@example.com", "role": "admin" }
]
```

Invite a member

```
POST /v1/members
```

Field	Type	Required	Description
email	string	yes	Email of the person to invite.
role	admin | member | viewer	yes	Role to assign. owner cannot be granted via the API.

```
curl -X POST https://api.cookiemunch.net/v1/members \
  -H "Authorization: Bearer $COOKIEMUNCH_API_KEY" \
  -H "Content-Type: application/json" \
  -d '{ "email": "teammate@example.com", "role": "admin" }'
```

Response 201: { "member": { "userId": "usr_2", "email": "teammate@example.com", "role": "admin" } }. 409 if the email is already a member.

> Note: inviting a member grants durable human access — that person keeps access to the dashboard independently of the API key that invited them. Treat members:write-capable keys with the same care as an admin password.

Change a member's role

```
PATCH /v1/members/{userId}
```

Field	Type	Required	Description
role	admin | member | viewer	yes	New role.

```
curl -X PATCH https://api.cookiemunch.net/v1/members/usr_2 \
  -H "Authorization: Bearer $COOKIEMUNCH_API_KEY" \
  -H "Content-Type: application/json" \
  -d '{ "role": "viewer" }'
```

400 { "error": "the owner's role cannot be changed" } if userId is the org owner.

Remove a member

```
DELETE /v1/members/{userId}
```

```
curl -X DELETE https://api.cookiemunch.net/v1/members/usr_2 \
  -H "Authorization: Bearer $COOKIEMUNCH_API_KEY"
```

400 if userId is the owner, or if they're the org's last remaining member. Otherwise 200 { "ok": true }.

SDK equivalent

```
await client.members.list();
await client.members.invite('teammate@example.com', 'admin');
await client.members.setRole('usr_2', 'viewer');
await client.members.remove('usr_2');
```

API keys

List key prefixes

```
GET /v1/keys
```

Returns display metadata only — the secret is never retrievable after issuance.

```
curl -H "Authorization: Bearer $COOKIEMUNCH_API_KEY" \
  https://api.cookiemunch.net/v1/keys
```

```
[{ "prefix": "fck_a1b2c3d4", "createdAt": 1730332800000 }]
```

501 { "error": "key listing not configured" } on a self-host without key metadata storage wired.

Issue a new key

```
POST /v1/keys
```

Takes no body. The key is returned once — store it immediately; it can't be recovered later, only revoked and replaced.

```
curl -X POST https://api.cookiemunch.net/v1/keys \
  -H "Authorization: Bearer $COOKIEMUNCH_API_KEY"
```

```
{ "key": "fck_9e8d7c6b5a4f3e2d1c0b", "prefix": "fck_9e8d7c6b" }
```

> Note: keys issued this way are unscoped (full access), because this route is itself admin-only and reachable only by unscoped keys. Create scoped, least-privilege keys from the dashboard's API Keys settings page.

SDK equivalent

```
await client.keys.list();
const { key, prefix } = await client.keys.issue();
```

Usage

```
GET /v1/usage
```

Current resource usage / quota summary for the org (plan-relative).

```
curl -H "Authorization: Bearer $COOKIEMUNCH_API_KEY" \
  https://api.cookiemunch.net/v1/usage
```

```
{ "domains": 4, "seats": 3, "monthlyEvents": 128430 }
```

501 { "error": "usage reporting not configured" } on a self-host without usage tracking wired. Unlike members/keys/webhooks, usage needs no scope — any valid key (scoped or not) can read it.

SDK equivalent

```
const usage = await client.usage();
```

Webhooks

Full event catalogue, payload shape, and signature verification are covered in Webhooks & events. This section is the CRUD reference.

List subscriptions

```
GET /v1/webhooks
```

The signing secret is omitted from list responses.

```
curl -H "Authorization: Bearer $COOKIEMUNCH_API_KEY" \
  https://api.cookiemunch.net/v1/webhooks
```

```
[
  {
    "id": "whk_1",
    "orgId": "org_9f01",
    "url": "https://example.com/hooks/cookiemunch",
    "events": ["consent.recorded", "dsar.created"],
    "cbid": null,
    "active": true,
    "createdAt": 1730332800000
  }
]
```

Create a subscription

```
POST /v1/webhooks
```

Field	Type	Required	Description
url	string	yes	HTTPS endpoint to deliver events to.
events	string[]	yes	Non-empty list from the event catalogue.
cbid	string	no	Scope delivery to a single site. Omit for all sites in the org.

```
curl -X POST https://api.cookiemunch.net/v1/webhooks \
  -H "Authorization: Bearer $COOKIEMUNCH_API_KEY" \
  -H "Content-Type: application/json" \
  -d '{ "url": "https://example.com/hooks/cookiemunch", "events":
["consent.recorded", "dsar.created"] }'
```

Response 201 includes the signing secret once:

```
{
  "id": "whk_1",
  "orgId": "org_9f01",
  "url": "https://example.com/hooks/cookiemunch",
  "secret": "whsec_3f9a...",
  "events": ["consent.recorded", "dsar.created"],
  "cbid": null,
  "active": true,
  "createdAt": 1730332800000
}
```

Delete a subscription

```
DELETE /v1/webhooks/{id}
```

```
curl -X DELETE https://api.cookiemunch.net/v1/webhooks/whk_1 \
  -H "Authorization: Bearer $COOKIEMUNCH_API_KEY"
```

204 No Content on success.

SDK equivalent

```
await client.webhooks.list();
await client.webhooks.create({ url, events, cbid });
await client.webhooks.delete('whk_1');
```

Cookie declaration table

Cookie declaration table

The cookie declaration is the human-readable list of every cookie your site sets, its purpose, the provider that sets it, and how long it lives — the artifact most privacy regulations expect you to publish (GDPR Art. 13, and effectively required by most DPAs' cookie-banner guidance). Cookie Munch generates and keeps it current for you from scanner results.

Viewing the table

Dashboard !’ Cookies shows every cookie detected for the active property, grouped into a badge-colored category and sortable/filterable by:

Column

Description

Cookie

The exact cookie name

Category

Necessary / Preferences / Statistics / Marketing / Unclassified

Provider

The company or service that sets the cookie (e.g. Google Analytics)

Purpose

A plain-language description of what the cookie does

Expiry

How long the cookie persists (e.g. "2 years", "session")

Use the search box to find a cookie by name, domain, provider, or purpose, and the category/provider dropdowns to narrow the list. A Scanned on badge shows when the underlying snapshot was last refreshed.

Publishing it on your site

The declaration is also available as a drop-in embeddable widget — no dashboard access required for whoever maintains your privacy policy page:

```
<script src="https://YOUR-CMP-DOMAIN/cookie-declaration.js" data-cbid="YOUR_SITE_ID"></script>
```

Drop that tag anywhere in your privacy policy or cookie policy page. It:

- Finds its own <script> tag and reads data-cbid.
- Fetches the live /api/v1/:cbid/cookies feed for that site.
- Renders a self-contained HTML table, grouped by category, with a "Last updated" timestamp — no CSS or JS dependencies required from the host page.

Note: The embed always reflects the current snapshot. There's nothing to re-publish when cookies change — the widget updates itself on the visitor's next page load.

Keeping it accurate

The declaration is only as good as the underlying scan. If you add a new analytics/marketing script to your site:

- Run a manual scan (Cookies !’ Scan now), or wait for the next scheduled crawl.
- Review any cookies that land in Unclassified and confirm their category — unmatched cookies default to Unclassified until a human (or a future scanner-database update) classifies them.

See also: Cookie scanner · REST consent for the underlying /api/v1/:cbid/cookies feed.

Consent log

Consent log

Every consent decision your banner captures — accept, reject, or a granular save-preferences click — is written to a durable, hash-chained log entry. This is the system of record you'd point a regulator or auditor at to prove a specific visitor consented (or didn't) at a specific time.

What gets recorded

Each entry is stored per site (cbid) with:

Field	Description
stamp	The unique consent-receipt ID for this decision — the key you use to look up, export, or erase a subject's record
receivedAt	Server-side timestamp the decision was ingested
region	The visitor's detected region at decision time (e.g. de, us-ca)
method	explicit (visitor clicked something) or implied (passive/default)
choices	The granted booleans: preferences, statistics, marketing (necessary is always on)
anonIp	A truncated/anonymized form of the visitor's IP — never the raw address
url	The page URL the decision was made on

> Note: Necessary cookies never appear as a toggle-able choice — they're always granted, since the site cannot function without them, and are recorded in every receipt as an "implied" purpose.

Querying the log

The log is queryable per site over a time range and returns the most recent entries first:

`GET /api/v1/:cbid/consent/log?from=<ms>&to=<ms>&limit=<n>`

limit defaults to 200 and caps at 1000. Results are newest-first. This is the same data source that backs audit export & receipts and the DSAR erase/export tools, which look a subject up by their stamp.

Tamper-evidence

Records are hash-chained: each entry's hash incorporates the previous entry's hash, so altering or deleting a historical record breaks the chain from that point forward. You can verify chain integrity for a site at any time:

`GET /api/v1/:cbid/consent/verify`

which returns { "valid": true | false }. A false result means the stored chain has been tampered with or corrupted — treat it as an operational incident.

Finding a specific visitor's record

If a visitor gives you their consent receipt ID (the stamp, typically surfaced to them after they

interact with the banner), you can look up every record tied to it — this is the mechanism DSAR access/erasure requests use under the hood, and how the preference center resolves a subject.

> Note: Consent-log reads require authentication in multi-tenant deployments — the log itself is never publicly readable, unlike the cookie declaration feed.

See also: Audit export & receipts · REST consent

OpenAPI spec

OpenAPI spec

The entire /v1 developer API is described by a hand-written OpenAPI 3.1 document, served live and kept in lockstep with the actual route handlers — it's generated from the same server package, not maintained as a separate artifact that can drift.

```
GET /v1/openapi.json
```

This is the one unauthenticated route on the /v1 surface — no API key required, so tooling (API explorers, SDK generators, CI schema-diff checks) can fetch it freely.

```
curl https://api.cookiemunch.net/v1/openapi.json
```


```
{
  "openapi": "3.1.0",
  "info": {
    "title": "Cookie Munch Developer API",
    "version": "1.0.0",
    "description": "Stable, versioned developer API for Cookie Munch.
Authenticate every request with an API key (`Authorization: Bearer fck_…` or `X-
API-Key: fck_…`). The organization is derived from the key — never pass an
orgId."
  },
  "servers": [{ "url": "https://api.cookiemunch.net" }],
  "security": [{ "ApiKeyBearer": [] }, { "ApiKeyHeader": [] }],
  "components": { "securitySchemes": { "...": "..." }, "schemas": { "...":
"..." } },
  "paths": { "...": "..." }
}
```


Security schemes

Two equivalent schemes are declared — use whichever fits your HTTP client better:

Scheme	Type	How
ApiKeyBearer	http / bearer	Authorization: Bearer fck_<hex>
ApiKeyHeader	apiKey / header	X-API-Key: fck_<hex>

Every operation in the document lists both under security, except GET /v1/openapi.json itself (security: []).

What's covered

The document enumerates every /v1/* path with per-operation summaries, parameters, request bodies, and response schemas — sites, config, consent stats/log/export, receipts, DSAR, vendors, RoPA, cookies, scans, A/B results, brand kits, preferences, members, keys, usage, webhooks, and the banner library — plus a handful of public (no-auth) /api/v1/... routes used by the embed itself (cookie-declaration feed, client-observed cookie ingest, cross-device consent profile sync).

Generating a client

Point any OpenAPI-3.1-compatible generator at the live document to produce a typed client in a language other than TypeScript. For example, with openapi-typescript:

```
npx openapi-typescript https://api.cookiemunch.net/v1/openapi.json -o
```

```
cookiemunch.d.ts
```

Or with openapi-generator-cli for other languages:

```
npx @openapitools/openapi-generator-cli generate \
  -i https://api.cookiemunch.net/v1/openapi.json \
  -g python \
  -o ./cookiemunch-client
```

> Note: if you're on TypeScript/JavaScript, prefer the hand-written @cookiemunch/sdk — it ships richer types (enums, discriminated unions) than a generic OpenAPI codegen produces, and tracks the API in the same monorepo release.

Versioning

The document's info.version reflects the /v1 surface's semantic version. Because the prefix itself is /v1, breaking changes ship under a new prefix (/v2) rather than mutating this document's meaning out from under existing integrations — additive fields and new optional parameters can appear within /v1 without a version bump.

Install & initialize

Install & initialize

@cookiemunch/sdk is a small, fully-typed REST client over the /v1 developer API. It has zero required dependencies beyond a fetch implementation, works in Node, Deno, Bun, and the browser, and mirrors every route documented in the REST API section.

Install

```
npm install @cookiemunch/sdk
# or
pnpm add @cookiemunch/sdk
# or
yarn add @cookiemunch/sdk
```

Initialize

```
import { createCookieMunch } from '@cookiemunch/sdk';

const client = createCookieMunch({
  apiKey: process.env.COOKIEMUNCH_API_KEY!,
  baseUrl: 'https://api.cookiemunch.net', // the /v1 prefix is appended
automatically
});

const sites = await client.sites.list();
```

CookieMunchOptions

Field	Type	Required	Description
apiKey	string	yes	Your fck_… API key (see authentication for how to issue one).
baseUrl	string	yes	Your Cookie Munch server origin, e.g. https://api.cookiemunch.net. /v1 is appended internally — don't include it.
fetch	typeof fetch	no	Inject a custom fetch (for tests, non-standard runtimes, or request instrumentation). Defaults to the global fetch.

Note: the SDK never accepts or sends an orgId. Every request is authenticated by the API key, and the server derives the org from that key — this makes cross-org access structurally impossible from client code.

Error handling

Every non-2xx response throws CookieMunchApiError:

```
import { CookieMunchApiError } from '@cookiemunch/sdk';

try {
  await client.sites.get('unknown-cbid');
} catch (err) {
  if (err instanceof CookieMunchApiError) {
    console.error(err.status, err.message, err.code);
    // 404 "site not found" undefined
  }
  throw err;
}
```

Property	Type	Description
status	number	The HTTP status code.
message	string	The server's error field, or a generic request failed with status N if the body wasn't JSON.
code	string | undefined	The server's code field, when present (e.g. banner_in_use, cbid_not_in_org).

Injecting fetch (testing / non-browser runtimes)

```
import { createCookieMunch } from '@cookiemunch/sdk';
import { vi } from 'vitest';

const mockFetch = vi.fn(async () => new Response(JSON.stringify({ orgId:
'org_1', plan: 'free', keyPrefix: 'fck_test' }), { status: 200 }));

const client = createCookieMunch({ apiKey: 'fck_test', baseUrl: 'http://
localhost:8787', fetch: mockFetch as unknown as typeof fetch });

await client.me();
```

Next steps

- Client reference — every client.* method, with signatures and examples.
- MCP setup — expose the same API as tools an AI agent can call directly, built on this SDK.
- REST API: authentication — how API keys and scopes work under the hood.

Audit export & receipts

Audit export & receipts

Two related but distinct tools sit on top of the consent log: a bulk CSV export for compliance reporting, and per-decision signed receipts you can hand to an individual visitor (or their lawyer) as portable proof of what they consented to.

CSV export

Export the full consent history for a site, over a time range, as CSV:

`GET /api/v1/:cbid/export?from=<ms>&to=<ms>`

Returns Content-Type: text/csv with a consent-log.csv attachment — one row per consent record, ready to hand to a DPO, auditor, or regulator without any conversion step.

> Note: Both /export and /consent/log require the site to be verified (domain ownership confirmed) before they'll return data — this prevents a stolen cbid from being used to scrape another site's consent history.

Signed consent receipts

A receipt is a structured, ISO/IEC 27560-inspired proof-of-consent document for a single decision, built from its stamp:

`GET /api/v1/:cbid/receipt/:stamp`

The JSON receipt includes:

Field	Description
version	Receipt schema version (1.0)
collector	The site's cbid
consentTimestamp	ISO-8601 timestamp of the decision
proofId	The consent stamp this receipt was built from
region	Visitor region at decision time
purposes[]	Per-purpose breakdown — category, human-readable purpose text, and whether it was explicit or implied
policy	Optional linked privacy-policy URL + version, when configured

Every receipt is signed with HMAC-SHA256 over a canonical (stable key order) JSON serialization, so the pair { receipt, signature } can travel between systems and still be verified as untampered — no shared database access required by the verifying party.

Formats

The same receipt is available in three formats via content negotiation:

Request	Response

default

Signed JSON ({ receipt, signature })

?format=html

Self-contained HTML page

?format=pdf (or Accept: application/pdf)

Downloadable PDF, Content-Disposition: attachment

Note: Receipts require both a verified site domain and a configured signing secret on the server. If receipts aren't configured, the endpoint returns 501.

When to use which

- CSV export — bulk reporting, regulator requests covering many visitors, internal audits.
- Signed receipt — a single visitor asks "prove what I agreed to," or you need a portable, independently-verifiable artifact for one decision (e.g. attached to a support ticket or legal file).

See also: Consent log · DSAR requests · REST consent

Client reference

Client reference

Every method on the @cookiemunch/sdk client, grouped by namespace. All methods return Promises and throw CookieMunchApiError on non-2xx responses (see install & initialize). Each method maps 1:1 to a route documented in the REST API section — follow the links for full request/response details.

```
import { createCookieMunch } from '@cookiemunch/sdk';
const client = createCookieMunch({ apiKey: 'fck_…', baseUrl: 'https://
api.cookiemunch.net' });
```

client.me()

```
me(): Promise<Identity>
```

Identity for the API key: { orgId, plan, keyPrefix }.

```
const { orgId, plan } = await client.me();
```

client.sites

Maps to /v1/sites/....

```
sites: {
  list(): Promise<Site[]>;
  create(input: SiteCreate): Promise<Site>;
  get(cbid: string): Promise<Site>;
  delete(cbid: string): Promise<void>;
  getConfig(cbid: string): Promise<SiteConfig>;
  putConfig(cbid: string, config: SiteConfig): Promise<SiteConfig>;
  cookies(cbid: string): Promise<SiteCookie[]>;
  scan(cbid: string): Promise<ScanResult>;
  scanStatus(cbid: string): Promise<ScanResult>;
  ab(cbid: string): Promise<AbResult[]>;
  snippet(cbid: string, opts?: SnippetOptions): Promise<InstallSnippet>;
  verify(cbid: string, method: 'dns' | 'meta' | 'file'): Promise<VerifyResult>;
  brand(cbid: string): Promise<BrandExtractionResult>;
}
const site = await client.sites.create({ domain: 'example.com' });
const config = await client.sites.getConfig(site.cbid);
await client.sites.putConfig(site.cbid, { ...config, banner:
{ ...config.banner, layout: 'popup' } });

const { snippet } = await client.sites.snippet(site.cbid, { blockingMode:
'auto', culture: 'en' });
// paste `snippet` into <head>

await client.sites.scan(site.cbid);                 // kick off a crawl
const status = await client.sites.scanStatus(site.cbid); // { status:
'scanning' | 'idle', lastScannedAt }

const { verified } = await client.sites.verify(site.cbid, 'dns');
const { suggestion } = await client.sites.brand(site.cbid); // "Match my site"
theme extraction
```

client.consent

Maps to /v1/sites/{cbid}/consent/..., /v1/sites/{cbid}/receipt/{stamp}, /v1/sites/{cbid}/erase-consent, /v1/sites/{cbid}/subject-export — see consent & sites.

```
consent: {
  stats(cbid: string, query?: RangeQuery): Promise<ConsentDay[]>;
  log(cbid: string, query?: LogQuery): Promise<ConsentLogRow[]>;
  export(cbid: string, query?: RangeQuery): Promise<string>; // raw CSV text
  receipt(cbid: string, stamp: string): Promise<SignedReceipt>;
  eraseSubject(cbid: string, stamp: string): Promise<{ erased: number }>;
```

```
  exportSubject(cbid: string, stamp: string): Promise<{ cbid: string; stamp:
string; records: unknown[]; count: number }>;
}
const days = await client.consent.stats(cbid, { from: Date.now() - 30 *
86_400_000 });
const recent = await client.consent.log(cbid, { limit: 50 });
const csv = await client.consent.export(cbid); // requires a verified domain

const receipt = await client.consent.receipt(cbid, 'stamp_abc123'); // requires
a verified domain

// GDPR/CCPA erasure and portability, keyed by the consent-receipt stamp:
await client.consent.eraseSubject(cbid, 'stamp_abc123');   // irreversible
const bundle = await client.consent.exportSubject(cbid, 'stamp_abc123');
```

client.dsar

Maps to /v1/dsar.

```
dsar: {
  list(): Promise<DsarRequest[]>;
  create(input: DsarCreate): Promise<{ request: DsarRequest }>;
  advance(id: string, toStatus: DsarStatus): Promise<{ request: DsarRequest }>;
}
const { request } = await client.dsar.create({ type: 'access', subjectEmail:
'jane@example.com', regulation: 'gdpr' });
await client.dsar.advance(request.id, 'verifying');
```

client.vendors

Maps to /v1/vendors.

```
vendors: {
  list(): Promise<ScoredVendor[]>;
  create(input: VendorInput): Promise<{ vendor: Record<string, unknown>; risk:
{ score: number; band: string } }>;
}
const { vendor, risk } = await client.vendors.create({
  name: 'Segment', category: 'analytics', dataShared: ['device_id'],
  dpaSigned: true, subprocessors: 3, certifications: ['SOC2'], region: 'US',
});
```

client.ropa

Maps to /v1/ropa.

```
ropa: {
  list(): Promise<RopaEntry[]>;
  create(input: RopaInput): Promise<{ entry: RopaEntry }>;
}
const { entry } = await client.ropa.create({
  name: 'Email marketing', purpose: 'Send product updates', legalBasis:
'consent',
  dataCategories: ['email'], recipients: ['Mailchimp'], retentionDays: 730,
crossBorderTransfer: true,
});
```

client.brandKits

Maps to /v1/brand-kits.

```
brandKits: {
  list(): Promise<BrandKit[]>;
  create(input: BrandKitCreate): Promise<{ kit: BrandKit }>;
  delete(id: string): Promise<void>;
}
const { kit } = await client.brandKits.create({ name: 'House style', theme:
{ primary: '#1a1a1a' } });
await client.brandKits.delete(kit.id);
```

client.preferences

Maps to /v1/preferences.

```
preferences: {
  list(): Promise<PreferenceItem[]>;
  save(subjectId: string, purposes: Record<string, boolean>): Promise<unknown>;
}
await client.preferences.save('jane@example.com', { newsletter: true, sms:
false });
```

client.members

Maps to /v1/members.

```
members: {
  list(): Promise<Member[]>;
  invite(email: string, role: string): Promise<{ member: { userId: string;
email: string; role: string } }>;
  setRole(userId: string, role: string): Promise<{ member: { userId: string;
email: string; role: string } }>;
  remove(userId: string): Promise<unknown>;
}
await client.members.invite('teammate@example.com', 'admin');
await client.members.setRole('usr_2', 'viewer');
await client.members.remove('usr_2');
```

> Note: members, keys, and webhooks are admin-only — only an unscoped API key can call these. See API keys, members, webhooks & usage.

client.keys

Maps to /v1/keys.

```
keys: {
  list(): Promise<ApiKey[]>;
  issue(input?: ApiKeyIssueInput): Promise<ApiKey>;
}
const { key, prefix } = await client.keys.issue(); // `key` is shown once —
store it now
```

client.usage()

```
usage(): Promise<Usage>
```

Maps to GET /v1/usage.

```
const { domains, seats, monthlyEvents } = await client.usage();
```

client.webhooks

Maps to /v1/webhooks. Event catalogue and signature verification: Webhooks & events.

```
webhooks: {
  list(): Promise<WebhookSubscription[]>;
  create(input: WebhookCreate): Promise<WebhookSubscription>;
  delete(id: string): Promise<void>;
}
const sub = await client.webhooks.create({
  url: 'https://example.com/hooks/cookiemunch',
  events: ['consent.recorded', 'dsar.created'],
});
console.log(sub.secret); // store this now — it's never returned again
await client.webhooks.delete(sub.id);
```

client.banners

Maps to banner library.

```
banners: {
  list(): Promise<BannerSummary[]>;
  create(input: { name: string; json: SiteConfig }): Promise<BannerRecord>;
```

```
  get(id: string): Promise<BannerRecord>;
  update(id: string, patch: { name?: string; json?: SiteConfig }):
Promise<BannerRecord>;
  delete(id: string): Promise<void>;
  assignments(id: string): Promise<{ cbids: string[] }>;
  setAssignments(id: string, cbids: string[]): Promise<{ cbids: string[] }>;
  publish(id: string): Promise<{ publishedCbids: string[] }>;
}
const design = await client.banners.create({ name: 'EU banner — dark', json:
{ v: 2, flow: { views: [] }, categories: [] } });
await client.banners.setAssignments(design.id, [site.cbid]);
const { publishedCbids } = await client.banners.publish(design.id);
```

Full example: onboard a site end to end

```
import { createCookieMunch } from '@cookiemunch/sdk';

const client = createCookieMunch({ apiKey: process.env.COOKIEMUNCH_API_KEY!,
baseUrl: 'https://api.cookiemunch.net' });

const site = await client.sites.create({ domain: 'shop.example.com' });
await client.sites.verify(site.cbid, 'dns');

const { snippet } = await client.sites.snippet(site.cbid);
// hand `snippet` to whoever manages the site's <head>

await client.webhooks.create({ url: 'https://example.com/hooks/cookiemunch',
events: ['consent.recorded'], cbid: site.cbid });
```

See also: MCP tools reference — the same capabilities exposed as tools for an AI agent, built directly on this SDK.

Analytics & A/B testing

Analytics & A/B testing

Dashboard !’ Analytics turns your raw consent events into the metrics you actually care about: how many visitors opt in, which categories they grant, and where they're coming from — plus an A/B testing tool to improve opt-in rate.

KPIs

At the top of the page:

KPI

What it means

Consents

Total consent decisions in the selected range

Opt-in rate

Share of decisions that granted at least one non-necessary category

Impressions

How many times the banner was shown

Ratio

Consents ÷ impressions — how many banner views convert into a decision

Each KPI shows a delta against the equivalent prior period (e.g. the previous 30 days), so you can spot a regression (a copy change that tanked opt-in rate) quickly.

Use the range selector (7d / 30d / 90d / All) to change the window everything on the page reflects.

Charts

- Opt-in trend — daily opt-in rate over time.
- Consent volume — daily opt-in vs. opt-out counts, stacked.
- Implied vs. strict — donut of decisions made passively (implied) vs. via explicit interaction (strict/explicit).
- Per-category opt-in — trend lines for Preferences, Statistics, and Marketing opt-in rates individually, so you can see e.g. that visitors are happy to accept Statistics but reject Marketing.
- Top countries — bar chart + table of consent volume by country. Click a country (in either the chart or the table) to filter every other chart on the page down to that country; click it again to clear the filter.
- Funnel — impressions !’ opt-ins !’ opt-outs, showing drop-off at each stage.

Note: All charts respect the active country filter and the selected date range simultaneously — drilling into "DE" and switching to "7d" narrows every chart at once.

Multi-property

If your org has more than one site, use the site switcher in the page header — analytics are always scoped to one property at a time to keep the numbers meaningful.

A/B testing

Under Banner settings !’ Targeting, enable A/B testing to split visitors between your current banner (variant A) and a challenger (variant B) at a configurable split percentage. Results (impressions and opt-ins per variant) feed back into the same consent-stats pipeline, so you can judge whether a design or copy change actually moves opt-in rate before rolling it out to everyone.

Note: A/B testing is a paid ("Scale" plan) capability — see Banner settings for where it lives in

the editor.

See also: Consent log · Banner settings

DSAR requests

DSAR requests

A Data Subject Access Request (DSAR) is any formal ask from a visitor to access, delete, correct, or export the personal data you hold on them. Dashboard !’ Privacy !’ DSAR gives you an intake queue, a status workflow, and the tools to actually fulfill the request against your consent records.

Logging a request

Click New request and fill in:

Field	Options
Request type	Access, Deletion, Rectification, Portability, Opt-out
Regulation	GDPR, CCPA
Subject	Any identifier the requester gave you — email, account ID, phone number, etc.

The due date is set automatically from the regulation's statutory deadline: 30 days for GDPR (Art. 12(3)), 45 days for CCPA.

Status workflow

Requests move through a fixed set of statuses, and only in the allowed direction:

Status	Can advance to
received	verifying, rejected
verifying	in_progress, rejected
in_progress	completed, rejected
completed	— (terminal)
rejected	— (terminal)

Use the Advance dropdown on each row to move a request forward — Cookie Munch only offers the statuses that are legal next steps from where it currently sits.

> Note: A request past its due date and not yet in a terminal status is flagged Overdue in red — the header badge shows your total overdue count at a glance.

Fulfilling access/deletion requests

Once you've verified the requester's identity, two actions become available on rows where you have their consent receipt stamp (from the consent log):

- Export data — downloads a JSON file of every consent record tied to that stamp for the given site, satisfying an access/portability request. Nothing is deleted.
- Erase data — permanently crypto-erases the subject's consent records (their user agent and

page URL) for that site. The tamper-evident hash chain stays intact — only the personal fields are unrecoverable — and this cannot be undone.

Both require the site's cbid and the subject's stamp. If crypto-erasure isn't configured (no encryption key set on the server), the erase action tells you so instead of silently no-oping.

> Note: Crypto-erasure only removes what's encrypted per-subject (user agent, URL). It does not remove the subject from your own CRM, order history, or any other system — DSAR fulfillment for those systems is your responsibility outside Cookie Munch.

See also: Consent log · Preference center · Audit export & receipts

Setup

Setup

@cookiemunch/mcp is a Model Context Protocol server that exposes the Cookie Munch Developer API as tools an AI agent (Claude Desktop, Claude Code, or any other MCP client) can call directly — list sites, read consent stats, edit a banner flow, open a DSAR, and more, all from a chat session. It's built on @cookiemunch/sdk, so every tool maps to a documented /v1 route.

It talks stdio (the transport MCP clients use to spawn local servers), not HTTP — the client process launches it as a child process and speaks MCP over its stdin/stdout.

Get an API key

You need an API key first — see authentication for how to issue one, or call POST /v1/keys / client.keys.issue().

Run it

```
npx -y @cookiemunch/mcp
```

npx resolves the package's single published binary (cookiemunch-mcp, at packages/mcp/src/bin.ts) automatically. Two environment variables control it:

Variable	Required	Default	Description
COOKIEMUNCH_API_KEY	yes	—	Your fck_… API key. The process exits immediately with an error if unset.
COOKIEMUNCH_BASE_URL	no	http://localhost:8787	Your Cookie Munch server origin (the /v1 prefix is appended automatically, same as the SDK). Set this to your real API origin, e.g. https://api.cookiemunch.net.

```
COOKIEMUNCH_API_KEY=fck_9e8d7c6b5a4f3e2d1c0b \
COOKIEMUNCH_BASE_URL=https://api.cookiemunch.net \
npx -y @cookiemunch/mcp
```

Configure an MCP client

Claude Desktop / Claude Code

Add an entry to your MCP client's config (claude_desktop_config.json, or .mcp.json for Claude Code):

```
{
  "mcpServers": {
    "cookiemunch": {
      "command": "npx",
      "args": ["-y", "@cookiemunch/mcp"],
      "env": {
        "COOKIEMUNCH_API_KEY": "fck_9e8d7c6b5a4f3e2d1c0b",
        "COOKIEMUNCH_BASE_URL": "https://api.cookiemunch.net"
      }
    }
  }
```

```
}
```

Restart the client, then ask it something like "list my Cookie Munch sites" or "open a GDPR access DSAR for jane@example.com" — it will call the matching tool automatically.

Any other MCP client

Any client that supports stdio MCP servers works the same way: point it at npx -y @cookiemunch/mcp (or a local cookiemunch-mcp binary if you installed the package globally/locally) with the two environment variables set.

```
npm install -g @cookiemunch/mcp
cookiemunch-mcp
```

Programmatic use

You can also build and connect the server yourself — for example to inject a pre-built SDK client (tests, custom auth) instead of API-key/base-URL:

```
import { createCookieMunchMcp } from '@cookiemunch/mcp';
import { StdioServerTransport } from '@modelcontextprotocol/sdk/server/
stdio.js';

const server = createCookieMunchMcp({ apiKey: process.env.COOKIEMUNCH_API_KEY!,
baseUrl: 'https://api.cookiemunch.net' });
await server.connect(new StdioServerTransport());
```

> Note: the MCP server has exactly the same authority as the API key you give it — every write it can perform (editing a site's config, inviting a member, issuing a new key) is scoped by that key's org and, if the key is scoped, by its scope list. Give an agent a scoped key limited to what it actually needs to do, not an unscoped admin key, especially in an autonomous workflow.

Next steps

- Tools reference — every tool the server registers, with its inputs.
- Client reference — the underlying SDK surface every tool calls into.

Tools reference

Tools reference

Every tool registered by @cookiemunch/mcp (packages/mcp/src/tools.ts), grouped the way the server registers them. Each tool wraps one @cookiemunch/sdk call (or a short sequence of them) and returns its JSON result as the tool's text content; SDK errors are caught and returned as an isError result rather than crashing the MCP session. See Setup for how to connect a client.

> Note: every tool acts on the org tied to the configured API key — no tool accepts an orgId. Admin tools (invite_member, set_member_role, remove_member, issue_api_key) require an unscoped key, exactly like the REST routes they call — see API keys, members, webhooks & usage.

Sites & config

Tool	What it does	Inputs
whoami	Return the identity tied to the API key: orgId, plan, key prefix.	—
list_sites	List all sites (cbids) in your organization.	—
create_site	Create a new site. The cbid is auto-generated if omitted.	domain (string), cbid? (string)
delete_site	Delete a site and its configuration from your org.	cbid
get_site_config	Get the banner/consent configuration for a site.	cbid
update_site_config	Upsert (merge) the banner/consent configuration for a site.	cbid, config (partial SiteConfig patch)
verify_site	Verify domain control via dns / meta / file challenge. Required to unlock consent export and signed receipts.	cbid, method (dns | meta | file)
match_site_brand	Extract theme colors/typography from a site's homepage to suggest a matching banner theme.	cbid
get_install_snippet	Get the exact <script> tag to install Cookie Munch on a site.	cbid, blockingMode? (auto|manual|checklist), culture?

Consent & privacy

Tool	What it does	Inputs
get_consent_stats	Aggregated per-day consent statistics for a site.	cbid, from?, to? (epoch-ms)
get_consent_log	Recent anonymised consent records for a site.	cbid, from?, to?, limit? (default 200)
export_consent	Export a site's consent log as raw CSV text.	cbid, from?, to?
get_receipt	Fetch the signed consent receipt for one consent record, by stamp.	cbid, stamp
erase_subject_data	Irreversibly crypto-erase a data subject's consent records by receipt stamp (GDPR/CCPA deletion).	cbid, stamp
export_subject_data	Export a data subject's consent records (GDPR access/portability).	cbid, stamp
save_preference	Save/update an end-user's consent preferences (purpose !’ boolean map).	subjectId, purposes (record of boolean)
list_preferences	List configurable consent preferences/purposes for your org.	—

DSAR

Tool	What it does	Inputs
list_dsar	List all Data Subject Access Requests for your organization.	—
create_dsar	Open a new DSAR.	type (access|deletion|rectification|portability|opt-out), subjectEmail, regulation (gdpr|ccpa), note?
advance_dsar	Advance a DSAR to a new status.	id, toStatus (received|verifying|in_progress|completed|rejected)

Governance (vendors & RoPA)

Tool	What it does	Inputs
list_vendors	List vendors (processors) with their computed risk scores.	—
create_vendor	Register a vendor and compute its risk score.	name, category, dataShared[], dpaSigned, subprocessors, certifications[], region
list_ropa	List Records of Processing Activities for your organization.	—
create_ropa	Create a RoPA entry.	name, purpose, legalBasis, dataCategories[], recipients[], retentionDays, crossBorderTransfer

Cookies & scanning

Tool	What it does	Inputs
get_site_cookies	List the cookies discovered on a site, with their consent categories.	cbid
scan_site	Trigger a cookie scan for a site. Returns the scan job state.	cbid
get_scan_status	Get the status/result of a site's most recent cookie scan.	cbid
get_ab_results	Get A/B banner experiment results (per variant) for a site.	cbid

v2 banner flow editing

Tool	What it does	Inputs
get_flow	Get the v2 banner flow config for a site (views, categories, lint issues).	cbid
edit_flow	Apply a batch of structured edit ops (addView, removeView, addElement, setButtonTransition, addCustomCategory) to a site's flow. Validated and lint-clean before	

PUT; rejects empty op lists.

cbid, operations[] (each with an op field)

set_flow

Wholesale-replace a site's v2 flow with a complete config (e.g. from a template). Validates before PUT.

cbid, config (full v2 FlowConfig)

Structured v1 config (non-flow domains)

Each of these fetches the current config, patches only its own slice, and PUTs — safe to call without clobbering unrelated settings.

Tool

What it does

Inputs

set_blocking

Configure script-blocking mode.

cbid, mode (auto|manual), ignoreSelectors?[]

set_geo_rules

Set geo-targeting rules (banner mode per country/region).

cbid, geoRules[] ({ match: { countries?, regions? }, mode }), defaultMode?

set_languages

Configure i18n: default culture, auto-detect, per-locale copy overrides.

cbid, defaultCulture?, autoDetect?, translations? (per-locale record)

set_consent_mode

Configure Google Consent Mode signals.

cbid, enabled, mode (basic|advanced), waitForUpdate? (ms)

set_ab_experiment

Configure the A/B banner experiment.

cbid, enabled, splitB (0–100), variantB? (partial banner override)

set_consent_policy

Configure consent persistence (re-prompt policy).

cbid, expiryDays?, version?

set_banner_basics

Configure v1 banner appearance: type, layout, theme, copy.

cbid, type?, layout?, theme?, content?

Org, members & keys

Tool

What it does

Inputs

list_members

List the members of your organization.

—

invite_member

Invite a person to your org by email. owner is not assignable via API key.

email, role (admin|member|viewer)

set_member_role
Change a member's role. The owner's role cannot be changed.
userId, role
remove_member
Remove a member. The owner cannot be removed.
userId
list_api_keys
List API keys for your org (secrets are never returned).
—
issue_api_key
Issue a new API key. The secret is returned once.
—
get_usage
Get the current usage/quota summary for your organization.
—

Webhooks

Tool
What it does
Inputs
list_webhooks
List webhook subscriptions for your organization.
—
create_webhook
Create a webhook subscription. The response includes the signing secret (shown once).
url, events[], cbid?
delete_webhook
Delete a webhook subscription by id.
id

Brand kits

Tool
What it does
Inputs
list_brand_kits
List reusable brand kits (colors/logo/typography) for your org.
—
create_brand_kit
Create a reusable brand kit.
name, theme, content?, logoUrl?, customCss?
delete_brand_kit
Delete a brand kit by id (must belong to your org).
id

Banner library

Tool	What it does	Inputs
list_banners	List reusable banner designs in your org.	—
create_banner	Create a reusable banner design.	name, json (a v2 BannerConfig)
get_banner	Get a banner design by id.	id
update_banner	Update a banner design's name and/or config.	id, name?, json?
delete_banner	Delete a banner design (fails if still assigned to sites).	id
assign_banner	Set which sites (cbids) use a banner design.	id, cbids[]
publish_banner	Publish a design live to all its assigned sites.	id

Example session

Once the server is connected, an agent can chain tools naturally:

```
User: Create a site for shop.example.com, install it, and open a GDPR
      access request for jane@example.com.

Agent calls: create_site({ domain: "shop.example.com" })
          -> get_install_snippet({ cbid: "site_9f01ab" })
          -> create_dsar({ type: "access", subjectEmail: "jane@example.com",
regulation: "gdpr" })
```

Every call above is a thin wrapper over the identically-named SDK method — if a tool's behavior is ambiguous, its underlying client.* call and REST route are the source of truth.

Preference center

Preference center

Beyond the cookie banner's necessary/preferences/statistics/marketing categories, most sites also collect standing marketing preferences — newsletter, SMS, profiling consent — that live outside the banner's lifecycle. Dashboard !’ Privacy !’ Preferences is where you look up and edit a specific subject's preferences on their behalf (e.g. when they email you asking to be unsubscribed from something the banner doesn't cover).

Looking up a subject

Enter any identifier the subject is known by — email address, account ID, or another reference you use internally — and click Look up.

- If a record exists, its current purposes load with their granted/denied state.
- If nothing is found, a fresh record is prepared with every known purpose defaulted to denied, and a No record on file badge shows so you know you're creating, not editing.

Purpose set

Out of the box, four purposes are recognized with friendly labels; any other purpose key you save is displayed as-is:

Purpose key	Label
email_marketing	Email marketing
newsletter	Newsletter
sms	SMS
profiling	Profiling

Each purpose renders as a switch. Toggle any combination, then Save preferences to persist the change.

> Note: This tool edits the record of preference, not the delivery systems themselves (your ESP, SMS provider, etc.) — pair it with your own automation if you need those toggles to actually suppress sends.

When to use this vs. the DSAR flow

- Use Preference center for routine, non-statutory "please update my marketing preferences" requests — it's a direct edit, no due-date tracking.
- Use DSAR requests when the ask is a formal, regulation-backed access/deletion/rectification/portability request that needs an auditable due date and status trail.

See also: DSAR requests · Consent log

Vendors & ROPA

Vendors & ROPA

Dashboard !’ Privacy !’ Governance covers two GDPR-driven recordkeeping obligations that sit behind your banner: a Record of Processing Activities (RoPA) — Art. 30's requirement to document what you process and why — and a vendor register with automatic risk scoring for every third party you share data with.

Record of Processing Activities (RoPA)

Click New record and document a processing activity:

Field	Description
Activity name	What the processing activity is called internally
Purpose	Why you're processing this data
Legal basis	consent, contract, legal-obligation, vital-interests, public-task, legitimate-interests
Retention (days)	How long the data is kept
Cross-border transfer	Whether this activity moves data outside your home region

The table shows retention in a friendly format (e.g. "1.5 years" once it crosses 365 days) and a Transfers out / In region badge per record, so a compliance review can scan the whole RoPA at a glance.

Vendor register

Click New vendor to add a third party your site shares data with:

Field	Description
Vendor name	e.g. "Google Analytics"
Category	e.g. "Analytics", "Advertising"
Region	Where the vendor processes data — see the region list below
DPA signed	Whether you have a Data Processing Agreement in place

Risk scoring

Every vendor gets an automatic risk score and band (Low / Medium / High) so you can prioritize which vendor relationships need attention first. Scoring weighs:

- Whether a DPA is on file (no DPA raises risk).
- Whether the vendor's region is GDPR-adequate.

GDPR-adequate regions (no extra transfer safeguard needed) are:

`EU, EEA, UK, US, Switzerland, Canada, Japan, New Zealand`

A vendor whose region falls outside that set is scored as a higher-risk cross-border transfer, since it typically requires Standard Contractual Clauses or another Art. 46 transfer mechanism to stay compliant.

> Note: The region dropdown in the vendor dialog matches this exact adequacy list plus an Other / non-adequate catch-all — pick "Other" for any vendor whose region isn't independently GDPR-adequate.

Why this matters

A regulator (or a customer's procurement/security review) asking "what do you process, under what legal basis, and who do you share it with" is one of the most common privacy audits. Keeping RoPA and the vendor register current means that question has a five-minute answer instead of a scramble.

See also: DSAR requests · Geo-targeting & regional rules

Webhooks & events

Webhooks & events

Webhooks push events to your own HTTPS endpoint as they happen, so you don't have to poll the API. Subscriptions are managed via /v1/webhooks or client.webhooks.*; this page covers the event catalogue, delivery payload, and how to verify signatures.

Event catalogue

Event	Fires when
consent.recorded	A visitor records a consent decision on a subscribed site.
scan.completed	A cookie scan finishes for a site.
scan.cookies_changed	A cookie scan finds a different cookie set than the last scan.
dsar.created	A new DSAR is opened (via the dashboard or POST /v1/dsar).
dsar.updated	A DSAR's status advances (via the dashboard or POST /v1/dsar/{id}/advance).
banner.published	A banner-library design is published to a site.

Subscribe

```
curl -X POST https://api.cookiemunch.net/v1/webhooks \
  -H "Authorization: Bearer $COOKIEMUNCH_API_KEY" \
  -H "Content-Type: application/json" \
  -d '{
    "url": "https://example.com/hooks/cookiemunch",
    "events": ["consent.recorded", "dsar.created", "dsar.updated"]
  }'
```

Pass cbid to scope delivery to a single site; omit it to receive the event for every site in the org. Full CRUD reference (list/create/delete, request and response shapes): API keys, members, webhooks & usage.

> Note: the create response includes the signing secret once — store it immediately. It's never returned by GET /v1/webhooks afterward.

Delivery payload

Every delivery is a POST to your url with this JSON body:

```
{
  "id": "evt_1b1e6f2a-...",
  "type": "consent.recorded",
  "cbid": "site_abc123",
  "createdAt": 1730332800000,
  "data": { "...": "event-specific payload" }
}
```

Field	Type	Description

id

string

Unique event id (evt_<uuid>), safe to use for idempotency dedup.

type

string

One of the event catalogue values above (or ping — see Testing).

cbid

string | null

The site the event is about, or null for org-level events.

createdAt

number

Epoch-ms when the event was emitted.

data

object

Event-specific payload (e.g. the DSAR record for dsar.created/dsar.updated).

Headers

Header

Description

Content-Type

Always application/json.

X-CookieMunch-Event

The event type, duplicated as a header for routing without a body parse.

X-CookieMunch-Signature

sha256=<hex hmac> — an HMAC-SHA256 of the raw request body, keyed by your subscription's secret.

Verifying signatures

Recompute the HMAC over the raw, unparsed request body using your subscription's secret, and compare it to the X-CookieMunch-Signature header with a constant-time comparison:

```
import { createHmac, timingSafeEqual } from 'node:crypto';
import express from 'express';

const app = express();

app.post('/hooks/cookiemunch', express.raw({ type: 'application/json' }), (req,
res) => {
  const signature = req.header('X-CookieMunch-Signature') ?? '';
  const expected = `sha256=${createHmac('sha256',
process.env.COOKIEMUNCH_WEBHOOK_SECRET!).update(req.body).digest('hex')}`;

  const sigBuf = Buffer.from(signature);
  const expBuf = Buffer.from(expected);
  if (sigBuf.length !== expBuf.length || !timingSafeEqual(sigBuf, expBuf)) {
    return res.status(401).send('invalid signature');
  }

  const event = JSON.parse(req.body.toString('utf8'));
  console.log('received', event.type, event.id);
  res.status(200).end();
});
```

Note: use the raw body bytes for the HMAC, not a re-serialized JSON.stringify of the parsed object — even a single whitespace difference will fail the signature check. That's why the

example above uses express.raw() instead of express.json() on this route.

Delivery & retries

- Delivery is fire-and-forget and best-effort — a failed delivery never blocks or fails the API call that triggered the event.
- A non-2xx response or network error is retried up to 3 attempts total, with linear backoff (roughly 500ms, then 1000ms between attempts).
- After the final attempt, a failed delivery is dropped — there is no dead-letter queue. Return 2xx quickly (verify the signature, enqueue the work, respond) rather than doing slow processing inline.
- Only active subscriptions receive events; a subscription's cbid filter (if set) must match the event's site, or it's skipped.

Testing a subscription

The dashboard's webhooks page can send a one-off test delivery to a subscription — a single attempt, no retries, delivered even if the subscription is currently paused, so you can verify your endpoint before flipping it live:

```
{
  "id": "evt_...",
  "type": "ping",
  "cbid": null,
  "createdAt": 1730332800000,
  "data": { "message": "This is a test delivery from Cookie Munch." }
}
```

It's signed exactly like a real event — verify it the same way; expect type: "ping" rather than one of the catalogue events.

SDK equivalent

```
const sub = await client.webhooks.create({
  url: 'https://example.com/hooks/cookiemunch',
  events: ['consent.recorded', 'dsar.created', 'dsar.updated'],
});
console.log(sub.secret); // store once

await client.webhooks.list();
await client.webhooks.delete(sub.id);
```

See also: banner library for the banner.published trigger, and DSAR, preferences, ROPA & vendors for the dsar.* triggers.

Google Consent Mode v2

Google Consent Mode v2

Consent Mode v2 lets Google tags (gtag.js, GTM, Google Ads, GA4) adjust their own behavior based on the visitor's consent state, instead of you having to block or unblock each tag manually. Cookie Munch implements it natively — no separate GTM template required.

Enable it

In the banner settings editor, go to Consent !’ Consent Mode and toggle Enable Consent Mode. Two fields become available:

Field	Options	Purpose
Mode	basic / advanced	See below
Wait for update	0–3000 ms (slider)	How long gtag holds tag firing while waiting for the consent signal

What gets signaled

Cookie Munch's three visitor-controlled categories map to the full Consent Mode v2 signal set:

Category	Signals
Marketing	ad_storage, ad_user_data, ad_personalization
Statistics	analytics_storage
Preferences	functionality_storage, personalization_storage
— (always granted)	security_storage

Before any vendor script loads, the banner pushes a default-denied block for every signal except security_storage. As soon as the visitor decides (or a stored decision is restored), an update signal reflects their actual choices — Google tags that respect Consent Mode adjust their behavior (e.g. cookieless pings instead of full tracking) automatically.

Basic vs. advanced mode

- Basic — pings are either fully blocked or fully allowed based on consent; no additional signals are set.
- Advanced — in addition to the signals above, the default block also sets ads_data_redaction: true and url_passthrough: true, which tell Google Ads to redact ad-click data and preserve click IDs via URL passthrough until consent is granted — enabling conversion modeling for visitors who haven't yet decided or who declined, instead of losing that data entirely.

Note: wait_for_update only delays tag firing — it does not change the default (denied) state. If the visitor hasn't decided within that window, tags fire with consent still denied, and update the

moment a decision arrives.

Interaction with other frameworks

Consent Mode signals are derived from the same three-category consent state that drives IAB TCF and US State Privacy — you don't configure them separately. If your site also runs TCF, granting purposes 2, 3, 4, or 7 (ad selection/delivery/measurement) sets marketing = true, which in turn grants the Consent Mode ad signals.

See also: Banner settings · Geo-targeting & regional rules

Integrations (GTM, WordPress…)

Integrations (GTM, WordPress…)

Every integration below installs the same canonical embed — a single <script id="CookieMunch"> tag. Pick the channel that matches your stack; you don't need more than one.

What you need

- Host — your Cookie Munch server origin, e.g. https://api.cookiemunch.net.
- Site ID (cbid) — from the dashboard, or GET /v1/sites/{cbid}/snippet / client.sites.snippet(cbid) (see sites & config), which returns the exact tag pre-filled for you.

The canonical embed

Place this as high in <head> as possible — ideally the very first script — so automatic prior-blocking can intercept trackers before they run:

```
<script id="CookieMunch"
        src="https://<host>/consent.js"
        data-cbid="<SITE_ID>"
        data-blockingmode="auto"></script>
```

Attribute	Required	Values	Meaning
id="CookieMunch"	yes	CookieMunch	Lets the engine find its own script tag to read config from.
src	yes	https://<host>/consent.js	The embed served by your Cookie Munch server.
data-cbid	yes	your site id	Identifies this site in your dashboard.
data-blockingmode	yes	auto | manual | checklist	auto blocks third-party scripts until consent; manual never blocks; checklist blocks per category.
data-culture	no	e.g. en, de	Optional banner language override.
data-api	no		

https://<api-host>

Only needed when consent.js is served from a CDN separate from the API (beacons + the TCF bundle go here instead).

The cookie-declaration table (for a Cookie Policy page) is a second, optional embed placed in the page body:

```
<script src="https://<host>/cookie-declaration.js" data-cbid="<SITE_ID>"></
script>
```

WordPress plugin

Location: plugins/wordpress/forgeconsent/ in this repo.

- Upload the folder to wp-content/plugins/ (or zip it and upload via Plugins !’ Add New !’ Upload Plugin), then activate it.
- Configure under Settings !’ Cookie Munch: API host / base URL, Site ID (cbid), blocking mode (auto/manual), culture (optional), and an enable/disable toggle.
- The plugin injects the embed into <head> at wp_head priority 1 — as early as possible, so prior-blocking works even on themes/plugins that enqueue analytics scripts aggressively.
- Render the cookie table anywhere with the [cookiemunch_cookie_declaration] shortcode.

Google Tag Manager

Location: integrations/gtm/. Two install paths — pick one:

Option 1 — Custom Template (recommended)

- In GTM, open your Web container !’ Templates !’ New (under Tag Templates).
- In the template editor's overflow menu ("î) !’ Import !’ select integrations/gtm/template.tpl from this repo !’ Save.
- Tags !’ New !’ Tag Configuration !’ choose Cookie Munch (under "Custom"). Fill in: API host/base URL, Site ID (cbid), blocking mode, optional culture.
- Triggering !’ Consent Initialization - All Pages (fall back to Initialization - All Pages if unavailable) — this fires before every other trigger, so the auto-blocker installs first.
- Save, then Submit / Publish the container.

Option 2 — Custom HTML tag (fallback)

- Tags !’ New !’ Tag Configuration !’ Custom HTML, paste the canonical embed with <host> / <SITE_ID> filled in.
- Leave "Support document.write" unchecked.
- Triggering !’ Consent Initialization - All Pages (or Initialization - All Pages).
- Save and Submit / Publish.

Note: the Custom Template passes cbid, blockingmode, and culture both via the consent.js query string and a window.CookieMunchConfig global, since GTM's sandboxed injectScript API can't set arbitrary data-* attributes. The Custom HTML fallback uses the literal data-* attributes and is byte-for-byte identical to the standard install — use it when you need exact parity.

Copy-paste snippets

Location: integrations/snippets/. Per-platform, paste-where guides for site builders that don't have a dedicated plugin:

Platform

Guide

Shopify

integrations/snippets/shopify.md — paste into theme.liquid, immediately after <head>, before any analytics/pixel/app scripts.

Wix

integrations/snippets/wix.md — Settings !’ Custom Code, added to the <head> of all pages.

Webflow

integrations/snippets/webflow.md — Site Settings !’ Custom Code !’ Head Code (site-wide).

Generic HTML

integrations/snippets/generic-html.md — for any hand-built or templated site.

Each guide uses the same canonical embed and the same cookie-declaration embed for a Cookie Policy page.

Shopify example

```
<!-- theme.liquid, immediately after <head> -->
<script id="CookieMunch"
        src="https://<host>/consent.js"
        data-cbid="<SITE_ID>"
        data-blockingmode="auto"></script>
```

If your theme has multiple layout files (theme.liquid, checkout.liquid), add the snippet to each one you want the banner on. Full checkout customization requires Shopify Plus.

Generating the snippet programmatically

Rather than hand-copying the template, fetch it pre-filled for a specific site — useful for a self-serve onboarding flow or a deploy script that injects the tag into a CMS:

```
curl -H "Authorization: Bearer $COOKIEMUNCH_API_KEY" \
  "https://api.cookiemunch.net/v1/sites/YOUR_CBID/snippet?
blockingmode=auto&culture=en"
```

```
{
  "snippet": "<script id=\"CookieMunch\"\n  src=\"https://api.cookiemunch.net/
consent.js\"\n  data-cbid=\"YOUR_CBID\"\n  data-blockingmode=\"auto\"\n  data-
culture=\"en\"></script>",
  "src": "https://api.cookiemunch.net/consent.js",
  "api": "https://api.cookiemunch.net",
  "cbid": "YOUR_CBID",
  "blockingMode": "auto"
}
```

```
const { snippet } = await client.sites.snippet(cbid, { blockingMode: 'auto',
culture: 'en' });
```

See sites & config for the full endpoint reference, and the MCP get_install_snippet tool for the same capability from an agent.

Verify the install

After installing on any channel: open the site, open browser devtools, and confirm a request to https://<host>/consent.js fires and the banner appears on first visit. Then run POST /v1/sites/{cbid}/verify (dns/meta/file challenge) to unlock consent export and signed receipts for the domain.

IAB TCF v2.2

IAB TCF v2.2

Cookie Munch's IAB Transparency & Consent Framework (TCF) v2.2 support lets you run programmatic/RTB advertising in the EU by publishing a standard TC string every TCF-compliant ad vendor can read — the same signal Google, index exchanges, and DSPs expect from any registered CMP.

> Note: TCF is a paid ("Pro") capability. Enabling it without an entitled plan is blocked in the editor; server-side, framework: 'iab' is silently stripped back to 'none' for orgs without the feature.

Enabling TCF

In banner settings !’ Consent !’ TCF, toggle Enable IAB TCF, then configure:

Field	Description
CMP ID	Your IAB-registered CMP ID (0 = unregistered, for testing)
Publisher country code	2-letter ISO code (e.g. DE) — sets the TC string's publisher country field
Vendor IDs	Comma-separated GVL vendor IDs your site actually uses
Purpose IDs	Comma-separated TCF purpose IDs you request consent for
Auto framework by region	When on, TCF only activates for visitors in the EU/EEA/UK — everyone else gets the standard banner

The Ad-Settings panel

Beyond accept/reject, TCF requires visitors be able to make granular per-purpose, per-vendor choices. Add the TCF Ad-Settings panel element to your banner layout (Studio !’ element palette) to render it in-banner:

- Purposes — a consent toggle per TCF purpose, plus a legitimate-interest toggle wherever a vendor declares that purpose under legitimate interest instead of consent.
- Special features — opt-in toggles for precise geolocation and active device fingerprinting/ scanning.
- Vendors — a lazily-built, scrollable list (built on first expand, so it doesn't slow down initial banner render) with per-vendor consent + LI toggles and a link to each vendor's privacy policy.

The panel is driven live by the IAB Global Vendor List (GVL) fetched at runtime — there are no per-element props to configure in the editor; the inspector just shows a usage hint.

Legitimate-interest gating and objection

TCF v2.2 purposes 1, 3, 4, 5, and 6 are consent-only — Cookie Munch never renders a legitimate-interest toggle for them, even if a vendor's own declaration mis-labels them as LI-eligible. Only purposes 2, 7, 8, 9, 10, and 11 can show an LI toggle.

Where LI applies, the toggle defaults to established (on) — per TCF semantics, the visitor must actively untick it to object, rather than opt in. Two bulk controls make this practical for the visitor: Object to all and Allow all, which apply to every purpose- and vendor-level LI toggle at once (forcing the lazy vendor list to build first, so bulk actions never miss vendors that haven't been scrolled to yet).

Layer-1 vendor count disclosure

The vendor count element (tcfVendorCount) shows a first-layer disclosure like:

> "We and our {count} partners (IAB TCF vendors) process data."

{count} resolves live against the fetched GVL and vendor filter, satisfying the v2.2 requirement to disclose vendor count before the visitor drills into the full Ad-Settings panel. Its template text is editable from the inspector.

What gets encoded

The consented purposes, vendors, LI establishments, and special-feature opt-ins are encoded into a standard TC string via the IAB reference encoder. Purpose mapping from Cookie Munch's four categories:

Category	TCF purposes
Necessary	1 (store/access info on device) — always granted
Preferences	5, 6
Statistics	8, 9, 10
Marketing	2, 3, 4, 7

The TC string is exposed via the standard __tcfapi window function (ping, getTCData, addEventListener/removeEventListener) plus the cross-frame __tcfapiLocator so third-party iframes on your page can read it, exactly as any TCF-registered vendor script expects.

See also: Google Consent Mode v2 · Geo-targeting & regional rules · Banner settings

US State Privacy (GPP)

US State Privacy (GPP)

For US traffic, Cookie Munch emits the IAB Global Privacy Platform (GPP) string — the current standard for signaling opt-outs under CCPA/CPRA and the growing set of other US state privacy laws — plus the legacy US Privacy String for vendors that haven't migrated to GPP yet. Unlike IAB TCF, this is a free capability, independent of any paid plan.

Enabling it

In banner settings !’ Consent !’ US State Privacy:

Field

Description

Enable GPP

Turns on GPP string generation

US states

Comma-separated state sections to emit, e.g. usca, usva, usco

Enable US Privacy String

Turns on the legacy 4-character USP string alongside GPP

How the opt-out maps

GPP's US-state sections model consent as opt-out fields, not opt-in toggles. Cookie Munch maps your visitor's Marketing category choice onto every applicable field the requested section defines:

Marketing choice

SaleOptOut / SharingOptOut / TargetedAdvertisingOptOut

Denied

Opted out (1)

Granted

Did not opt out (2)

Only fields the specific state section actually defines are written — e.g. a state section without a SharingOptOut field simply omits it.

Legacy US Privacy String

The older 4-character USP string ([Version][Notice][OptOutSale][LSPA], e.g. 1YNN) was formally deprecated by the IAB in favor of GPP, but plenty of ad vendors still read it via __uspapi. Cookie Munch derives it from the same Marketing choice:

- Marketing denied !’ opt-out flag Y
- Marketing granted !’ opt-out flag N
- CCPA doesn't apply to this visitor !’ 1--- (not-applicable)

Note: Enable both GPP and US Privacy String together unless you're certain every downstream vendor already reads GPP — the two run side by side with no conflict.

Runtime API

Both strings are exposed the standard way: __gpp(command, callback, parameter) (ping, getGPPData, hasSection, getSection, getField, addEventListener/removeEventListener) and __uspapi('getUSPData', 1, callback) !’ { version: 1, uspString }. Cross-frame access is provided via the standard __gppLocator/__uspapiLocator bridge iframes for third-party scripts embedded on your page.

Auto-activation by region

If Auto framework by region is enabled, GPP/US Privacy only activate for visitors whose detected region resolves to the US — visitors elsewhere get the standard banner with no US-specific signaling.

See also: IAB TCF v2.2 · Geo-targeting & regional rules

Geo-targeting & regional rules

Geo-targeting & regional rules

Not every visitor needs the same banner behavior — GDPR requires opt-in in the EU, CCPA-style US laws are effectively opt-out, and plenty of jurisdictions require nothing at all. Geo-targeting lets you set a default banner mode and override it per country.

How region is detected

Every request to /config/:cbid (which the embed calls to hydrate the banner) resolves the visitor's region server-side, in order:

- An explicit x-cookiemunch-region header, if your CDN/proxy sets one.
- Common CDN geo headers — Cloudflare's cf-ipcountry, Vercel's x-vercel-ip-country (plus the region sub-header for US states), or a generic x-country header.
- A bundled, license-free IP!'country database (no MaxMind account required out of the box) as a fallback when no upstream header is present — self-hosted operators can point MAXMIND_DB at a MaxMind GeoIP2/GeoLite2 database for higher accuracy if they prefer.

The resolved region (e.g. de, us-ca) is returned alongside the config so all region-based logic — geo rules, TCF and GPP auto-activation — resolves client-side without a second round trip.

Setting rules

In banner settings !' Targeting !' Geo:

Field	Description
Default mode	The banner mode used when no rule matches — opt-in, opt-out, or off
Countries	Comma-separated 2-letter country codes for a rule (e.g. de, fr, es)
Mode	The mode applied when the visitor's region matches this rule

Rules are evaluated in order — the first match wins. A country-level rule (e.g. us) also matches state-level regions under it (e.g. us-ca), so you can set a US-wide rule and layer state-specific rules above it if needed. Add as many rules as you need with Add rule; remove one with Remove.

Modes

Mode	Behavior
opt-in	Nothing beyond necessary cookies fires until the visitor explicitly consents (GDPR-style)
opt-out	Non-necessary categories are granted by default; the visitor can decline (CCPA-style)
off	No banner shown; treated as no restriction

Global Privacy Control & Do Not Track

Independent of your geo rules, if the visitor's browser sends Global Privacy Control (navigator.globalPrivacyControl), Cookie Munch forces the effective mode to opt-out and treats it as an explicit decline — this holds even if a matching geo rule says opt-in or off, since GPC is a

legally-recognized opt-out signal in several US states. This mirrors how Google Consent Mode and GPP opt-outs behave.

Framework auto-activation

Turn on Auto framework by region (in the TCF settings) to have the banner automatically switch on IAB TCF for EU/EEA/UK visitors and GPP/US Privacy for US visitors — with everyone else getting the plain banner and no regulatory framework overhead.

See also: Google Consent Mode v2 · IAB TCF v2.2 · Vendors & ROPA for the GDPR-adequate region list used in vendor risk scoring.

Banner settings

Banner settings

The Settings dialog inside the banner Studio editor holds everything about your banner that isn't drawn directly on the canvas — blocking behavior, consent-framework signals, targeting, branding, and custom categories. It's organized into five tabs.

Behaviour

Setting	Description
Blocking mode	auto (Cookie Munch detects and blocks known trackers automatically) or manual (you tag scripts yourself)
Ignore selectors	CSS selectors to exclude from auto-blocking (e.g. #chat, .no-block)
Re-ask after	Slider, 0–730 days — how long a consent decision is remembered before the banner reappears (0 = never re-ask)
Suppress bots	Skip showing the banner to detected bot/crawler traffic
Observe cookies	Let the embed report live cookie names back to the scanner for declaration accuracy
Implied consent	Whether interacting with the page (e.g. scrolling) counts as implied consent

Consent

Signals sent to ad and analytics vendors, in one place:

- Consent Mode — enable/mode/wait-for-update. See Google Consent Mode v2.
- TCF — enable/CMP ID/publisher country/vendor IDs/purpose IDs/auto-by-region. See IAB TCF v2.2.
- US State Privacy — GPP + legacy US Privacy String. See US State Privacy (GPP).

Targeting

- Languages — default culture + auto-detect. See Languages.
- Geo — default mode + per-country rules. See Geo-targeting & regional rules.
- A/B testing (Scale plan) — split visitors between two banner variants and measure opt-in impact. See Analytics & A/B testing.

Appearance

Setting	Description
White-label (Pro plan)	Hide the "powered by" attribution from the banner
Brand & theme presets	Match-my-site detection and saved brand kits
Custom CSS	

Raw CSS injected into the banner for pixel-level overrides

Mobile breakpoint

Slider, 360–1024px — the width below which the banner switches to its mobile layout

Categories

Beyond the standard Necessary / Preferences / Statistics / Marketing categories, define your own custom categories — each with a label, description, default state, and an optional mapsTo mapping onto one of the standard categories (so a custom category's consent state still drives blocking and Consent Mode signals the same way a standard one would). Up to 24 custom categories per site.

> Note: Behaviour and Appearance settings apply immediately to the live config; paid features (TCF, white-label, A/B testing) show a badge and stay disabled in the UI until your plan includes them.

See also: Account security · Languages

Account security

Account security

Dashboard !’ Settings !’ Security covers everything about how you (and your teammates) authenticate — password management, active sessions, and connected sign-in methods.

Password

If your account has a password set, a Change password card lets you update it — current password, new password (minimum 8 characters). Changing your password signs out your other active sessions as a safety measure.

> Note: SSO-only accounts (created via Google, GitHub, or Discord with no password ever set) see a note explaining there's no password to manage instead of the change form — set one first if you want a password-based fallback login.

Sessions

Every device currently signed into your account is listed with:

- A best-effort browser + OS label (parsed from the device's user agent).
- The device's IP and how long ago it was last active.
- A This device badge on the session you're currently using.

Click Revoke on any other session to sign it out immediately — useful after losing a device or noticing unfamiliar activity. Two broader actions sit below the list:

Action	Effect
Sign out	Ends your current session only
Sign out everywhere	Revokes every active session across all your devices

Connected sign-in methods

Link or unlink OAuth providers — Google, GitHub, and Discord — from the same account. Each shows whether it's connected (and the linked email, when available) or offers a connect button.

> Note: You can't disconnect your last remaining sign-in method — if you have no password and only one OAuth connection, unlinking it is blocked so you're never locked out of your own account.

Related: organization danger zone

Settings !’ Danger zone (separate from Security) holds the destructive, confirmation-gated actions: deleting the current organization, or deleting your entire account. Both require typing DELETE to confirm and are irreversible.

See also: Banner settings

Languages

Languages

Cookie Munch separates two different "language" settings: the language your banner shows to site visitors, and the language your dashboard UI is displayed in. This page covers the banner side, configured under Banner settings !’ Targeting !’ Languages.

Banner language

Setting	Description
Auto-detect	When on, the banner matches the visitor's browser language automatically
Default language	The fallback culture used when auto-detect is off, or when the visitor's browser language isn't one you support

Ten languages ship built-in banner translations out of the box:

Code	Language
en	English
de	Deutsch
fr	Français
es	Español
pt	Português
it	Italiano
nl	Nederlands
pl	Polski
sv	Svenska
da	Dansk

Pick any of these as the default language from the dropdown. With auto-detect on, a French visitor's browser sends fr, and they see the French banner regardless of what you set as default — the default only kicks in when their browser's language isn't in the supported set, or auto-detect is off.

> Note: Every category label, button, and legal disclaimer element in the banner is translated for the selected language — including elements you add from the Ad-Settings panel and custom category labels you define yourself, where you supply the translated text.

Dashboard language

Separately, the Cookie Munch dashboard itself (the app you're using right now) is fully localized — it follows your browser/account locale automatically, independent of any individual site's banner language setting. You can run a French-language banner on a site while managing it from an English dashboard, or vice versa.

Per-region content overrides

Language and geo-targeting are independent, but they compose: you can pair a specific region rule with region-specific banner copy, so (for example) your German-market banner mode and its wording are both tailored, not just the mode.

See also: Banner settings · Geo-targeting & regional rules

Welcome to Cookie Munch

Welcome to Cookie Munch

Cookie Munch is a free, self-hosted consent management platform (CMP). It gives your sites a compliant cookie banner, an auditable consent log, and a dashboard for managing every property in your fleet from one place — without a per-domain SaaS bill.

Under the hood, Cookie Munch runs the same building blocks the big commercial CMPs do: a lightweight embed script that renders the banner and blocks scripts until consent is given, a signed and hash-chained consent record for every decision, and a config API that drives banner appearance, categories, and regional rules per site.

What you get out of the box

- A themeable consent banner (accept/reject/customize) with granular categories.
- A tamper-evident consent log you can export or query per visitor.
- Optional IAB TCF v2.2 support for programmatic advertising compliance.
- Cookie scanning that keeps your cookie declaration table current automatically.

Adding the banner to a site is a single script tag:

```
<script src="https://cdn.cookiemunch.net/consent.js" data-cbid="YOUR_SITE_ID" async></script>
```

From there, visitor choices flow into your dashboard in real time, and every downstream script (analytics, ads, embeds) can check consent state before it fires.