PermitCore

User Documentation

All Plans

November 2026

Table of Contents

QUICK START GUIDES

- PermitCore Quick Start
- Quick Guide to Store
- Quick Guide to Resellers
- Quick Guide to Stripe
- Quick Security Overview
- Business Continuity & Backups
- GDPR & Data Handling

INTRODUCTION

- What is PermitCore?
- Quick Start

SECURITY & ENCRYPTION

- How We Protect Your Data
- Overview
- Encryption & Access
- Security Deep Dive
- External Checks & API Discovery
- Business Continuity
- SSO / OAuth 2.0
- Geo-blocking
- Self-Hosted Deployment

LICENSE TYPES & CONTROL

- Overview
- Advanced License Management
- Trial Licenses
- Test / Sandbox Mode
- Named User Licensing
- License Templates
- License Transfer
- Offline Activation
- Node-locked / HWID
- Floating Licenses
- License Bundles
- Feature Flags
- Version Management
- Offline Grace Period

Post-Expiry Grace Period
Maintenance / Annual Support

SALES & REVENUE
Overview
License Store
Checkout providers & Paddle setup
Store setup checks
Customer Portal
Reseller Portal
Metered Billing
Stripe Auto-Charge
Dunning Management
Customer Notifications
GDPR Data Export
Account Recovery Package

DEVELOPER EXPERIENCE
Overview
API Overview
POST /validate
GET /nonce
POST /activate
Error codes
SDK Overview
.NET SDK
Node.js SDK
Python SDK
Java SDK
C++ SDK
PHP SDK
Go SDK
Unity SDK
Godot SDK
Webhooks
Analytics
Activity Log

ADMIN PANEL
Dashboard
Products
Licenses

Creating & Managing Licenses
Migrating from Other Systems
Active Device Capacity — Deep Dive
Users & Roles
Billing & Plans
Settings
Load from Template
Expiry Warning Widget
Weekly Email Digest
Expiry Notifications
Notification Preferences
Extend Expiry (Bulk)
Search by User Email
Customer Management
Test / Sandbox Keys

PermitCore Quick Start

Before you begin

The Free plan has no time limit and needs no credit card. The homepage Start free action opens Free signup. A three-calendar-month new-customer trial is a separate offer on eligible paid plans: choose a paid plan on Pricing or in the registration plan picker. Trial availability follows that paid plan's settings; checkout confirms eligibility and exact dates.

- Register a company account and verify your email.
- Keep a password manager ready for your license encryption passphrase.
- Choose an SDK or use the HTTP example below.

Your account password signs you in. Your license passphrase encrypts and reveals license keys. They serve different purposes and should be stored separately.

Quick Guide to Store

6. Publish your Store

Complete the full **create → implement → sell** path without building a separate checkout service. Connect your chosen Stripe, Paddle or Lemon Squeezy account, create a Store product linked to the software product you already licensed, then publish your branded storefront. PermitCore takes 0% sales commission and delivers a working license automatically after checkout. Follow the Store setup guide.

Quick Guide to Resellers

Next: work with resellers

On Enterprise, assign partner accounts access to specific products and issuance caps. Creating a key does not occupy a seat; new device registrations do. Existing node-locked reactivations reuse their seats. See reseller management.

Quick Guide to Stripe

Next: configure payments

Store purchases use the one Stripe, Paddle or Lemon Squeezy account selected by the vendor. Your own PermitCore subscription is managed separately in Billing. For usage charges, configure event pricing and customer mappings under metered billing.

Quick Security Overview

Next: secure the account

Enable MFA, configure recovery, and assign custom roles before inviting teammates. Read the security guide for the differences between browser-created keys, Store-issued keys, and optional escrow.

Business Continuity & Backups

Next: plan for disconnection

Choose between temporary signed offline grace and admin-issued offline activation tokens. Both have validity limits. See offline licensing and business continuity.

GDPR & Data Handling

Next: manage account data

Use Company Settings to export account records and Settings → Danger Zone to request erasure. The account ZIP includes encrypted license inventory and metadata, not plaintext keys or your passphrase. It is not a complete restorable database snapshot. See data export and the privacy policy.

What is PermitCore?

What you are building

PermitCore manages software license keys. You create a product and a license; your application asks the API whether the key is valid and decides which product features to unlock.

The hosted API base is https://api.permitcore.dev. The admin panel is https://app.permitcore.dev. Use your own API origin for a self-hosted deployment. SDK constructors take the origin without /api/v1.

Quick Start

Create and check your first license

1. Create a product

In the admin panel, open **Products → New Product**. Enter a name for the software you are licensing and save it. A product organizes keys; creating a product does not activate an installation.

2. Generate a key

Open **Licenses → Create License**. Select the product, choose a seat limit, and optionally set an expiry. For a desktop installation, enable node locking and use a stable device ID from your SDK. Choose **Test / Sandbox** while developing to label test records separately; their occupied seats still count against account capacity.

Enter an encryption passphrase of at least 16 characters and save it securely. After creation, reveal/copy the key with that passphrase. Configure passphrase recovery under **Settings → Security** before issuing production keys.

3. Validate the key

Replace the sample key below with the key you just created. Validation does not consume an activation seat. No admin API key or bearer token is required.

cURL

```
curl -X POST https://api.permitcore.dev/api/v1/validate \
  -H "Content-Type: application/json" \
  -d '{"licenseKey":"PERMIT-XXXX-XXXX-XXXX-XXXX","version":"1.0.0"}'
```

Example response

```
{
  "isValid": true,
  "productName": "My desktop app",
  "productId": "7f3e9a2b-1c4d-4e8f-9a6b-3d2c1e0f5a8b",
  "remainingActivations": 1,
  "expiresAt": null,
  "errorCode": null
}
```

HTTP 200 alone does not grant access. Require `isValid: true` and check that the response grants the product and features your application expects. By default, any active key belonging to your tenant validates successfully regardless of which of your products it was issued for — either check `productId` yourself, or pass the optional `expectedProductId` request field and let the server reject a mismatch (`errorCode: "WrongProduct"`). If rejected, read `message` and `errorCode`; see troubleshooting.

4. Activate each new installation

Call your SDK's activation method once when registering the installation. The SDK obtains the required nonce. With raw HTTP, first call `GET /api/v1/nonce`, then include that fresh nonce in `POST /api/v1/activate` together with the key and device ID. If you sell more than one product, also pass `expectedProductId` here (same field as validation) so a key issued for a different product is rejected at activation time rather than silently accepted. See the complete activation request.

On later launches, call validation. A local “activated” flag helps avoid repeated registration; it is not proof that a license is still valid. Node-locked reactivation with the same device ID does not use another seat or occupied device seat. Count-based activation increments on every successful call.

Command-line tools and CI

Building a command-line application? Follow the CLI licensing guide for CI, Docker and SSH for safe secret input, non-interactive exit codes and the separate activation/floating-seat lifecycle. For MCP servers, see paid tool gates.

5. Test the important outcomes

- A valid key unlocks only the intended product and features.
- A revoked or expired test key is rejected.
- A full seat limit produces a clear recovery message.
- A network failure does not automatically unlock your app.
- If you need offline access, test signed-token caching and expiry before release.

How We Protect Your Data

Understand the encryption boundary

Hosted BLUE/staging and GREEN/production share production PostgreSQL, Valkey, signing/encryption material and account records. Preview actions can change shared data; hosted QA is restricted to authorized Nexova records and sandbox payment connections. Browser sign-in storage is host-specific; the one-active-session policy applies across slots. Both slots remain on the same host, so this is not high availability or full disaster recovery. Isolated restore drills use their own separate fixtures.

Hosted PostgreSQL, Valkey, core application proxy links and API metrics use verified internal TLS, including server certificate names. Other monitoring transport remains a separate review scope. Self-hosted installations must configure their own transport and environment boundaries; visitor HTTPS alone does not establish all-link encryption.

PermitCore protects license storage, account access, and tenant boundaries in different ways. “Zero-knowledge” describes the default browser-created license workflow; it does not describe every operation the service performs.

Platform SuperAdmins have privileged operational access across tenants. This does not supply an unknown browser passphrase, but server-managed delivery material and opt-in escrow are recoverable by the service. Escrow recovery is audited and attempts to notify active CompanyAdmins; notification delivery is not guaranteed. The browser, delivered JavaScript and deployment secrets remain part of the trust boundary.

Workflow	What happens	Who can recover the key?
Admin-created license	Your browser generates and encrypts the key with your passphrase before submitting ciphertext and a lookup hash.	Someone with the passphrase or your configured recovery material.
Validation / activation	The app sends the plaintext license key over HTTPS in the POST body. The server hashes it for lookup without decrypting stored ciphertext.	This operation does not recover your passphrase or decrypt the stored key.
Store-issued license	The server generates the key for checkout delivery and stores protected delivery material using server-managed encryption.	The service can decrypt that material for authorized delivery or customer key reveal.
Optional passphrase escrow	You explicitly send your passphrase for encryption with a server-managed key.	The service can recover it. Enabling escrow changes the zero-knowledge boundary.

Keep license keys, API keys, passphrases, and webhook secrets out of application logs and analytics.

Using POST avoids URL/access-log exposure; it does not stop your own code or middleware from logging request bodies.

The supplied API gateway, Public, Admin and configured outer NPM access formats retain generated request IDs and operational metadata, but omit request paths, query strings, Referer, credentials and bodies. Client IP addresses remain operational data. Reset-mail, tenant SSO exchange and global API failure records use bounded type/status/code metadata without raw exception messages or private reset URLs; global failures use route templates rather than bound paths. Verification, welcome, Store/Portal recovery, OTP and license-delivery mail failures also omit raw mail payloads. License-delivery retries receive a safe exception so the job error record cannot echo the key/body; intended emails and existing retry settings are retained. Reviewed billing/price administration, Stripe webhook and dunning/expiry/trial-reminder error records use the same bounded diagnostics without changing charges, refunds, grace or retry decisions. Reviewed contact/admin-test, GDPR, escrow, self-host and digest failure records also omit raw exception payloads, preserving existing operation results/audits and best-effort notification behavior. Failed mail is not guaranteed delivery. Reviewed license/webhook, metering, analytics, expiry, retention and self-host job failure records also omit raw exception payloads. Existing responses, retry/fallback and business decisions remain. Operational identifiers/emails/IPs, lower-level library logs, job arguments, stored fields, external proxies and older records remain separate review scopes. Treat emailed reset, verification and download links as private; referrer protection does not remove the original URL from browser history or every logging layer.

Database and SMTP connection tests return safe operation messages instead of raw provider exceptions. Check the saved server, port, credentials and TLS settings; SMTP also needs a valid sender address. A missing mail provider remains a specific configuration error. Successful test sending is unchanged and confirms SMTP acceptance, not inbox arrival. Reviewed startup and installation catch records retain bounded code/type diagnostics without raw exception text. A skipped startup task is not successful migration or deployment; lower-level library logs and older records remain separate review scopes.

The supplied Console/File log outputs omit raw selected HTTP, database and authentication library diagnostics, plus attached exception payloads, while retaining bounded code/status metadata. Other custom sinks, job history and proxy error logs remain separate. Existing file-count/size rotation is not a calendar erasure or backup-deletion guarantee; this change does not purge old data.

Configured nginx error outputs for Public, Admin, gateway and outer NPM retain only time, hop, severity and a fixed failure category; raw URL/message payloads are discarded before storage. The isolated collector must be healthy before ingress upgrades. Error delivery is best effort: a logging outage must not stop customer requests. Direct subprocess/library output, custom overrides, external proxies and old records remain separate; no history is purged.

Normal Certbot diagnostic files omit messages, arguments, private-key data and exception text. New records contain only time, fixed logger family, level, an exception flag and a diagnostic category. Console output and command results keep their behavior; original rotation and historical records remain. This is not certificate issuance or renewal acceptance; custom entrypoints and direct process streams are outside this file-only policy.

NPM application logger channels omit raw certificate objects, commands, credentials and error payloads. Existing severity, scope and fixed operation categories remain. Command results and errors,

certificate operations and initial-admin login are unchanged. Direct subprocess/library output and stored audit/history remain separate; this change does not purge records.

Protected API 401 exception responses preserve fixed sign-in explanations and otherwise use a generic authentication message instead of raw exception text. Upgraded Store mail deployments can queue only an outbox reference; the worker uses the original purchase download snapshot. Existing queued jobs remain compatible and may retain older arguments. Outbox data, legacy history and queue recovery remain separate review scopes.

Self-host check-in failure audits use bounded diagnostics in Details and an independent tenant-scoped save, without retrying failed check-in writes. If audit storage is unavailable, the existing response is retained and a safe warning records the failed audit attempt. This is best-effort failure auditing; historical records and other error fields are not rewritten.

In account exports, a Store-issued license's `EncryptedKey` can be a `store-order:` reference rather than passphrase-decryptable ciphertext. The ordinary ZIP excludes order delivery material and server encryption secrets; it cannot recover those Store key values by itself. Preserve delivered keys and arrange authorized recovery while the service is available. A standalone bulk Store-key recovery package is not currently provided by this ZIP.

Overview

Protect your account

For a configured SuperAdmin MFA rollout, the notice shows a fixed deadline with 14 days from activation to enroll. Open **Set up MFA**, scan the locally generated QR code (or enter the manual time-based key), save the one-time recovery codes separately, then verify the six-digit code. After the deadline, unenrolled SuperAdmins must finish setup before business access, including existing sessions. Company accounts retain their existing tenant policy.

If an unfinished setup key or recovery codes were exposed, use **Restart setup**. This generates a new authenticator key and recovery codes and invalidates the previous unfinished set. Reloading the page does not redisplay old setup material; starting setup again generates a fresh set. Existing enabled authenticators cannot be replaced this way.

If your authenticator is lost, enter a saved recovery code with your password at sign-in. Each code works once. To replace an enabled authenticator, first disable it in Settings → Security using both your password and an existing authenticator or recovery code. Password reset alone does not remove MFA. Never send recovery codes to support in chat.

Sign-in and authenticated API responses instruct browsers and proxies not to store them. API writes and authorization failures use the same no-store policy. Public catalog reads can retain a short cache lifetime; unchanged frontend scripts use separate content-derived cache addresses. These response rules do not clear sign-in storage or change the 30-minute Admin inactivity policy.

1. Enable MFA in **Settings → Security** and save the backup codes separately from your authenticator.
2. Give teammates custom roles containing only the menus and actions they need. See Account & team.
3. Keep admin API keys on trusted servers, in a secret store. Never ship them inside a desktop app, browser bundle, mobile app, or public repository.
4. Store your encryption passphrase in a password manager and configure recovery before distributing keys.
5. Subscribe to `license.abuse_suspected` webhooks and review unusual activation or login activity.

Feature availability depends on your plan. A hidden menu can also mean your custom role does not have access; ask a Company Admin to check both before troubleshooting an integration.

Encryption & Access

Browser-created licenses and passphrase recovery

General-purpose random-key snippets are educational examples, not issued PermitCore licenses or offline tokens. Use Admin license creation or Store fulfillment for issuance. See the C#, Python and Node.js key-generation guide for the distinction between a random identifier, a server record and signed proof.

1. The browser creates a random key in the format PERMIT-XXXX-XXXX-XXXX-XXXX.
2. PBKDF2-SHA256 derives a 256-bit encryption key from your passphrase and tenant salt, using 600,000 iterations.
3. AES-256-GCM encrypts the license key with a random IV. The browser sends ciphertext, IV, and the SHA-256 lookup input.
4. The server applies HMAC-SHA256 with a server-held secret before storing the lookup value.
5. For validation, the server receives the customer-supplied plaintext key transiently, recomputes that lookup value, and checks the license restrictions.

A stolen database alone does not contain the passphrase needed to decrypt browser-created keys, or the server secret needed to check guesses against the HMAC lookup values. Protecting the browser, passphrase, and server secrets remains necessary.

Keep the original passphrase

The passphrase is needed to reveal or copy encrypted keys. Changing your account password does not change it. Losing it does not stop online validation of keys your customers already hold, but you may lose the ability to retrieve their plaintext from the admin panel.

Recovery key

Under **Settings → Security → Passphrase Recovery**, configure the browser-based recovery option and save the generated recovery key securely. Recovery requires the matching recovery material; simply resetting your login password cannot decrypt old license ciphertext.

Optional escrow

Escrow sends the passphrase to the service for server-managed encryption. Use it only if this recovery model fits your organization's requirements. It allows service-side recovery and therefore removes the zero-knowledge property for that passphrase.

Security Deep Dive

Technical security reference

Area	Implementation	Integration consequence
License ciphertext	AES-256-GCM; browser PBKDF2-SHA256 derivation	Retain the passphrase and recovery material.
License lookup	SHA-256 followed by server-keyed HMAC-SHA256	Send the original key in an HTTPS POST body.
Passwords	Argon2id	Use unique passwords and MFA.
Access tokens	RS256 JWT signatures and rotating refresh tokens	Use the supported login/refresh flow; never embed a signing secret in clients.
Offline tokens	ECDSA P-256 / SHA-256 signatures	Verify with the trusted public key and enforce token expiry/binding.
Tenant isolation	Application authorization plus PostgreSQL row-level policies on protected tenant tables	Admin operations remain scoped to your tenant. Self-hosted installations must use the restricted app database role.
Webhook authenticity	HMAC-SHA256 over the raw body	Verify the signature and deduplicate event IDs. A signature alone does not prevent replay.

Rate limits

Public endpoints have per-IP controls in addition to the tenant's plan allowance and per-license abuse checks. A higher plan allowance does not remove a per-IP restriction. Handle HTTP 429, honor Retry-After when present, and avoid synchronizing every customer's checks at the same moment. See error handling.

Operational security

Audit retention commits each cleanup batch together with a count-only audit receipt. A failed batch rolls back and the job reports incomplete work; earlier committed receipts remain. Tenant cleanup follows plan windows. System audit cleanup requires a separate operator switch, disabled by default; deploying an update does not enable it. Ordinary application access still cannot delete audit rows. This is not an immutable archive or complete erasure guarantee.

Activation history minimization clears IP, country, city and end-user email on records older than the configured window (365 days by default), including records whose IP is already empty. The daily job preserves activation rows, device binding, license relationships and usage counters. Retained device/software metadata, old backups and job-failure handling remain separate; this is not complete

anonymization or an exact calendar-erasure guarantee.

Production uses HTTPS and security headers. New backup generations stream database dumps into GPG encryption and verify off-site copies before expiry. New deployment bundles include the encrypted database and selected configuration bytes with a verified manifest. Complete new copies become eligible after 30 days and after verification; holds, failed verification and the newest deployment recovery point can extend retention. Legacy generations remain retained separately under operator-managed review; old gzip compression is not encryption. An isolated data restore does not prove full application recovery. The public status page reports a database-connectivity probe, not independent checkout, email or full license-flow monitoring. For current infrastructure and data-residency information, see the security page. For data-processing terms and subprocessors, see the DPA and privacy policy.

Report a suspected vulnerability to info@permitcore.dev. Include affected URLs, reproduction steps, and impact; do not include real customer keys or unrelated personal data.

External Checks & API Discovery

External checks and API discovery

Frontend deployment images support an HTTP Content Security Policy in report-only or enforced mode. The target allows the Blazor WebAssembly runtime without general script eval or inline event handlers; report-only observes the target while retaining existing protections. Self-hosted operators must verify their API origin and browser/payment flows before enforcement. Dynamic branding still uses inline styles and HTTPS images.

Browser policy diagnostics go to the first-party API. Only grouped surface, directive, source-family, disposition, counts and timestamps are retained in memory for up to 24 hours. Raw URLs, query parameters, tokens and script samples are discarded. A SuperAdmin session can inspect the no-store summary; API keys cannot. These untrusted diagnostics are not an audit or proof that every browser sent a report.

The hosted service requires TLS 1.2 or newer, with TLS 1.3 enabled. Self-hosted operators manage their own TLS edge. The Security page records dated, host-specific SSL Labs and Mozilla Observatory results; these are not an application audit or certification.

The gateway root GET / returns public JSON discovery links with no authentication or caching. This static response does not check API availability, the database or payments. Use /healthz for application readiness and /api/status for the separately documented database probe. Unknown application routes retain their existing behavior.

Business Continuity

Backups and business continuity

Backups recover account data. Offline tokens and signing-key export address whether your software can keep verifying licenses without a live service. Plan for these separately.

Your own recovery package

- Keep account-data exports and license-key exports separately, along with the passphrase/recovery material needed to use encrypted keys. The account-data ZIP includes encrypted license values, IVs and salt, not plaintext keys or private signing keys. It also includes customer/Store/payment records, usage, configuration and historical public verification keys; asset URLs are references.
- Keep your product identifiers, integration configuration, and trusted signing public keys with your release records.
- Test the offline flow before a customer needs it; a machine without network access cannot fetch its first public key.
- For self-hosting, back up both the database and required deployment secrets, then test a restore in an isolated environment.

Available continuity options

Option	What it provides	Limit
Offline activation (Professional+)	A signed token verified on the customer's machine.	Only works until that token expires and cannot learn about revocation while fully offline.
Signing-key export (all plans, including Free)	Your own copy of signing material for independent issuance.	Protect the private key; anyone holding it can issue tokens trusted by your software.
Self-hosting (Enterprise)	Your own deployment, database, and operational control.	You operate and recover the infrastructure. See the deployment guide.

Rotation versus compromise

Read the signing-key rotation guide before changing a signer. Rotate activates the new signer immediately; deliver its approved public key through your trusted app/configuration update process. Regeneration does not update disconnected applications or mark every historical deprecated key compromised.

The exit plan and recovery checklist separates offline proof, encrypted data exports, signing-key recovery and the services your app still needs. The account-data ZIP is a selected export, not a

restorable database backup.

Take your keys anytime: signing-key export and the associated rotation, regeneration and history tools are available on every plan, including Free, to your CompanyAdmin. In **Settings → Security → Business Continuity**, read the warning and acknowledge responsibility before exporting. The private PKCS8 key is a vendor secret; customer apps receive only trusted public verification keys. Hosted offline-token generation remains subject to its separate plan entitlement, and self-hosting remains Enterprise.

For an end-of-life release, inventory sign-in, activation, refresh, downloads and hosted-feature dependencies. Test reinstalls on a clean device after the signed token deadline with your endpoints unavailable. A final offline build is your own release strategy; an exported key or backup does not distribute that build. See the software ownership and shutdown planning guide.

Rotate creates a new active signing key and keeps the previous key available as deprecated for verifying older tokens. **Regenerate** marks the prior key compromised and revokes tracked tokens. Online verification can apply that change immediately; a disconnected app with an old public key cannot discover it until it reconnects or its token expires. Preserve the relationship between a token's `kid` and its verification key.

These mechanisms do not make every existing online license permanently offline-capable. Issue and distribute the required tokens or deploy your continuity solution before relying on it.

SSO / OAuth 2.0

Company single sign-on

Company SSO supports tenant-configured Google, Microsoft or GitHub sign-in for your team. It is separate from customer license validation and Store customer accounts.

1. Open **Settings → SSO** and choose an available provider.
2. Create the corresponding application in that provider's console.
3. Copy the redirect URI shown in PermitCore exactly into the provider settings.
4. Enter the client ID and secret, enter the required email domain, and save.
5. Create the teammate's PermitCore record first, or explicitly enable Auto-provision and select its required default role.
6. Test with a non-owner account before rolling it out to the team.

With Auto-provision off, only existing active tenant users can sign in. When you explicitly enable it, a matching-domain user can be created with the configured default role; missing or deleted roles reject provisioning. **Require organization SSO** blocks password and general social sign-in for tenant users, including renewal of sessions without a verified required-provider binding. Only enabled required SSO providers are accepted. Existing sessions without a binding need one fresh SSO sign-in. Test the provider before enabling and keep an administrator recovery process. API keys and authorized support impersonation retain their separate rules.

Google profile lookup sends its provider access token in the Authorization header over HTTPS. PermitCore returns a short-lived one-time sign-in code to the Admin callback and redeems it through a POST request; raw local access and refresh tokens are not put in the callback URL. Existing client IDs, redirect URIs and configured sign-in flows remain unchanged.

Geo-blocking

Country and IP restrictions

Set **Allowed Countries** in a license's metadata using comma-separated ISO country codes such as US, DE, GB. An empty list means no country restriction. Both validate and activate check the configured restrictions.

The country is inferred from the request IP. A VPN or proxy can change the apparent country; this is not proof of residence or a complete compliance control. If a legitimate customer is blocked, check their exit IP and the license's allowed countries before changing the policy. IP allowlists provide a separate network restriction.

Self-Hosted Deployment

Deployment

Self-Hosted Deployment (Enterprise)

On direct connections, the Public and Admin nginx images serve the published Brotli-compressed WASM files when the browser accepts them, with gzip or uncompressed fallback. Cloudflare requests retain the existing gzip path because that proxy rewrites the client's encoding header; static Brotli there remains held pending complete negotiation verification. Keep the generated _framework files and their .br/.gz siblings together when deploying. Preserve Vary: Accept-Encoding, the WASM content type, security headers and cache rules in any additional reverse proxy. No extra compression module is required. Smaller transfers do not eliminate browser startup CPU work.

A self-hosted Enterprise instance runs the exact same PermitCore.Api/ PermitCore.Web.Admin/ PermitCore.Web.Public images PermitCore runs for its own SaaS — there is no separate self-hosted build target. Self-hosting is enabled purely through environment variables, and a self-hosted tenant owns the entire stack — database, signing keys, everything — from day one.

Isolated local development

On localhost, the public frontend normally uses http://localhost:5127 for its API. To run a separate local sandbox alongside that environment, set LocalApiBaseUrl in the frontend's served appsettings.json to another HTTP(S) loopback endpoint, such as http://localhost:5190/. Remote URLs, embedded credentials and query/fragment URLs are rejected for this override. Production and staging continue to derive their API hostname from the frontend hostname.

Prepare the deployment

Arrange Enterprise provisioning before installation. You need a host that can run the supplied containers, persistent database storage, DNS and HTTPS for your API and web frontends, access to the private image registry, and a configured mail provider for account emails.

1. Choose the API, admin and public-site hostnames. Configure their URLs and the allowed origins consistently in the supplied environment file.
2. Set the trusted API host in both SELF_HOST_TRUSTED_API_HOST and ALLOWED_HOSTS. A hostname missing from either allowlist is rejected even if DNS and TLS work.
3. Keep database credentials, signing material and registry access outside source control. Save the one-time self-host license securely before closing the provisioning screen.
4. Back up the database and the secrets needed to decrypt its protected fields. Test a restore before relying on the instance for customer licensing.

First startup and database roles

Keep the maintenance database connection (DB_CONNECTION) separate from the restricted runtime connection (DB_CONNECTION_APP). First startup runs migrations and creates the initial SuperAdmin

using a short-lived maintenance connection. Normal requests keep the restricted role and tenant isolation. Set a unique ADMIN_EMAIL and strong ADMIN_PASSWORD in your private environment file before startup. Once an administrator exists, changing those values does not reset its password or create another administrator. Missing bootstrap credentials do not create an account; inspect server logs if installation remains incomplete.

How it works

1. **Requesting a deployment** — POST /api/v1/selfhost/request (CompanyAdmin+, gated behind the self_hosting feature key, Enterprise-only) creates a SelfHostLicense row (SH-XXXX-XXXX-XXXX-XXXX format key, hashed the same way as customer license keys — plaintext is never persisted). One active self-host license per tenant is enforced at the database level.
2. **Provisioning** — a SuperAdmin reviews the request at /admin/selfhost, issues a scoped GitHub token for the private image registry, and provisions the instance. The customer then downloads a pre-filled .env (plaintext key shown once) and runs the provided docker-compose.selfhost.yml.
3. **Check-in** — the running instance calls a public check-in endpoint daily. This is a no-op unless SelfHost:LicenseKey is configured, so the same image is harmless running as PermitCore's own SaaS.
4. **Continuity tokens** — for the "PermitCore the company disappears" case, a SuperAdmin can issue a long-lived (commonly 20-year) signed continuity token, using the same per-tenant ECDSA P-256 keypair already used for offline activation tokens. Once configured in the instance's .env, it's verified entirely locally — zero network calls — and bypasses check-in while the continuity token is valid.
5. **Distribution** — private container images, per-customer pull-token issuance handled by a SuperAdmin. Revoking a pull token blocks new image pulls only — it never touches an already-running container.

> **Effect of a self-host license lapse:** the check-in path is a separate code path from the public /api/v1/validate//activate endpoints a self-hosted instance's own downstream customers call — different entity, different hash namespace, zero shared references. If a self-host license lapses, the only effect anywhere is a banner plus a reseller-feature flag disabled locally on that instance, after a 30-day grace period. The instance's own customers' license validation and activation are not disabled by that self-host check-in failure. They still depend on the instance being healthy and the individual customer's license rules.

Key versioning — rotation vs. emergency response

Routine key rotation and "this key may have leaked" are different situations with different blast radius, so they're two different endpoints on purpose — an earlier design that only offered one destructive action meant any rotation forced every offline-installed customer to reactivate, even when nothing was actually wrong.

Action	What happens	When to use it
POST /api/v1/signing-key/rotate	Current Active key becomes Deprecated, a new key becomes	Safe to run on a schedule — annual hygiene rotation, offboarding an

Action	What happens	When to use it
	Active. **No tokens are revoked** — a Deprecated key stays fully valid for verifying tokens it already signed.	engineer who had export access — without disrupting any already-running offline install.
POST /api/v1/signing-key/regenerate	Current key becomes Compromised — rejected outright at verification regardless of any individual token's own status. Every currently-tracked, non-expired offline token for the tenant is also revoked.	Use after a suspected compromise. Requires an explicit confirmation flag. Distribute the replacement trusted key and tokens to affected installations.

GET /api/v1/signing-key/versions returns a read-only history of every key version (id, status, timestamps, no key material) for a vendor to audit their own rotation history.

An already disconnected verifier cannot learn that a key was compromised. Server-side revocation takes effect only where the new trust state is available; plan how offline installations receive key-status updates and replacement tokens.

Verify and operate the instance

- Confirm admin sign-in, verification/reset emails, product creation and license activation against your own API hostname.
- Check validation from a customer application and test an offline token with the instance's trusted signing key.
- Monitor API availability, database capacity, mail delivery, backups and certificate expiry.
- Before upgrades, review release instructions, take a restorable backup and verify the integration in a staging instance.

Overview

License Types & Control

License Types & Control

Subscription renewals & past_due

For monthly and annual Store purchases, a paid Stripe subscription invoice extends the existing license to the end of its paid billing period. The customer keeps the same key. expiresAt is the paid-through date, not the grace deadline.

A failed renewal sets paymentStatus to past_due. Validation and activation continue through the configured payment grace window: 14 days by default, measured from the paid-through date. Vendors can set 0–365 days in Store → Settings; a license's explicit PostExpiryGraceDays takes precedence, including zero. unpaid keeps the same deadline. Once it is reached, access expires. A successful payment restores active and advances the paid-through date.

During payment grace, the response includes isInGracePeriod: true and paymentStatus: "past_due" or "unpaid". Show a card-update notice while isValid remains true. Non-subscription licenses return no payment state. Offline cache tokens and newly generated offline activation tokens for Store subscriptions are capped at paid-through plus grace. Reconnect to receive a refreshed signed cache token after payment; already-issued offline tokens retain their signed deadline and cannot learn about a cancellation while disconnected.

Scheduling cancellation at period end preserves access through that paid period. When Stripe ends the subscription, customer.subscription.deleted revokes the license. Duplicate invoices and webhook deliveries have no additional effect; older payments never shorten the paid period or reactivate a revoked key.

Register invoice.paid, invoice.payment_failed, and customer.subscription.updated alongside checkout.session.completed, payment_intent.payment_failed, checkout.session.expired, customer.subscription.deleted, and charge.refunded on the vendor's Store webhook endpoint. This is the customer's Store subscription; the vendor's own PermitCore billing is managed separately.

Every licensing model in one platform. Perpetual, subscription, trial, metered, floating, node-locked, bundled — all controlled by the same license key format. Mix and match per product without changing your integration.

Perpetual & Subscription
Set
ExpiresAt
to null for lifetime licenses or a future date for subscriptions. Renewal extends the date — the key stays the same.
Trial Licenses
Issue trial keys instantly, no card required. Automatically expire on a set date. Convert to paid by extending the expiry — same key, no re-integration.
Floating / Concurrent Licenses

Seat-based licensing with real-time checkout/checkin. Enforce maximum concurrent users across any number of machines.

Node-locked / HWID

Bind a license to a specific machine fingerprint. The activation API tracks distinct device IDs up to the key's seat limit; the same ID reuses its seat.

License Bundles

One key that unlocks multiple products simultaneously. Great for suites — one purchase, one key, access to everything in the bundle.

Feature Flags per License

Store a JSON feature map on each license. Your app reads which modules are enabled at validation time — no new keys required for upsells.

Version Management

Lock a license to a maximum software version. The API checks configured version bounds when your request supplies a version. Maintenance dates are separate metadata.

Offline Activation

ECDSA P-256 signed challenge/response for air-gapped machines. No internet required after initial activation file generation.

Test / Sandbox Mode

Sandbox keys behave identically to production keys but are excluded from billing counters and analytics. Safe for CI/CD pipelines.

License Templates

Save and reuse license configurations — expiry, max activations, feature flags, and metadata. One click to apply a template when creating a new license.

Grace Periods

Offline grace period (validate without network for N days) and post-expiry grace period (continue working N days after expiry). Configurable per license.

Maintenance & Annual Support

Track maintenance contract end dates separately from license expiry. Returned to your app for its own update/support policy; the date does not itself reject validation.

Advanced License Management

Licensing Features

Advanced filtering

Click the **Filters** button on the Licenses page to open the advanced filter panel. You can filter your license list by multiple criteria simultaneously:

Filter	What it does
Status	Show only Active, Revoked, or Expired licenses
Expires from / to	Find licenses expiring within a date range (e.g. "next 7 days")
Tag contains	Filter by tag substring — type "beta" to find all beta licenses
Min activations %	Slider (0–100%) — e.g. set to 80% to find nearly-saturated keys

Active filters show an **"ON" badge** on the Filters button. Click "Clear" to reset all filters at once.

Bulk update

Select multiple licenses using the checkboxes, then click **"Edit Selected"**. The bulk edit modal lets you update up to 500 licenses in a single operation:

Field	Behaviour
Max Activations	Set a specific number (must be ≥ current activations on each key)
Expiry Date	Set a new expiry, or tick "Clear expiry" to make keys perpetual
Add Tags	Tags are *merged* with existing tags — existing tags are not removed
Add Notes	Text is *appended* to existing notes (one new line per update)

> Fields you leave blank are **not modified** — only fill in what you want to change.

SDK integration snippets

Click the **Integrate** button in the top-right of the Licenses page to open the SDK snippet generator. It shows copy-ready integration code in **curl, Python, Node.js, C#**, and **PHP** — pre-filled with your actual API base URL. Use these to quickly wire up license validation in your application.

Expiry notifications

PermitCore automatically emails **Company Admins** when licenses are about to expire. Emails are sent at three checkpoints:

- **30 days** before expiry — early warning, plan renewals
- **7 days** before expiry — action required soon
- **1 day** before expiry — urgent reminder

Each license gets at most one email per checkpoint (no daily spam). The email lists every affected key with product name, expiry date, days remaining, and current activation count.

> Expiry notification emails require your account to have a working mail provider configured (the deployment mail configuration (managed by the operator)). Emails are sent daily at 08:00 UTC.

Trial Licenses

Licensing Features

Trial licenses let potential customers try your software for a fixed number of days — no manual follow-up required. The license key is valid from creation and automatically expires after the configured trial period.

Creating a trial license

1. Go to **Licenses → New License**
2. Toggle **"Trial License"** on
3. Enter the trial period in days (1–365; default 30)
4. Generate the key and send it to the prospect — expiry is set automatically

> Trial licenses are visually marked with a **purple Trial badge** in the license list and detail view. The expiry date is computed from the moment the license is created, not from when the customer first activates.

Behaviour

Setting	Trial license	Regular license
Expiry	Auto-set to *now + TrialDays*	Optional, set manually
Validation	Standard — returns isValid: false once expired	Standard
Conversion	Revoke + issue new regular key, or extend expiry in Edit Metadata	—
Badge	IsTrial: true returned in API and portal	IsTrial: false

Converting a trial to full license

When a trial customer purchases, you have two options:

- **Extend:** Go to the trial license → Edit Metadata → clear or extend Expires At. The customer keeps their existing key.
- **Replace:** Revoke the trial key and generate a new non-trial license. Send the customer the new key.

Test / Sandbox Mode

Licensing Features

Sandbox licenses label integration and test records separately from live licenses. Test licenses work **exactly like production licenses** — validate, activate, floating seats, feature flags, offline tokens — and their occupied seats **count** against your MaxTotalActivations account active device capacity, including after expiry or revocation.

No separate sandbox environment is needed. Every tenant has a single environment with both live and test licenses. The filter in the license list defaults to **Live only** to keep your daily workflow clean.

Creating a test license

1. Go to **Licenses → Create License**.
2. Fill in the form as normal (product, max activations, expiry, features…).
3. Before encrypting, toggle **Test / Sandbox license** on.
4. Proceed through passphrase and generate as usual.

Identifying test licenses

Test licenses are visually distinguished throughout the admin panel:

Location	Indicator
License list	Yellow **Test** badge next to the encrypted serial
License list filter	Dropdown: *Live only* (default) / *Live + Test* / *Test only*
License detail	Returned in LicenseDto.isTest = true via API

Billing behavior

Action	Live license	Test license
Activation	New device seats count against total account capacity	Occupied seats count; the same account capacity limit applies
Validate	Standard	Identical — same response shape
Floating check-in/out	Standard	Identical
Billing dashboard	Shows total occupied device seats	Included in the account capacity count

> **Cannot switch after creation.** The IsTest flag is set at creation time and cannot currently be changed after the license is saved. If you need to promote a test license to production, revoke it and generate a new live license.

Named User Licensing

Licensing Features

By default, PermitCore tracks *how many* activations a license has consumed and *from where* (IP address, country, device ID). **Named user licensing** goes one step further: it records *who* activated the license — the end user's email address.

This is especially useful for B2B software sold to companies where IT admins need to audit which employees are using which licenses. Instead of seeing "5 activations on 5 devices", you see "activated by alice@acme.com (2 devices), bob@acme.com (1 device), carol@acme.com (2 devices)".

Raw HTTP activation also requires a fresh `nonce` from `GET /api/v1/nonce`. The snippets here focus on device/user fields; use the complete activation flow when sending a request.

How to use it

Pass the optional `userEmail` field when calling `POST /api/v1/activate`:

JSON — activate with named user

```
{
  "licenseKey":        "PERMIT-A3K7-MN2P-QR9X-TZ5W",
  "deviceId":          "a3f8d2b1c9e4...",
  "deviceName":        "Alice-ThinkPad",
  "userEmail":         "alice@acme.com",
  "expectedProductId":  "7f3e9a2b-1c4d-4e8f-9a6b-3d2c1e0f5a8b"
}
```

JavaScript / Node.js

```
const response = await fetch('https://api.permitcore.dev/api/v1/activate', {
  method: 'POST',
  headers: { 'Content-Type': 'application/json' },
  body: JSON.stringify({
    licenseKey: key,
    deviceId:   hwid,
    userEmail:  currentUser.email,  // pass the logged-in user's email
    expectedProductId: '7f3e9a2b-1c4d-4e8f-9a6b-3d2c1e0f5a8b' // optional
  })
});
```

Viewing named user data in the admin panel

Go to **Licenses → View → Activation History**. The table includes a **User Email** column that shows the email address recorded at activation time.

Key considerations

- **Optional field** — if `userEmail` is omitted, the activation is recorded without a user email. Licenses without it still work normally.
- **Not enforced** — PermitCore does not restrict one email to one seat; it records the field for

reporting only. Enforce seat limits via MaxActivations.

- **Max 256 characters** — the field is truncated server-side if longer.
- **Immutable after activation** — the email is stamped at activation time and not updated on re-activation.

> **Privacy:** User emails are stored only within your PermitCore instance (your database). Use the minimum personal data needed for your workflow and review the privacy policy and subprocessors. Follow your own GDPR obligations when collecting end-user emails.

License Templates

Licensing Features

If you create many licenses with the same configuration — same max activations, same expiry, same feature flags — **License Templates** let you save that configuration once and apply it in one click when creating new licenses.

Creating a template

1. Go to **Templates** in the left sidebar (or navigate to /templates)
2. Click **New Template**
3. Fill in the fields you want to pre-populate: **Name** — e.g. "Pro Annual", "Starter Trial", "Enterprise Seat" **Max Activations** — default 1 **Expires After (days)** — leave blank for never-expiring licenses **Version Range** — MinVersion / MaxVersion to lock the template to specific software versions **Activation Period (days)** — sliding window from first activation **Trial** — check if this is a trial template and enter the trial duration **Notes** — internal description for your team
4. Click **Save Template**

Template API

Method	Endpoint	Description
GET	/api/templates	List all templates for the tenant
GET	/api/templates/{id}	Get a single template
POST	/api/templates	Create a new template
PUT	/api/templates/{id}	Update a template
DELETE	/api/templates/{id}	Delete a template

All template endpoints require a **CompanyAdmin** JWT token (Authorization: Bearer <token>).

POST /api/templates — request body

```
{
  "name":              "Pro Annual",
  "maxActivations":    5,
  "expiresAfterDays":  365,
  "isTrial":           false,
  "trialDays":         0,
  "minVersion":        "2.0.0",
  "maxVersion":        "3.9.9",
  "activationPeriodDays": null,
  "featuresJson":      "[\"export\",\"api\",\"pro\"]",
  "notes":             "Pro plan annual — 5 seats, expires 1 year from issue"
}
```

Save time at scale: If you issue 500 licenses a month, using templates means you never have to remember which feature flags go with which plan tier — just pick the template and the correct config is applied automatically.

License Transfer

Licensing Features

When a customer needs to move their license to a new device — because their hardware was stolen, replaced, or they changed jobs — **License Transfer** lets you instantly reset all activations so the license can be used on new machines.

What transfer does

- Deletes *all* existing activation records for the license
- Resets `CurrentActivations` to `0`
- Leaves the license key itself unchanged — the customer reuses the same key
- Logs the action in the **Activity Log** with the number of activations cleared

How to transfer (admin panel)

1. Go to **Licenses → View** for the license
2. In the **Activation History** header, click **Transfer** (blue button)
3. Confirm in the dialog — all activations are cleared immediately
4. Give the customer their existing key — they can activate on new devices

Transfer API

```
POST /api/licenses/{id}/transfer
curl -X POST https://api.permitcore.dev/api/licenses/{id}/transfer \
  -H "Authorization: Bearer <admin-token>"
```

No request body required. Response:

```
JSON
{
  "message": "License transferred: 3 activation(s) cleared."
}
```

> **Irreversible:** Transfer permanently deletes the activation records. If you need to keep a history of which devices were ever activated, use the Activity Log — the transfer action is logged there with the count of cleared activations.

Transfer vs. Reset Activations

	Transfer	Reset Activations
Purpose	Customer moves to new hardware	Admin maintenance / troubleshooting

	Transfer	Reset Activations
Effect	Clears activations, keeps key	Clears activations, keeps key
Audit log entry	Yes — "TransferLicense"	Yes — "ResetActivations"
Semantic difference	Customer-facing workflow	Admin-facing workflow

Offline Activation

Licensing Features

SDK 2 offline setup

Offline access needs a public key your application already trusts. A key stored beside a license token in an editable cache file cannot establish that trust. Earlier 1.x cache behavior has a confirmed trust weakness; use the migration below before relying on the SDK 2 offline cache. Use the language-specific installation command in the SDK references. Godot uses `trusted_public_keys`, an application-specific `cache_directory` and native ECDSA verification; see the Godot SDK reference.

1. Find your tenant slug in your application integration settings. While connected to your verified API origin, request `GET /api/v1/{tenantSlug}/public-key`. Take the returned `publicKey`: this is your tenant's ECDSA P-256 verification key.
2. Put that public key in trusted application configuration distributed with your app. Never ship the private signing key. Stripe keys, Paddle tokens and Admin JWT keys are different credentials and cannot verify these license tokens.
3. Configure the SDK's trusted public key list, an application-specific cache directory, and the same stable device ID on each launch. Pass the expected product ID when validating or activating. No configured trust keys means cached authorization is disabled; online license calls remain available.
4. For automatic grace, make a successful online validate/activate call with an offline-capable license. SDK 2 requires a fresh signed proof containing the license-key SHA-256 and checked product. Activation fallback and node-locked validation also need signed evidence of an existing activation for that device. Validation never consumes a seat.
5. For manual offline activation, generate a device token in Admin, then call the configured client's offline activation method. On later launches, call its offline validation method. Both verify against configured trust; every disk read verifies the original signed token.
6. Test a genuine token, changed token, wrong signer, missing trust keys, wrong license, wrong product, wrong device and expired deadline before shipping your app.

Migration changes: .NET offline activation/validation are now instance methods; Go uses client methods and the `/v2` module path; C++ persisted offline validation is now an instance method. Manual activation helpers no longer accept a public key argument. Raw signature verification still takes an explicit key and leaves key trust to the caller.

The new files contain only the original token and use separate `_v2_` filenames with a full SHA-256 suffix. Legacy cache files are ignored. Original signed manual tokens can still verify with a trusted key, but product-pinned use needs a new token containing `productId`. Multiple approved keys support deliberate rotation; removing a compromised key requires an application or configuration update.

When connected, the optional server token-status check needs no Admin session. It checks the token tenant's signing-key status and any tracked revocation. Valid tokens issued independently with your exported signing key also work, but untracked tokens have no individual server revocation record. SDK 2.1.1 supports this check under tenant row-level security without an Admin session.

Offline verification cannot learn about revocation, device removal or a later payment change until

reconnection. Previously issued proofs retain their signed deadline. Local clock manipulation and application patching remain outside this protection. Store payment-grace caps still apply. See the language-specific SDK references.

For Store subscriptions, newly generated offline tokens expire no later than paid-through plus the effective payment grace. Reconnect to obtain an updated signed token after renewal. Already-issued offline tokens retain their signed deadline while disconnected.

Offline activation lets customers activate a license key on a machine that has no internet access. Instead of calling the public API, the customer presents a **signed offline token** that was generated for their specific device by a Company Admin.

How it works

1. Customer generates their **hardware fingerprint** (HWID) and sends it to you via email / support portal.
2. Admin opens the license in PermitCore → scrolls to **Offline Activation Tokens** → clicks **Generate Token**.
3. Admin enters the Device ID, optional device name, and how long the token should be valid (1–365 days).
4. A signed token is shown *once* — admin copies and sends it to the customer.
5. Customer's application calls ActivateOffline(token) in your SDK. The SDK verifies the ECDSA signature, expiry, and device binding, then stores the token locally.
6. On every subsequent launch, the SDK verifies the cached token offline — no internet needed.

> The offline token is verified using your tenant's **ECDSA P-256 public key**. The public key is available at GET /api/v1/{tenantSlug}/public-key — the SDK uses it for verification; distribute it in trusted application configuration before disconnecting. Distribute the trusted public key to an air-gapped machine before activation. Offline verification works only until the token expires; the machine cannot discover revocations while disconnected.

Step-by-step in the Admin Panel

Obtain the public verification key from the public API and generate the device token in Admin. Prepare both before disconnecting the customer machine.

1 **Fetch your public verification key (once, or after rotation)**

Call GET /api/v1/{tenantSlug}/public-key and keep the returned publicKey and kid with your application or offline package. The public verification key can verify signatures, not create them. **Export Signing Key**, available on every plan including Free, returns private signing material as well; keep that vendor recovery package private and distribute only the public verification key to customers.

2 **Get the customer's Device ID**

The customer's application calls the SDK's hardware-fingerprint helper (e.g. GetHardwareId()) and sends you the resulting string — by email, support ticket, or an in-app "Copy Device ID" button you build. This is the value the token gets bound to.

3 **Generate the token**

Open the customer's license: **Licenses → [find the license] → View**. Scroll to **Offline Activation Tokens** and click **Generate Token**. Paste the Device ID from step 2, optionally name the device (e.g. "Warehouse PC 3"), and choose how many days the token stays valid (1–365).

4 **Copy the token — shown once**

The signed pc_offline_v1.... string is displayed exactly once. Copy it now and send it to the customer alongside the public key from step 1 (not the token alone — the customer's app needs both to verify).

5 **Customer activates offline**

The customer's application calls the SDK's offline-activate function with the token and your public key (see the SDK code below). From then on it verifies the cached token locally on every launch — see "What the SDK does" below.

> **"Invalid signature" almost always means the wrong public key.** Every downloadable SDK demo ships with a pre-filled *sample* public key and token pair purely to illustrate the fields — pasting a real token from your own tenant against that sample key will always fail signature verification, by design. Replace both fields with your own tenant's real public key (step 1 above) and a token generated for the actual device before testing.

Token format

Tokens have the format pc_offline_v1.<payload>.<signature>:

Part	What it contains
pc_offline_v1	Version prefix — validated first by the SDK
<payload>	Base64URL-encoded JSON: tokenId, licenseKeyHash, deviceId, productName, expiresAt, ...
<signature>	Base64URL-encoded ECDSA P-256/SHA-256 signature over the UTF-8 bytes of the base64url payload string

What the SDK does

1. Split on ., check prefix is pc_offline_v1
2. Use an independently configured tenant public key originally obtained from /api/v1/{slug}/public-key
3. Verify ECDSA signature — reject if invalid or tampered
4. Check expiresAt > now — reject if expired
5. Check deviceId matches local hardware fingerprint — reject if different machine
6. Store token on disk — verified offline on every future launch

> All nine official SDKs implement offline token verification natively — no manual crypto required in your app. The critical implementation detail every SDK matches exactly: the server signs with **ECDSA P-256 / SHA-256,**
> **IEEE P1363 raw signature format** (a 64-byte r‖s concatenation, not ASN.1 DER) over the UTF-8

> bytes of the base64url payload string.

C# (.NET SDK)

```
// SDK 2.1.1; configure trusted keys before disconnecting.
using var client = new PermitCoreClient(new PermitCoreOptions {
    BaseUrl = "https://your-domain.com",
    TrustedPublicKeys = new[] { trustedTenantPublicKey },
    CacheDirectory = appCacheDirectory,
    DeviceId = hwid
});
var activated = client.ActivateOffline(offlineToken, expectedProductId: productId);
var cached = client.ValidateOffline(expectedProductId: productId);
```

Python (Python SDK)

```
# pip install cryptography  (lazy-imported — base client stays dependency-free)
client = PermitCoreClient("https://your-domain.com", trusted_public_keys=[trusted_public_key],
    cache_directory=app_cache_directory, device_id=hwid)
result = client.activate_offline(offline_token, expected_product_id=product_id)

# On every subsequent launch — no internet needed
result = client.validate_offline(expected_product_id=product_id)
```

The following Node.js and PHP examples assume clients configured with trusted public keys, cache directory and a stable device ID as explained above.

Node.js / TypeScript

```
const result = client.activateOffline(offlineToken, hwid, productId);

// On every subsequent launch — no internet needed
const result = client.validateOffline(hwid, productId);
```

PHP

```
$result = $client->activateOffline($offlineToken, $hwid, $productId);

// On every subsequent launch — no internet needed
$result = $client->validateOffline($hwid, $productId);
```

Java

```
OfflineTokenResult result = client.activateOffline(offlineToken, hwid, productId);

// On every subsequent launch — no internet needed
OfflineTokenResult result = client.validateOffline(hwid, productId);
```

C++

```
auto result = client.activate_offline(offline_token, hwid, product_id);
```

```
// On every subsequent launch — no internet needed
auto result = client.validate_offline(hwid, product_id);
```

Revoking a token

To revoke an offline token, go to the license detail page → Offline Activation Tokens → click the **ban icon** next to the token. The token is immediately marked as revoked in the database.

Revoking is **soft enforcement**: the SDK can optionally call POST /api/v1/offline/verify to check revocation status when online. Strictly offline deployments won't see revocations until the token naturally expires. Use short expiresInDays values (e.g. 30–90 days) for tighter control.

> **Design trade-off:** Offline activation cannot be instantly revoked on an air-gapped machine — the token must expire. Set short validity periods (30–90 days) for sensitive deployments. For perpetual licenses, use annual renewal tokens.

Optional server-side verification

When the customer's machine has internet access, your SDK can also call the server to check for revocation:

Request: POST /api/v1/offline/verify

```
{
  "token": "pc_offline_v1.eyJ2ZXJzaW9uIjoxLCJ..."
}
```

Example response

```
{
  "isValid":     true,
  "tokenId":     "3fa85f64-5717-4562-b3fc-2c963f66afa6",
  "productName": "Acme Video Editor Pro",
  "deviceId":    "a3f8d2b1c9e4...",
  "expiresAt":   "2026-08-29T00:00:00Z",
  "message":     "Token valid."
}
```

Security properties

Property	How it's achieved
Tamper-proof	ECDSA P-256 signature — any modification invalidates the signature
Device-bound	Token payload includes deviceId — SDK rejects tokens for other machines
Time-limited	expiresAt checked by both SDK and server verify

Property	How it's achieved
	endpoint
Revocable	TokenId stored server-side — checked via /offline/verify when online
Public key distribution	Public key is per-tenant, fetched once from /api/v1/{slug}/public-key

Node-locked / HWID

Licensing Features

Node-locked / HWID Licensing

Node-locked licensing ties a license key to a specific device (hardware fingerprint). This is useful for single-machine perpetual licenses where you want to prevent one key from being shared across multiple computers.

How it works

Enable **Node Locked** when creating a license (or update it after creation). Then in your application:

1. Generate a **device fingerprint** — a stable SHA-256 hex string identifying this install. Prefer your SDK's built-in helper (see below) over hashing raw hardware identifiers.
2. Pass it as `deviceId` when calling `POST /api/v1/activate` — or just omit it if you're using an SDK, and it's collected automatically
3. Optionally pass a human-readable `deviceName` (e.g. "John-MacBook-Pro")

Node-locked activation

```
{
  "licenseKey":       "PERMIT-A3K7-MN2P-QR9X-TZ5W",
  "deviceId":         "a3f8d2b1c9e4...",
  "deviceName":       "John-MacBook-Pro-M3",
  "expectedProductId": "7f3e9a2b-1c4d-4e8f-9a6b-3d2c1e0f5a8b"
}
```

Raw HTTP activation also requires a fresh `nonce` from `GET /api/v1/nonce`. The snippets here focus on device/user fields; use the complete activation flow when sending a request.

Re-installation is free

When a license is node-locked, re-activating from the **same device ID** does *not* consume an additional activation slot. The server recognises the device and counts it as a re-activation rather than a new registration. This means customers can reinstall their OS without using up activations — as long as the hardware fingerprint stays the same.

Choosing a device fingerprint

PermitCore is fingerprint-agnostic — it accepts any stable SHA-256 hex string as `deviceId` and never generates or interprets it. **Use your SDK's built-in helper first** — the .NET, Python, Node.js, and PHP SDKs ship a `GetHardwareId()` / `get_hardware_id()` helper that's called automatically whenever `deviceId` is omitted from `activate()`/ `checkout()`. It deliberately avoids hashing raw CPU serial, disk serial, or MAC address — those are unstable (cloned on VM snapshots, changed on hardware upgrades, unavailable in containers) and needlessly sensitive to collect. Instead it combines a few low-volatility attributes (hostname, OS, architecture) with a random seed persisted to a local file on first run.

SDK	Helper	Persisted seed location
.NET	PermitCoreClient.GetHardwareId()	%APPDATA%\.permitcore_seed
Python	PermitCoreClient.get_hardware_id()	~/.permitcore_seed
Node.js	PermitCoreClient.getHardwareId()	~/.permitcore_seed
PHP	PermitCoreClient::getHardwareId()	system temp dir (weaker isolation — avoid relying on it on shared hosting)
Java / C++	*none yet*	pass your own deviceId

> **Why not hash real hardware identifiers?** A fingerprint only needs to be stable for one install, not cryptographically tied to physical hardware — this isn't a DRM/anti-tamper mechanism, it's a seat-counting key. Raw CPU/disk/MAC hashing breaks on the cases that matter most in practice (VM re-provisioning, cloud instances, hardware swaps), and since re-activating from a *changed* fingerprint consumes a new seat on a node-locked license, an unstable ID directly costs your customer activations. If you're on Java, C++, or a custom integration without a built-in helper, prefer an **app-generated installation ID persisted to your own config file** over hashing hardware attributes.

Persist the installation identity

Use your SDK helper where available. For custom clients, generate an installation identifier once, persist it in the application's protected configuration, and send the same device ID on future activations. Reinstalling or deleting its seed may change that ID. Do not derive a supposedly stable identity from whichever network adapter happens to be returned first.

Node-locked vs. count-based — choosing the right mode

Scenario	Recommended mode
Desktop app, single-machine perpetual license	Node-locked (Max Activations = 1–2)
Desktop app, allow use on 2–3 personal devices	Count-based (Max Activations = 2–3) or Node-locked (Max Activations = 2–3)
SaaS, web app, or mobile app	Count-based — device ID is not meaningful
CI/CD or server software (multiple machines)	Count-based with a higher activation limit
Floating / concurrent license	Use the Floating License type — HWID not applicable

Managing device activations in the admin panel

1. Go to **Licenses** and click **View** on a node-locked license
2. Scroll to the **Active Devices** table — each row shows Device ID, Device Name (if provided), and activation timestamp

3. Click the **Deactivate** button next to any row to remove that machine's activation
4. The activation counter decrements immediately, freeing up a slot for the customer to use on a new machine

> Deactivating a device from the admin panel does *not* automatically invalidate tokens or local licenses on that machine. If you need the software to stop working immediately, also revoke the license key entirely, or rely on the online heartbeat in your SDK.

Deactivation API (programmatic)

Your application can let customers self-deactivate through the anonymous customer portal API (e.g. a "Deactivate this machine" button in Settings). First look up the license to get the `activationId` of the device to remove, then deactivate it by that ID:

POST /api/v1/portal/{tenantSlug}/lookup — find the activationId

```
// Request body
{
  "licenseKey": "PERMIT-A3K7-MN2P-QR9X-TZ5W"
}

// Response 200 OK — activations[].id is what you need below
{
  "productName": "Task Manager Pro",
  "activations": [
    { "id": "3b4c9f2a-1e7d-4a8f-9c2b-...", "deviceName": "John-MacBook-Pro-M3", "activatedAt": "2026-06-01T00:00:00Z"
}
  ]
}
```

POST /api/v1/portal/{tenantSlug}/deactivate — remove that activation

```
// Request body
{
  "licenseKey":   "PERMIT-A3K7-MN2P-QR9X-TZ5W",
  "activationId": "3b4c9f2a-1e7d-4a8f-9c2b-..."
}

// Response: 204 No Content on success (empty body)
```

Both endpoints are unauthenticated (identified by the license key itself) and always take the key/ID in the POST body — never the URL — to keep them out of access logs. After deactivation the customer can activate the key on a new machine; the activation counter is decremented server-side.

> If you've enabled the **"Key + email code"** or **"Account required"** portal security level (see Security — three levels), this exact key-only flow stops working at that level and your integration needs to change: at "Key + email code", call `POST /api/v1/portal/{tenantSlug}/deactivate/request-otp` first and pass the returned `otpToken` plus the code the customer enters as `otpCode` on the deactivate call above; at "Account required" this endpoint returns `403` outright and you'll need the authenticated `/account/licenses/{licenseId}/deactivate`

> endpoint instead. Most vendors stay on the default and never need this.

Floating Licenses

Licensing Features

Floating / Concurrent Licenses

Floating licenses (also called *concurrent licenses*) let multiple users share a pool of seats. Instead of permanently consuming an activation slot, a user **checks out** a seat when the app starts and **returns** it when they close it.

Example: a company has 50 employees but only 10 ever use the software at the same time. You sell them a license with 10 concurrent seats — far cheaper than 50 individual activations.

> Available on **Starter** (up to 5 concurrent seats per license) and without a plan-level seat cap on **Professional** and **Enterprise**. A license can still have its own seat cap. Not available on Free.

Creating a floating license — toggle the Floating mode and set the maximum concurrent sessions.

New to this model? Start with the floating license guide and shared-seat example.

For cloned images and VDI, read the VM licensing guide. Checkout renews an existing session for the same license and device ID; distinct intended concurrent consumers need distinct provisioned identities. Copying an ID or session token is not a new independent seat.

How floating sessions work

Action	API request	What happens
App starts	POST /api/v1/float/checkout	A session is created. If max concurrent seats are in use, returns an error — app should show "all seats in use".
Before expiresAt, normally every 4 min	POST /api/v1/float/heartbeat (body: {"sessionToken":"..."})	Checks the current license and floating plan. Grants up to 15 minutes, capped at the license access deadline including applicable grace. Denied renewal grants no new authorization.
App closes	POST /api/v1/float/checkin (body: {"sessionToken":"..."})	HTTP 204 confirms the seat was returned. A lost response does not confirm release.

> **Use the returned expiresAt deadline, not a fixed 15-minute timer.** Schedule heartbeats in your application, normally every four minutes, and handle each result. A grant lasts up to 15 minutes and can end earlier with the license's access deadline. Applicable post-expiry/Store payment grace is preserved; revoked licenses are denied and an active snapshot alone does not extend an expired paid-through period. Stop protected work

> on a denied renewal or missed deadline. Offline validation does not renew a floating seat.

Creating a floating license

1. Go to **Licenses → Create**
2. Toggle **"Floating / Concurrent License"**
3. Set **Max Concurrent Sessions** (blank = no license-specific cap; the plan's cap still applies)
4. Generate and distribute the key as usual

SDK lifecycle: checkout, renewal and awaited return

All nine SDKs expose these operations. This .NET helper uses the actual 2.1.1 methods, cancels protected work when authority cannot be renewed, and awaits work stopping before returning the seat. Avoid async-void timers and ProcessExit handlers: the process does not await them.

This helper manages a floating lease. Before calling it, validate the key for your application using `expectedProductId` and provide the application version when using version rules; see license validation. Checkout itself does not carry those validation fields.

C# / .NET SDK 2.1.1

```
using System;
using System.Threading;
using System.Threading.Tasks;
using PermitCore;

namespace PermitCore.Examples;

public static class FloatingLeaseLifecycle
{
    public static async Task RunAsync(PermitCoreClient client, string licenseKey,
        Func<CancellationToken, Task> protectedWork, CancellationToken stop = default,
        TimeSpan? heartbeatInterval = null)
    {
        var interval = heartbeatInterval ?? TimeSpan.FromMinutes(4);
        if (interval <= TimeSpan.Zero || interval > TimeSpan.FromMinutes(4))
            throw new ArgumentOutOfRangeException(nameof(heartbeatInterval));

        var seat = await client.CheckoutAsync(licenseKey, ct: stop);
        if (!seat.Success || string.IsNullOrWhiteSpace(seat.SessionToken))
            throw new InvalidOperationException("No floating seat was granted.");

        var token = seat.SessionToken;
        using var lease = CancellationTokenSource.CreateLinkedTokenSource(stop);
        Task? work = null;
        try
        {
            var remaining = Remaining(seat);
            lease.CancelAfter(remaining);
            work = protectedWork(lease.Token);
            while (!work.IsCompleted)
            {
```

```
            var pause = Task.Delay(remaining / 2 < interval ? remaining / 2 : interval, lease.Token);
            if (await Task.WhenAny(work, pause) == work) break;
            await pause;
            var renewed = await client.HeartbeatAsync(token, lease.Token);
            remaining = Remaining(renewed);
            lease.Token.ThrowIfCancellationRequested();
            lease.CancelAfter(remaining);
        }
        await work;
    }
    finally
    {
        lease.Cancel();
        try
        {
            if (work is not null) await work;
        }
        catch (OperationCanceledException) when (lease.IsCancellationRequested) { }
        finally
        {
            using var cleanup = new CancellationTokenSource(TimeSpan.FromSeconds(5));
            if (!await client.TryCheckinAsync(token, cleanup.Token))
                Console.Error.WriteLine("Seat return unconfirmed; the last granted deadline still applies.");
        }
    }
}

private static TimeSpan Remaining(FloatingSession session)
{
    if (!session.Success || !session.ExpiresAt.HasValue)
        throw new InvalidOperationException("Floating access could not be renewed.");
    var remaining = session.ExpiresAt.Value.ToUniversalTime() - DateTime.UtcNow;
    if (remaining <= TimeSpan.Zero)
        throw new InvalidOperationException("The floating session deadline has passed.");
    return remaining;
}
}
```

Pass your application's work as `protectedWork` and use its cancellation token in every protected operation. Cancellation is cooperative: the helper cannot stop code that ignores that token. It waits for work to stop before returning the seat. A failed checkin is reported as unconfirmed.

Python uses `checkout`, `heartbeat` and `try_checkin`. Apply the same success/deadline and cooperative-cancellation rules; see the Python reference for its API and configuration. Keep session tokens out of URLs and logs.

Viewing active sessions

You can see all active sessions for a license via the API: `POST /api/v1/float/sessions` (body: `{"licenseKey":"..."}`, requires authenticated access to the owning tenant — a license key is a sensitive credential). Expired sessions no longer authorize use. Eligible checkout and session-list queries remove expired rows.

License Bundles

Licensing Features

A bundle groups multiple products under a **single license key**. When a customer activates their bundle key, the API response includes every product they're entitled to — without issuing separate keys for each product.

> **Use case:** You sell "Acme Suite" — a bundle of Acme Video + Acme Audio + Acme Export. Instead of sending 3 separate keys, the customer gets one bundle key. Your app checks the `bundleProducts` array to know which modules to unlock.

Creating a bundle

1. Go to **Bundles** in the sidebar.
2. Click **New Bundle**, give it a name, and select at least 2 products.
3. Save. The bundle now appears as a selectable product type when creating license keys.

Validate / Activate response

```JSON
{
  "isValid": true,
  "productName": "Acme Suite",
  "productId": "7f3e9a2b-1c4d-4e8f-9a6b-3d2c1e0f5a8b",
  "bundleProducts": [
    { "productId": "...", "productName": "Acme Video" },
    { "productId": "...", "productName": "Acme Audio" },
    { "productId": "...", "productName": "Acme Export" }
  ]
}
```

`productId` is the key's own primary product; `bundleProducts` is a list of every product the bundle key unlocks (each an object with `productId` + `productName`, not plain strings). Prefer stable `productId` values for entitlement checks; names can change.

SDK pattern

```C#
var result = await client.ValidateAsync(licenseKey);
if (result.IsValid)
{
    var products = result.BundleProducts?.Select(p => p.ProductName) ?? Enumerable.Empty<string>();
    if (products.Contains("Acme Video")) EnableVideoModule();
    if (products.Contains("Acme Audio")) EnableAudioModule();
}
```

Constraints

- A bundle requires at least **2 products**.
- Deleting a bundle is blocked if active license keys exist for it.
- Bundle keys can be sold via the License Store — customers see the bundle as a single product with all included items listed.

Feature Flags

Licensing Features

Feature flags let you control **which parts of your software are unlocked** for a given license — without issuing separate keys per edition. Instead of "License A = Basic" and "License B = Pro", you attach a list of feature strings to each license and check them at runtime.

> **Example:** Your app has three tiers — Basic, Pro, Enterprise. Instead of three separate products, you use one product and set features to ["export"], ["export","api"], or ["export","api","sso","white-label"] per license. Your code checks the list on every call.

Feature flags configured in the Edit Metadata modal — comma-separated strings returned in every validate/activate response.

Setting feature flags

1. Open **Licenses → View → Edit Metadata**
2. Enter comma-separated feature names in the **Feature Flags** field (e.g. export, api, sso)
3. Save — changes take effect immediately on the next API call

Use consistent lowercase feature names. Raw string comparisons are case-sensitive; some SDK helper methods compare case-insensitively. Grant access only after a valid license decision.

Reading feature flags in your app

JavaScript

```
const res = await fetch('/api/v1/validate', {
  method: 'POST', headers: { 'Content-Type': 'application/json' },
  body: JSON.stringify({
    licenseKey,
    expectedProductId: '7f3e9a2b-1c4d-4e8f-9a6b-3d2c1e0f5a8b', // optional
  }),
});
const data = await res.json();

if (!data.isValid) { /* reject */ }

const features = new Set(data.features ?? []);

if (features.has('export')) { enableExport(); }
if (features.has('api'))    { enableApiAccess(); }
if (features.has('sso'))    { enableSso(); }
```

Python

```
data = requests.post('/api/v1/validate', json={
    'licenseKey': license_key,
    'expectedProductId': '7f3e9a2b-1c4d-4e8f-9a6b-3d2c1e0f5a8b',  # optional
}).json()
```

```
features = set(data.get('features') or [])

if 'export' in features:
    enable_export()
if 'sso' in features:
    enable_sso()
```

```
C#
var httpResp = await http.PostAsJsonAsync("/api/v1/validate", new {
    licenseKey = key,
    expectedProductId = "7f3e9a2b-1c4d-4e8f-9a6b-3d2c1e0f5a8b" // optional
});
var resp = await httpResp.Content.ReadFromJsonAsync<LicenseResult>();
var features = resp?.Features?.ToHashSet() ?? [];

if (features.Contains("export")) EnableExport();
if (features.Contains("api"))    EnableApiAccess();
```

Common patterns

Pattern	Feature strings	Notes
Tiered editions	basic, pro, enterprise	Check for the highest tier present
Module unlocks	export, api, sso, reports	Each feature is independent
Seat count in features	seats:10, seats:50	Parse the value after the colon
Expiring beta access	beta	Combine with ExpiresAt for time-limited beta

Version Management

Licensing Features

Version management lets you enforce software build constraints per license. Set a **Min Version** and/or **Max Version** on any license key — customers running builds outside the allowed range are blocked when they call validate or activate.

> **Use cases:** Block builds with known security vulnerabilities (set MinVersion), prevent customers on old support contracts from upgrading past a major version (set MaxVersion), or pin enterprise customers to a specific release while you stabilise the next major version.

Setting version constraints

At creation time — fill in **Min Version** and/or **Max Version** in the **Licenses → New License** form. Both are optional; leave blank for no restriction.

On existing licenses — go to **Licenses → View → Edit Metadata**. Changes take effect immediately on the next API call from the customer's software.

Version range examples

MinVersion	MaxVersion	Effect
2.0.0	—	Blocks all builds below 2.0.0
—	3.9.9	Blocks all builds above 3.9.9
2.0.0	3.9.9	Only 2.x and 3.x builds allowed
3.0.0	3.0.0	Pins to exactly version 3.0.0
—	—	No restriction — any version passes

Versions are parsed as MAJOR.MINOR.PATCH and compared numerically component-by-component (not as a lexicographic string compare), so 2.10.0 > 2.9.0 orders correctly. Any pre-release/build suffix (-beta1, +build5) is stripped before comparison.

How to send the version from your app

Pass the current software version in the JSON request body on both validate and activate:

curl — validate with version

```
curl -X POST https://api.permitcore.dev/api/v1/validate \
  -H "Content-Type: application/json" \
  -d '{"licenseKey":"PERMIT-XXXX-XXXX-XXXX-
XXXX","version":"2.3.1","expectedProductId":"7f3e9a2b-1c4d-4e8f-9a6b-3d2c1e0f5a8b"}'
```

curl — activate with version

```
curl -X POST https://api.permitcore.dev/api/v1/activate \
  -H "Content-Type: application/json" \
  -d '{"licenseKey":"PERMIT-XXXX-XXXX-XXXX-XXXX","version":"2.3.1","nonce":"<from GET /api/v1/
nonce>","expectedProductId":"7f3e9a2b-1c4d-4e8f-9a6b-3d2c1e0f5a8b"}'
```

JavaScript

```
const res = await fetch('/api/v1/validate', {
  method: 'POST', headers: { 'Content-Type': 'application/json' },
  body: JSON.stringify({
    licenseKey: key,
    version: '2.3.1',
    // Optional — reject a key issued for a different product instead of silently accepting it
    expectedProductId: '7f3e9a2b-1c4d-4e8f-9a6b-3d2c1e0f5a8b',
  }),
});
const data = await res.json();

if (!data.isValid && data.minVersion) {
  showError(`Please upgrade to v${data.minVersion} or newer.`);
} else if (data.errorCode === 'WrongProduct') {
  showError(`This key is for a different product: ${data.productId}`);
}
```

Blocked response

JSON — version blocked

```
{
  "isValid":    false,
  "message":    "Version 1.9.0 is below the minimum required version 2.0.0",
  "productId":  "7f3e9a2b-1c4d-4e8f-9a6b-3d2c1e0f5a8b"
}
```

Note: the version-blocked response doesn't populate minVersion/maxVersion itself (they come back null) — parse the required version out of message, or call validate() without a version first to read the license's bounds directly.

If the customer sends **no version at all**, version constraints are skipped — the call proceeds as if there is no restriction. This ensures backward compatibility with apps built before you added version checks.

Version tracking in activations

The version string is stored on every Activation record. This lets you see which software builds your customers are running in the activation history and analytics — useful for planning deprecation timelines or identifying customers who need to upgrade.

Offline Grace Period

Licensing Features

The offline grace period lets your software continue working even when it cannot reach the PermitCore server — for example, when the customer has no internet, or your server is temporarily down. Set a number of days, and your SDK should cache the last successful validation response and treat it as valid for that duration.

Setting the grace period

1. Go to **Licenses → View → Edit Metadata**
2. Enter a number in the **Offline Grace Days** field (e.g. 7)
3. Save — the value is returned in every validate/activate response as offlineGraceDays

Use signed offline grace

When offline grace is enabled, successful responses can include an ECDSA-signed offlineCacheToken with the pc_grace_v1 prefix. Official SDKs cache and verify this token when the service cannot be reached. Configure the SDK and its required cryptography dependencies; do not cache a bare JSON success and trust it later.

A custom implementation must validate token structure, signature, trusted public key, expiry, and matching license/device binding where applicable. Protect public-key distribution as part of your app's trust model. A public key loaded from the same untrusted cache as a token is not independently trustworthy. Do not use offline grace to override an explicit online rejection.

Use admin-issued offline activation for machines that cannot obtain an initial online validation.

Recommendations

Scenario	Suggested grace period
Desktop app (typical internet user)	7 days — covers a holiday or travel period
Server-side / always-online software	0 or 1 day — connectivity expected
Air-gapped / industrial software	Use offline activation tokens instead
Mobile app	3–7 days

> **Security note:** The cached token is cryptographically signed — forging a fake "valid" entry from nothing now requires PermitCore's private signing key, not just local file-write access. What this still doesn't and can't prevent: a user manipulating their system clock to keep an already-issued token looking unexpired, or copying a genuinely-issued cache file to a second machine (unless your app also checks the payload's device binding). **Offline grace is availability protection for legitimate customers going offline, not a DRM/anti-tamper mechanism.** For high-security licensing, use shorter grace periods (1–3 days) and consider offline activation tokens (individually revocable, admin-issued) for genuinely air-gapped environments.

Post-Expiry Grace Period

Licensing Features

A post-expiry grace period lets a license remain valid for a configurable number of days *after* its ExpiresAt date. This is useful for customers who are in the process of renewing — it prevents a hard cutoff at the exact expiry moment.

> **Different from Offline Grace Days.** *Offline Grace Days* covers the case where the customer has no internet and can't reach your server. *Post-Expiry Grace Days* covers the case where the server is reachable but the license has just passed its expiry date.

How it works

1. Set PostExpiryGraceDays on a license at creation time (or via Edit License).
2. After ExpiresAt, POST /api/v1/validate and POST /api/v1/activate still return isValid: true.
3. The response includes "isInGracePeriod": true so your app can show a renewal reminder.
4. Once the grace period is exhausted, the key behaves as expired (isValid: false).

Validate response during grace period

POST /api/v1/validate { "licenseKey": "PERMIT-XXXX-XXXX-XXXX-XXXX", "expectedProductId": "7f3e9a2b-1c4d-4e8f-9a6b-3d2c1e0f5a8b" } → { "isValid": true, "isInGracePeriod": true, "expiresAt": "2026-05-01T00:00:00Z", "message": "License is valid", "productId": "7f3e9a2b-1c4d-4e8f-9a6b-3d2c1e0f5a8b" }

Setting grace days

Enter a value (e.g. 7) in the **Post-Expiry Grace Days** field when creating or editing a license. For Store subscriptions, blank inherits Store payment grace (14 days by default), applied while past_due or unpaid; zero disables that grace. Other licenses have no post-expiry grace when the field is blank.

Maintenance / Annual Support

Paid feature maintenance and security support need separate policies. The CRA planning guide explains the distinction and product limits; licensing controls do not distribute patches or establish compliance.

Licensing Features

Maintenance / Annual Support Period

Read the perpetual fallback planning guide. Maintenance fields do not automatically calculate a last eligible release or convert a cancelled Store subscription to perpetual access.

Maintenance expiry covers the classic desktop ISV model: a customer buys a **perpetual license** (activates forever) but receives paid feature upgrades and commercial support under the annual contract. Assess security-support obligations separately; maintenance expiry must not become a blanket denial of applicable security fixes.

How it works

Set MaintenanceExpiresAt on a license. The validation response always includes this date. Your SDK or application logic decides what to do with it — typically blocking upgrades to versions released after the maintenance expiry date.

Validate response

POST /api/v1/validate { "licenseKey": "PERMIT-XXXX-XXXX-XXXX-XXXX", "expectedProductId": "7f3e9a2b-1c4d-4e8f-9a6b-3d2c1e0f5a8b" } → { "isValid": true, "maintenanceExpiresAt": "2027-01-15T00:00:00Z", "message": "License is valid", "productId": "7f3e9a2b-1c4d-4e8f-9a6b-3d2c1e0f5a8b" }

Recommended SDK pattern

var result = await client.ValidateAsync(key, expectedProductId: Guid.Parse("7f3e9a2b-1c4d-4e8f-9a6b-3d2c1e0f5a8b")); if (result.IsValid && result.MaintenanceExpiresAt.HasValue && result.MaintenanceExpiresAt.Value < DateTime.UtcNow) { // Block upgrade — maintenance expired. Allow running current version. ShowMessage("Your maintenance plan has expired. Upgrade to re-enable updates."); }

> PermitCore does not enforce the maintenance date itself — it only surfaces it in the API response. Your application is responsible for deciding which software versions require an active maintenance contract. This gives you full flexibility over your release policy.

Setting maintenance expiry

Pick a date in the **Maintenance Expires At** field in the Create License form. For perpetual licenses, leave ExpiresAt blank and only set MaintenanceExpiresAt.

Overview

Before you open your store

The License Store sells keys using your selected payment account. Create an internal product first, then configure payment credentials, the license issued after checkout, and the details shown to buyers. See the direct game-sales guide.

Work through a test purchase before sharing the store. Confirm that checkout, payment webhooks, license delivery, invoice details, and cancellation behavior all match your sales model. Use matching sandbox/test credentials throughout that test.

For your own PermitCore subscription and account exports, see Account & team. For license lookup and device self-service, see the Customer portal.

License Store

Licensing Features

To pause your storefront, turn off **Store is public** and click **Save Settings**. The switch alone does not publish the change. A Store listing with existing orders cannot be deleted; deactivate it instead to preserve history.

The License Store lets your customers **buy licenses directly** from a branded public storefront — no coding required on your end. A customer account is required to check out (see "Customer Accounts" below); PermitCore hosts the checkout page and automatically creates a license key when payment succeeds, emailing the buyer a link to sign in and view it from their account rather than the key itself. Connect your own Stripe, Paddle or Lemon Squeezy account for the whole Store. **PermitCore takes 0% sales commission**; provider fees and payout terms apply.

A Store 2.0 storefront at /store/{yourSlug} with vendor navigation, hero slides, editorial collections, search, and responsive layouts.

How it works

1. **Connect your payment account**
   Go to **License Store → Settings**, select one provider and configure its matching live or test account and signed webhooks. Follow the provider setup instructions above. Stripe settles to your Stripe account; Paddle pays out under your merchant-of-record agreement.
2. **Create store products**
   Each store product links to one of your internal products. You set the display name, tagline, description, feature bullet list, image, pricing (one-time, monthly, or annual), and which license settings to create on purchase.
3. **Prepare your provider catalog**
   Use Stripe auto-setup or the Paddle catalog action for the selected provider, then review the product and price mappings.
4. **Share your store URL**
   Your store is live at https://permitcore.dev/store/yourSlug. Add it to your website, email footer, or documentation.
5. **Verified payment → license issued**
   The customer signs in or creates an account, then pays through the Store's selected provider. PermitCore verifies payment through the signed webhook, issues each purchased license and queues an email linking to the buyer's account. Payment may finish before fulfillment; the account shows the order after processing.

Complete setup tutorial: Stripe example

This example uses Stripe. For Paddle, follow the provider setup steps instead of the Stripe credential, catalog and webhook steps below. Product license settings, Store design and buyer account verification apply to every provider. Test the selected provider's payment, invoice, renewal, refund and failure paths before opening the Store.

1. Prepare the vendor account and Stripe

1. Create the software product that the purchased license will belong to.
2. Open **License Store → Settings** and enable the Store.
3. Add Stripe publishable, secret, and webhook signing keys from the same Stripe mode: all test or all live.
4. Add the vendor legal name, VAT number, default tax rate, and any country overrides used on invoices.
5. Create the webhook shown below and enable every required event before accepting an order.

> **Do not mix Stripe modes.** A test secret key with a live webhook or live Price ID cannot produce a valid end-to-end purchase. Complete one test-mode checkout first, then replace the entire credential set when going live.

2. Create the product customers will buy

1. Open **License Store → Products → Add Product** and link the correct internal product.
2. Write the public name, short tagline, card description, long product-page description, and up to six feature bullets.
3. Upload a clear landscape cover image and, where useful, a product gallery. Every active product should have an image before launch.
4. Choose one-time, monthly, and/or annual pricing and select the currency.
5. Configure the generated license: activation count, node locking, expiry, trial days, and feature flags.
6. Add an optional badge, CTA label, featured state, bundle contents, or free trial.
7. Save, then use **Auto-Setup Stripe Products** in Store → Settings → Products & prices (with Stripe selected) to create missing Stripe Products and Prices.

The resulting product page combines a gallery, description, features, pricing choices, trial, discount disclosure, cart, and secure checkout.

3. Build the visual identity in Shop Designer

1. Open **License Store → Shop Designer**.
2. In **Branding**, set the store name, logo, primary/accent colors, text colors, page background color, and optional page background image.
3. In **Visual Style**, choose the heading font, card density, hero height, corner style, and desktop product columns.
4. Configure the default category-tile color or image. Then edit each visible tile—such as Featured, Special offers, Free trials, Bundles, or a product badge—to override its own background color or image. Select **Use default** to remove an override.
5. Add hero slides with desktop-safe images, headline, supporting copy, CTA label, and a safe internal or HTTPS destination. Put your main campaign first: its image receives high loading priority; later slides use lazy/low-priority hints. Use appropriately sized, compressed images and check mobile cropping. Your image URLs and automatic slide controls are preserved; browsers

decide when to honor loading hints.

6. Use the live preview at desktop, tablet, and phone widths before saving.

4. Compose the homepage

Under **Homepage Sections**, add the blocks that tell the product story. The section list is the published page order.

1. Add a Featured spotlight for the strongest product story.
2. Add horizontal shelves for offers, trials, bundles, badges, or products under a selected price.
3. Add Category tiles for fast discovery and Discovery tabs for compact comparison.
4. Keep a Product catalog so every active product remains searchable and sortable.
5. Use the up/down arrows to reorder sections and the Visible switch to prepare a block without publishing it.
6. Set **Space above** and **Space below** independently from 0–160 px. New sections start with a balanced 32 px top gap; reduce a gap when two related blocks should read as one group.

Collections, individually branded category tiles, and discovery tabs make a larger catalog easy to scan.

5. Configure offers without weakening checkout

1. Open a Store Product and create a discount campaign with a name, percentage, label color, UTC start, and UTC end.
2. Leave **Show lowest price in the previous 30 days** enabled when the disclosure is useful or required for the product and market.
3. Preview the crossed-out regular price, colored percentage badge, campaign label, effective price, and 30-day comparison text.
4. Test at least one cart and Stripe test checkout. The API—not browser state—selects the currently valid campaign and calculates the effective price.

6. Run the launch test

- Check the homepage, product page, cart, sign-in, account, invoices, and license view on desktop and a real mobile viewport.
- Buy one one-time item, start one trial, and test a recurring product if the store offers subscriptions.
- Confirm the Stripe webhook creates exactly one order and license, even if Stripe retries the event.
- Verify the issued license has the intended activation and feature settings.
- Cancel a test subscription and confirm the linked license follows the configured lifecycle.
- Review every product image, CTA, currency, tax rule, vendor invoice detail, and external link before switching to live Stripe credentials.

The Create Store Product form before entry, with display fields and a live preview. Pricing and generated-license settings continue below.

A new Store before setup: Products, Orders, Settings, and Shop Designer. Add a product and configure

your selected payment provider before accepting purchases.

> **Auto-Setup Stripe Products** After creating your store products, click **Auto-Setup Stripe Products** in **Store → Settings → Products & prices** with Stripe selected. PermitCore automatically creates the matching Stripe Products and Prices in your Stripe account and stores the Price IDs. You never need to copy-paste price_xxx IDs manually. Re-running the setup is safe — prices that haven't changed are skipped.

Store product settings

Setting	Description
Name & Slug	Display name and URL identifier (e.g. /store/yourco/pro-edition)
Tagline & Description	Short marketing text shown on the product card
Long Description	Full marketing copy shown on the product detail page
Feature Bullets	Up to 6 bullet points shown on the card and detail page (e.g. "Unlimited exports")
Pricing	One-time, monthly, and/or annual prices. Set at least one. Currency selectable.
Stripe Price IDs	Auto-filled by "Auto-Setup Stripe Products" button, or enter manually for recurring subscriptions managed directly in Stripe.
Badge	Optional label on the card (e.g. "Most Popular") with custom color
Generated license	Max activations, node locking and feature flags apply to purchased keys. One-time purchases use configured expiry days; subscriptions use the paid provider billing period. Free trial claims use trial days.
Featured	Featured products are shown larger at the top of the store grid
Discount campaign	Schedule a 1–90% discount with start/end time, campaign name and color. The server applies the effective price at checkout.
Lowest price disclosure	Optional checkbox, enabled by default, showing the lowest applied price during the 30 days before the campaign started. Price history is recorded even when the label is hidden.

Shop Designer reference

Store → Shop Designer is organized as a visual page builder. Branding controls the shared visual

system and full-page background; Visual Style controls typography, hero height, card density and desktop columns; Category Tiles adds per-tile artwork; Homepage Sections controls page order, content, visibility, and spacing.

Shop Designer and the static banner are included from Starter. To set the static banner, use Store → Settings → Banner Image URL. Editing and showing the Hero Slider requires Professional or Enterprise. Store Custom CSS applies on Enterprise. Saved slides and CSS become available again when the matching plan is active.

Section	Use it for	Available controls
Featured spotlight	One large editorial product story with a selectable rail.	Title, subtitle, product filter, maximum items, visibility and order.
Horizontal shelf	Fast, touch-friendly browsing similar to a marketplace collection.	All/featured/trial/bundle/badge/ under-price filter and maximum items.
Category tiles	Shortcuts generated from featured items, offers, trials, bundles and product badges.	Title, subtitle, visibility, order, shared fallback style, and a color/image override for each visible tile.
Discovery tabs	Compact lists for popular products, offers, trials and bundles.	Title, subtitle, maximum items, visibility and order.
Product catalog	The searchable and sortable card grid.	Title, visibility, order and global products-per-row setting.

Every section also has independent **Space above** and **Space below** controls. Use the arrow controls to reorder sections and the Visible switch to prepare a section without publishing it. If the catalog is removed, PermitCore still adds a safe catalog fallback so products never become impossible to browse.

Discounts and the EU 30-day rule

PermitCore stores regular and promotional price events and calculates the comparison period from the 30 days immediately before a campaign starts. The promotional price itself is excluded, so it cannot become its own “lowest price.” Display and checkout use the same percentage and UTC schedule, but checkout always recalculates on the server.

> Article 6a of the EU Price Indication Directive applies to goods. The European Commission guidance explicitly says it does not apply to services, including digital services and digital content. Software-license stores therefore get a configurable disclosure rather than automatic country enforcement; it defaults on for transparency. General rules against misleading promotions and any stricter local law still apply. See the official Commission guidance and obtain legal advice for your own product classification.

Store security and performance

- **Server-authoritative checkout:** browser cart prices are display-only. The API reloads the active product, price type, campaign and tax before creating the selected provider checkout.

- **Payment boundary:** card details are entered with the selected payment provider. PermitCore stores encrypted vendor credentials and issues a paid license only after verifying the tenant-specific provider webhook and canonical payment.
- **Tenant isolation:** store products, customers, orders, designs and price history are protected by tenant filters and PostgreSQL Row-Level Security.
- **Safe customization:** colors, image URLs and slide links are validated; legacy rows are sanitized again on public read; Enterprise CSS blocks network loads and executable/breakout constructs.
- **Traffic resilience:** public catalog/product responses use shared Valkey caching and dedicated browse rate limits; checkout and trial endpoints use stricter limits. Cache failure falls back to PostgreSQL.
- **Responsive storefront:** grids collapse for tablets/phones, shelves remain horizontally scrollable, controls are touch-sized and reduced-motion preferences are respected.

Stripe webhook setup

In your Stripe Dashboard, create a webhook endpoint pointing to:

```
Webhook URL
https://api.permitcore.dev/api/store/webhook/yourSlug
```

Enable these events: checkout.session.completed, payment_intent.payment_failed, checkout.session.expired, invoice.paid, invoice.payment_failed, customer.subscription.updated, customer.subscription.deleted, charge.refunded

Subscription renewals & past_due

For monthly and annual Store purchases, a paid Stripe subscription invoice extends the existing license to the end of its paid billing period. The customer keeps the same key. expiresAt is the paid-through date, not the grace deadline.

A failed renewal sets paymentStatus to past_due. Validation and activation continue through the configured payment grace window: 14 days by default, measured from the paid-through date. Vendors can set 0–365 days in Store → Settings; a license's explicit PostExpiryGraceDays takes precedence, including zero. unpaid keeps the same deadline. Once it is reached, access expires. A successful payment restores active and advances the paid-through date.

During payment grace, the response includes isInGracePeriod: true and paymentStatus: "past_due" or "unpaid". Show a card-update notice while isValid remains true. Non-subscription licenses return no payment state. Offline cache tokens and newly generated offline activation tokens for Store subscriptions are capped at paid-through plus grace. Reconnect to receive a refreshed signed cache token after payment; already-issued offline tokens retain their signed deadline and cannot learn about a cancellation while disconnected.

Scheduling cancellation at period end preserves access through that paid period. When Stripe ends the subscription, customer.subscription.deleted revokes the license. Duplicate invoices and webhook deliveries have no additional effect; older payments never shorten the paid period or reactivate a revoked key.

Register invoice.paid, invoice.payment_failed, and customer.subscription.updated alongside checkout.session.completed, payment_intent.payment_failed, checkout.session.expired, customer.subscription.deleted, and charge.refunded on the vendor's Store webhook endpoint. This is the customer's Store subscription; the vendor's own PermitCore billing is managed separately.

Copy the webhook signing secret and paste it into **Store → Settings → Webhook Secret**.

> **Monthly/annual licenses are revoked on cancellation.** If you sell a monthly or annual store product, PermitCore listens for customer.subscription.deleted and automatically deactivates the license when the customer's Stripe subscription ends — no manual revoke needed. Register this event alongside the renewal events so cancellation, paid-through, and payment grace stay synchronized.

> **0% PermitCore commission.** PermitCore does not charge a sales commission. Your payment processor's fees depend on your Stripe account, region, and payment method.

Tax & Pricing

Choose the tax calculation mode in **Store → Settings → Tax**.

Mode	How it works
Vendor rates	Your default and per-country rates, with the existing country/VAT fields in the cart. Existing stores retain this mode until you change it.
Stripe Tax	Stripe calculates tax from the buyer's address, product tax code and your active registrations. You remain the seller. New stores start with this mode; complete Stripe Tax settings and your origin address before enabling checkout.
Managed Payments	For eligible direct Stripe accounts and digital products, Stripe acts as merchant of record and handles its tax and invoicing responsibilities. Stripe validates availability when creating checkout.

Product settings select a Stripe tax code (default txcd_10103001, SaaS for business use) and **exclusive** pricing (tax added to the price) or **inclusive** pricing (tax included). Provider-calculated tax applies to single purchases, carts and subscriptions created with that mode. Existing subscriptions retain their original Stripe tax configuration and price behavior.

In Stripe modes, billing and business tax details are collected securely at Stripe Checkout. The final tax and total are shown before payment. Stripe Tax collects tax where the vendor has active registrations; a zero-tax result can be correct where no registration applies. If the selected tax setup is unavailable, checkout stops with a clear message. Switching to Vendor rates is an explicit vendor choice.

Business tax IDs and reverse charge

Stripe Tax collects supported business tax IDs and applies reverse-charge treatment where Stripe's tax rules determine it applies. PermitCore records the tax treatment reported by Stripe rather than inferring exemption merely from a zero amount. Vendor rates retains its separate EU VAT/VIES workflow; it is not used for Stripe-calculated tax.

Paid invoice events record actual tax, total, buyer country and business tax ID. Duplicate deliveries do not duplicate invoice records, and an older invoice cannot replace the latest paid figures.

B2B Invoicing

Paid Stripe purchases prefer the Stripe invoice PDF. HTML invoices remain available for trial and legacy orders and use the recorded financial figures. These Stripe and legacy documents include your vendor details, the buyer's details, a line-item breakdown, tax treatment, and a reverse-charge notice when applicable.

Setup: your vendor details

Before invoices are meaningful, add your company details in **Store → Settings → Invoicing**:

Field	Description
Legal Company Name	Your registered company name — printed as "Sold by" on Stripe and legacy vendor-seller invoices. Paddle issues its own documents.
Your VAT Number	Your own VAT registration number — printed on invoices for B2B compliance.

Cart fields collected from buyer

Field	Required	Description
Company Name	Optional	Printed on the invoice as "Buyer company". Leave blank for personal purchases.
Billing country/address	Required in Stripe modes	Collected by Stripe Checkout for provider tax calculation. Vendor rates retains the cart country selector.
Business tax ID	Where supported	Collected by Stripe in provider modes; tax treatment follows Stripe. Vendor rates retains the EU VAT field.

Accessing invoices

Who	How
Admin	Store → Orders → click **Invoice** next to any order. Opens

Who	How
	the Stripe invoice when available. Invoice history keeps each renewal document and tax amount.
Customer	My Account → Orders → click **Invoice** next to any order. The authenticated account also lists Stripe invoice history, preserving earlier renewal documents.

> The invoice opens as a full HTML page and immediately calls window.print(). Use your browser's **Save as PDF** option in the print dialog to get a PDF copy. Keep a downloaded copy for your accounting records.

Customer Accounts (My Account)

A store account is required to buy anything — including a free trial. Browsing products and adding to cart needs no account, but the checkout/trial button always requires sign-in first: an unauthenticated buyer is prompted to sign in or register (returning straight back to the product/cart afterward) before payment even starts. This is deliberate, not a gap that got closed by accident — a purchase with nowhere to log into afterward had no way to ever see the key/activations again except contacting you directly. Accounts are per-store (each vendor's store has its own account system, distinct from the separate, always-anonymous Customer Portal).

Registration & Login

A **Sign In** button appears in the store header. Customers can register with name, email, password, and optional address. After registration, a **verification email** is sent automatically.

Page	URL	Description
Login	/store/{slug}/login	Sign in with email + password. Includes "Forgot password?" link and a "Resend verification email" prompt if the account isn't verified yet.
Register	/store/{slug}/register	New account: name, email, password, country, address.
My Account	/store/{slug}/account	Tabs: **My Orders** (purchase history, invoices), **My Keys** (every license linked to the account — purchase date, expiry, activation/device list, "Add a license" to self-link one you hold the key for), Profile, Security.

Email verification

After registering, customers receive a verification link valid for **48 hours**. Until verified, a yellow banner is shown in My Account with a **Resend** link — also reachable directly from the Login page's error message if a customer tries to sign in before verifying. Verification is not required to buy — sign-

in alone is enough; verification is a separate, softer reminder shown after the fact.

Verification URL format

/store/yourSlug/verify-email?token=token

Forgot / Reset password

The login page has a **Forgot password?** link. The customer enters their email and receives a reset link valid for **1 hour**. For security, the forgot-password endpoint always returns HTTP 200 regardless of whether the email exists (prevents email enumeration).

Reset URL format

/store/yourSlug/reset-password?token=token

What customers can do in My Account

Tab	Features
Orders	Full order history with product name, date, amount. Click **Reveal Key** to decrypt and display the license key. Click **Invoice** to open a printable invoice.
Profile	Update name, email, country, and address.
Security	Change password. Email verification status badge + Resend button if not yet verified.

> **License keys are never emailed.** Purchase and trial notifications always link to the Store account. Customers sign in and verify their purchase email before revealing a retained key. Historical guest buyers create an account with that same email and verify it; keys no longer retained require seller assistance. Old custom template key placeholders are inert. **Browsing and cart building are anonymous; checkout is not.** A customer signs in or registers before PermitCore creates a provider checkout or free trial. This lets the resulting order, key and device activations remain available in My Account without relying on an email address supplied by the browser.

Checkout providers & Paddle setup

Choosing a checkout provider

In **Store → Settings → Payment provider**, choose Stripe, Paddle or Lemon Squeezy for the whole Store. The choice applies to new checkouts. Existing subscriptions and pending purchases keep their original account. Keep those connections enabled while they still receive payment events. An unavailable provider stops checkout until you fix its configuration or explicitly select another provider.

Stripe uses your Stripe account and its tax settings. Paddle and Lemon Squeezy act as merchant of record under your agreement with the selected provider and calculate and collect tax at checkout. PermitCore takes no sales commission; provider fees and payout terms apply.

Settings are grouped into Account, Products & prices, Payment events (Paddle or Lemon Squeezy) and Store settings. Help icons beside labels explain each field. Choosing a provider to view its setup does not activate it: use the provider button to save the checkout choice. Previous payment connections remain available for historical subscriptions without changing new checkout selection. Store settings contains public Store details, applicable tax settings and subscription payment grace.

Connect Paddle

This guide connects **your own Paddle Billing account** to your PermitCore Store. Paddle collects the payment and handles its tax and receipt. PermitCore delivers the software license after confirming payment. You do not need to write checkout code.

Start in sandbox. Sandbox is a separate practice account: use test cards there and no real money moves. A Store on the production website can still use sandbox. Changing the provider to view its settings does not activate it; the **Active** badge shows the saved choice.

1. Prepare your Store and Paddle account

In PermitCore, open **Store → Products**. Create the listings you want to sell, with a currency and at least one one-time, monthly or annual price. Set the license features and activation limit. Make the listings active when you are ready to test them. Your Store also needs to be public for buyers to open it.

Open your Paddle sandbox dashboard. This integration uses **Paddle Billing**, not Paddle Classic. In PermitCore, open **Store → Settings**, choose **Paddle** and leave **Use sandbox / test mode** checked. Stay on the **Account** tab.

For payment-by-payment behavior, read the Paddle license delivery and renewal guide. It explains paid renewals, grace, refunds and why card-change or credit-only updates do not grant paid access.

2. Find the three account values

Seller ID. Your numeric Paddle account ID, available in Paddle account settings. Copy the ID belonging to the workspace you are configuring.

API key. A private server credential. In Paddle, open **Developer tools → Authentication**, then create an

API key for PermitCore. A sandbox key starts with pdl_sdbx_; a live key starts with pdl_live_. Paste it only in PermitCore's **API key** field. Store your original securely: PermitCore will not display it again.

Paddle.js client-side token. A different, public credential that opens the payment form. In Paddle's **Authentication** settings, open **Client-side tokens** and create a token. Some dashboard versions place Authentication under **My account → Settings**. Copy a test_ token for sandbox or a live_ token for live. Paste it in **Paddle.js client-side token**, never the API key field.

Which API key permissions should I choose?

Enable these permissions for the current PermitCore flow. Write permission includes read. Dashboard labels may show the entity name followed by Read or Write.

- account.read — verify your Seller ID.
- product.write and price.write — read and sync the Store catalog.
- transaction.write — create checkouts and read paid transactions and their documents.
- subscription.read, customer.read and adjustment.read — confirm subscription, buyer and refund updates.
- customer_portal_session.write — allow Paddle to return authenticated subscription management links.

You configure notification destinations in the Paddle dashboard; the Store does not need permission to create them through the API. If a key has expired, been revoked or lost a required permission, replace it and verify the saved connection again.

3. Set the payment page and its domain

Your payment page is your public Store address with /paddle-checkout added. For example, a Store at https://permitcore.dev/store/your-slug uses https://permitcore.dev/store/your-slug/paddle-checkout. Replace your-slug with the last part of your own Store address. If you use another public website hostname, keep that hostname.

1. In Paddle, open **My account → Settings → Website approval**. Add the exact hostname used by that page. A staging subdomain is a separate hostname. Sandbox entries are approved automatically; live entries require review.
2. Open **Checkout → Checkout configuration**. Set **Default payment link** to the complete payment-page URL and save it. Adding the URL and obtaining live website approval are separate steps.
3. Paste the same complete URL into **Approved payment page URL** in PermitCore.

For a **new connection only**, leave **Keep this connection enabled for payment events** unchecked and click **Save account**. This saves the account as disabled while you obtain its signing secret and displays your connection-specific webhook URL. Enable it in the next step after entering that secret. Do not disable an existing connection that still receives payment events.

4. Connect payment notifications

A webhook is an automatic message from Paddle when a payment, renewal or refund changes. Without these messages, a customer can pay while license delivery remains pending.

1. After saving, copy **Webhook destination URL** from the PermitCore Account tab. Copy the

displayed value; do not construct it yourself or copy another vendor's URL.

2. In the same Paddle workspace, open **Developer tools → Notifications** and create a notification destination. Choose a webhook/URL destination, paste the URL and keep delivery enabled. It must receive real events; a simulation-only destination cannot deliver purchases.
3. Expand **Events to enable in your provider dashboard** in PermitCore. Select every listed event in Paddle, including transaction completion, subscription updates and both adjustment events. An adjustment is Paddle's record of a refund or credit.
4. Save the destination in Paddle. Copy its **signing secret** into PermitCore's **Webhook signing secret** field. Check **Keep this connection enabled for payment events**, then click **Save account** again.
5. Click **Verify saved API connection**. The **API verified** badge confirms credentials and account identity. It does not confirm domain approval or a completed purchase. Every account save resets this verification, so verify again after saving changes.

API key and signing-secret fields are intentionally blank after reload. The saved-value message confirms they are stored; leave them blank to keep them. The signing secret comes from the notification destination and is different from both account credentials.

5. Send your Store products to Paddle

Open **Products & prices** and click **Create / update Paddle catalog from Store**. This creates or updates Paddle products and connects the available one-time, monthly and annual prices. Running it again skips unchanged entries; existing subscription prices are retained. After editing Store prices, run sync again.

Check the resulting **Connected prices**. You normally do not need to type a price ID. **Manually connect a price** is for an existing Paddle price in the same workspace, with matching amount, currency, billing interval and tax treatment.

Review each product's tax category in Paddle before selling. PermitCore does not translate Stripe tax codes into Paddle categories. Inclusive Store prices contain tax; exclusive prices add tax at checkout. The buyer's final tax and provider document come from Paddle. Review these in a test purchase.

6. Activate sandbox and complete a purchase

1. At the top, confirm **Paddle** and **Use sandbox / test mode**, then click **Use Paddle**. The badge should read **Active: Paddle · Sandbox / test mode**. This changes new checkouts for the entire Store, including other buyers visiting it.
2. Open your public Store as a buyer, sign in or create a buyer account, and purchase a test listing. Use Paddle's documented sandbox test cards. Do not enter a real card in sandbox.
3. In PermitCore, open **Payment events** and refresh. Confirm the payment event is processed. Then check **Store → Orders** and the buyer's account: the order should be completed, labeled as sandbox, with its license and provider receipt.
4. Check license email delivery to your test inbox and validate or activate the license in your application. A success page or an API verified badge alone is not proof of delivery.
5. If you sell subscriptions, test a paid renewal, failed payment and recovery, and cancellation. A paid

renewal extends the original license key. Failed renewal access uses **Store settings → Subscription payment grace**, normally 14 days from the paid-through date.

6. Test a refund in Paddle and confirm the approved refund appears in Store Orders. Full refunds revoke affected license access; partial refunds retain it. Cancel a subscription separately if you also want future billing to stop.

Do not leave test listings public after testing. Refunds can remain pending until Paddle approves them; sandbox approvals may take up to ten minutes. Keep the connection enabled while you wait. Test licenses do not become live licenses when you change environments.

7. Go live with a separate connection

Finish Paddle's live account and website approval first. In PermitCore, uncheck **Use sandbox / test mode** to view the separate live connection. Repeat the account, payment-page, notification, verification and catalog steps with live credentials and a live_ client token. Sandbox keys, signing secrets and price IDs do not transfer.

Only click **Use Paddle** after the live account is ready. Confirm **Active: Paddle · Live mode**, then check a controlled real purchase and refund before inviting buyers. Real money moves in this mode. Sandbox success does not prove live approval, payouts or live tax configuration.

Changing to Stripe or another environment affects new checkouts only. Existing subscriptions keep their original Paddle connection. Keep old connections and notification destinations enabled until all their subscriptions, pending payments and refunds are finished.

Something is not working

Use Paddle is disabled. Save the selected environment's account with its signing secret, keep it enabled and verify it. Check which environment is selected. Product prices and a real purchase still need separate checks.

API verification fails. Check Seller ID, sandbox/live key prefix, required permissions and whether the saved key was revoked. After replacing a key, save and verify again.

Checkout says the default payment link is missing or the domain is not allowed. Check both the saved default payment link and the exact hostname in Website approval in the same Paddle workspace. Approval of a main domain does not cover every subdomain.

A price is missing or rejected. Save the Store product, run catalog sync and inspect Connected prices. For manual connections, compare account, environment, currency, amount, interval and tax treatment. Unsupported trials or price overrides need a compatible price.

Payment succeeded but no license arrived. Open Payment events. If no event arrived, check destination URL, enabled events and Paddle delivery logs. For failed events, correct the signing secret, credentials or product mapping, then retry the failed delivery or use Retry in PermitCore. Repeated delivery must not create another license. If the order is complete but email is missing, check the buyer email, spam folder and your configured mail delivery.

A refund is pending or a subscription is still billing. Wait for refund approval and check payment events. A refund does not cancel the subscription; cancel future billing separately in Paddle. Keep

historical connections enabled so updates can reach PermitCore.

Provider reference: Paddle quickstart, API permissions and default payment link. Dashboard wording can change; use the setting names above to locate them.

The payment page checks the signed-in buyer and the transaction's original Store connection before opening Paddle Checkout. Browser completion never grants a license on its own.

Review the tax category on each Paddle product before going live. Catalog creation starts with Paddle's standard category; existing provider categories are retained. The local Stripe tax code is not automatically translated into a Paddle category.

Connect Lemon Squeezy

Connect **your own Lemon Squeezy Store**. Lemon Squeezy handles payment, tax and its receipt under your merchant-of-record agreement. PermitCore delivers the license after verifying payment. Start with Test mode and matching test products; no real card details are needed.

1. Prepare the products

In PermitCore, create the Store listings, prices, license features and device limits. In the Lemon Squeezy dashboard, turn on **Test mode** and publish the corresponding products and variants. Products must be created in their dashboard; PermitCore cannot push its catalog through the Lemon Squeezy API. Match currency and one-time, monthly or annual intervals.

2. Connect the account

Open **Store → Settings → Payment provider → Lemon Squeezy → Account**. Viewing setup does not change your saved provider. Keep **Use sandbox / test mode** selected. Enter your Lemon Squeezy **Store ID** and an API key created under **Settings → API** in the same mode. Keep the key private. Leave an existing secret field blank to keep its saved value.

3. Save before creating the webhook

For a new connection, clear **Keep this connection enabled for payment events** and save the account. This creates its unique **Webhook destination URL**. Copy that URL. Existing connections receiving payment events should remain enabled.

In Lemon Squeezy **Settings → Webhooks**, create a webhook for that URL in the same mode. Choose a private signing secret and select the events listed under **Events to enable in your provider dashboard** in PermitCore. Copy that same signing secret into PermitCore, enable the connection, save and click **Verify saved API connection**. API verification checks credentials; it does not prove payment or webhook delivery.

4. Match prices and tax mode

Open **Products & prices → Manually connect a price**. Select your Store product and price option, enter the corresponding Lemon Squeezy variant ID and choose the license settings. Verify and save each mapping. Match PermitCore's Lemon Squeezy tax setting to **Tax-inclusive pricing** in Lemon Squeezy General Settings. Select the appropriate product tax category there. Do not infer the live tax

rate from a Test-mode zero-tax receipt.

5. Prepare a cart with several products

Lemon Squeezy checkout bills one variant. For a multi-product cart, create a generic order product in its dashboard and enter its variant ID under the aggregate checkout fields in Account. Prepare separate one-time, monthly and annual variants for the intervals you sell. PermitCore submits the cart total as a custom price and fulfills its own saved cart items. The provider receipt shows one aggregate line; PermitCore preserves the individual purchase shares.

A recurring aggregate creates one subscription for its grouped items. One-time and recurring items, or monthly and annual items, require separate checkouts. Never use a one-time variant for a subscription.

6. Test payment and access

Save the provider choice with **Use Lemon Squeezy** only when the account and mappings are ready. Check the **Active** provider and mode. Complete a Test-mode purchase using a provider test card, then confirm the payment event, issued license, customer history, provider receipt and email delivery. Test initial subscription creation, cancellation and refunds. Use supported event simulations to check signed renewal/failure/recovery delivery; a simulation does not update the invoice or prove a charge. Retried notifications must keep the same keys.

A dashboard-simulated success does not prove a paid renewal. PermitCore re-fetches the invoice and subscription; a pending invoice cannot extend access or clear payment debt. Failed renewals use your saved payment grace, normally 14 days. A paid renewal extends the existing key. Cancel future billing separately when refunding a subscription purchase.

7. Switch to live deliberately

Activate your Lemon Squeezy store with the provider, prepare live products, live API credentials and a live webhook, and verify them separately. Test products and keys do not automatically become live. Select live mode and save the provider choice only after completing your checks. Existing purchases and subscriptions keep their original provider connection; keep its payment events enabled.

Official references: API keys, webhook signing, checkout custom prices and testing and going live.

Test Mode limits.
A Croatian test order can show 0% VAT even with Software and tax-inclusive pricing. Pending renewal invoices may not be payable in Test Mode. Test checkout and initial subscription creation with the provider's test cards; simulated payment events do not change the invoice or prove payment. PermitCore checks the provider API before granting or extending paid access. Do not use a real card or self-purchase just to test tax or renewal processing.
If Payment events reports a missing price mapping, open View details to see notification price references and the saved mapping state. Verify the transaction in your provider account before linking its price to the correct Store product and billing cycle. A notification reference is a diagnostic hint, not proof of payment. Do not map an unrelated price to clear an error. Canonically verified zero-value Paddle transactions do not issue or renew licenses, send license emails or clear payment debt.

Provider invoices and payment history

Store Orders and the customer's Purchase History retain the recorded shares of every paid provider transaction or subscription invoice, including tax and refunds. Open View provider document for the complete provider-issued receipt or invoice. Paddle PDF links are refreshed when opened; an old signed link is not treated as permanent. PermitCore does not generate a separate vendor-seller invoice for Paddle or Lemon Squeezy merchant-of-record purchases.

Refunded purchases keep their payment history and provider documents while refunded license access is revoked. Sandbox orders are labeled. Existing payments and subscriptions retain their original provider connection when the Store selection changes.

Sandbox licenses and activations are excluded from Dashboard, Analytics and weekly digest statistics. Sandbox orders are excluded from reseller commission lists, totals, exports and payout marking. They remain in Licenses and Store Orders. Commission follows the initial order snapshot; renewals do not create a new commission. Partial refunds do not automatically reduce it. Reconcile already-paid commissions manually after refunds.

Use **Copy webhook URL** beside your provider destination, or **Copy URL** for Stripe. A status message confirms copying; if your browser blocks clipboard access, select and copy the displayed URL manually. Copying does not save or activate a provider.

Read the saved mode and payment events

The Active label always identifies the saved checkout provider and mode. Browsing another provider or changing its sandbox checkbox does not activate it. Stripe shows its saved key mode; mismatched or unreadable keys are shown explicitly. The mode label is not proof of API access or live account approval. On phones, payment events appear as labelled cards with their actions visible. In Payment events, use View details for status, timestamps, attempts, errors and allowed notification/object identifiers. Customer data, secrets and full payloads are not shown. Correct a failed event before Retry.

Subscription updates with no payment

Paddle completed subscription updates that only generate account credit (or have zero totals) are ignored after canonical API validation. They do not issue licenses, extend paid-through dates, send license emails or clear payment debt. Real paid renewals still require independent payment evidence. Paddle may show zero due for a downgrade that adds credit to the customer balance; that credit is not a new paid renewal.

If Payment events reports a missing price mapping, open View details to see notification price references and the saved mapping state. Verify the transaction in your provider account before linking its price to the correct Store product and billing cycle. A notification reference is a diagnostic hint, not proof of payment. Do not map an unrelated price to clear an error. After 10 unsuccessful attempts, Archive can acknowledge an exhausted failure while retaining its history and payment records. Confirmation is required, and Retry can reopen the archived notification. Pending retries and completed events cannot be archived. Canonically verified zero-value Paddle transactions do not issue or renew licenses, send license emails or clear payment debt.

Use **Copy webhook URL** beside your provider destination, or **Copy URL** for Stripe. A status message confirms copying; if your browser blocks clipboard access, select and copy the displayed URL manually. Copying does not save or activate a provider.

Verify delivery before opening your Store

API verification checks credentials and account identity. It does not prove webhook delivery, completed payment or license email delivery. Complete a sandbox purchase and check the license, email, invoice, renewal, failed payment, refund and cancellation. Test licenses stay in test mode. Correct failed event mappings or credentials before using Retry; repeated notifications must not create extra keys.

Paid renewal periods extend the existing key. Failed payments use the Store's configured payment grace, default 14 days. Scheduled cancellation preserves paid access through its boundary; an ended subscription or full approved refund stops access. A partial refund records the amount while retaining access. Refunding a license does not cancel future provider billing: cancel the subscription separately where needed. Previously issued offline tokens retain their signed expiry.

Store setup checks

Before accepting payments

Review the provider and mode you are configuring. API verification alone does not prove that checkout, webhooks or license delivery work. These checks do not activate a provider.

1. **Connect the account.** Save the correct test/live credentials and webhook signing secret in **Account**. For Paddle and Lemon Squeezy, verify the saved API connection.
2. **Connect products and prices.** Open **Products & prices**. Check every product and billing interval you intend to sell, including currency and tax treatment.
3. **Check webhook delivery.** For Paddle and Lemon Squeezy, review **Payment events**; for Stripe, inspect delivery in the Stripe dashboard. A received, ignored or processed notification alone is not proof of a paid order.
4. **Complete a sandbox purchase.** Review the confirmed payment and correct license in **Orders**, then sign in as that buyer and check account access. The notification email must link to the account and never contain the license key. Check cancellation/refund behavior for the sales model you use.

Closing the Store or changing plans: without Store access, the catalog and new purchases are unavailable. Existing buyers can still sign in directly to their account, recover their password and review their own licenses, orders and receipts. Saved payment connections still handle earlier purchases, refunds and subscription events; closing the catalog does not silently cancel those subscriptions.

Sandbox evidence does not verify the separate live account, domain approval or live credentials. Save the provider choice explicitly when its configuration is ready. Store publication is a separate saved setting.

Store Settings includes an expandable setup overview for the provider and mode being viewed. It summarizes saved API verification, enabled verified mappings and recent notifications where available. Stripe credentials and deliveries need their separate checks. The overview never marks a purchase or live readiness as verified automatically.

Customer Portal

Licensing Features

Customer Self-Service Portal

Every PermitCore tenant gets a public self-service portal at /portal/{your-slug}. Customers can look up their own license key, view their activation history, and remove devices — without contacting you.

Portal URL

URL format

https://your-domain.com/portal/your-company-slug

Your company slug appears on the **Settings → Profile** page. Share this URL in your app's About screen, license email, or support docs.

What customers can do

Action	How
View license status	Enter license key → see product, expiry, activation count
See activation history	Lists all activated devices with device name, country, and date
Remove a device	Click "Remove" next to any activation — frees one seat for re-use
Check custom fields	Any custom fields (edition, seats, etc.) are shown in the portal
See every license in one place (optional)	With an account (/portal/{your-slug}/account), link licenses once and see them all in a dashboard instead of looking one up at a time

Security — three levels

By default, the portal uses the **license key itself as the authentication token** — possession of the key proves ownership. Customers can only see and modify activations for keys they provide. The server computes a hash of the submitted key and looks it up — the plaintext key is never stored. This is the right default for most software vendors: a desktop or embedded app has nowhere to hold a "real" account credential that isn't itself just another copy of the key.

If your licensing model needs more than that — enterprise volume licenses managed by IT, named accounts, a stricter compliance requirement — **Settings → Security Policy** lets you raise the bar:

Level	What it requires	Plan
Key only (default)	License key alone, as described above.	All plans
Key + email code	Viewing status still only needs the key. Removing a device additionally emails a one-time code to the license's linked contact, which must be entered to confirm.	Professional+
Account required	The key-only portal is disabled entirely. Customers sign in with an email/password account and only see licenses linked to that account — a dashboard of everything they own, not one lookup per key.	Enterprise

> The email-code and account levels both need your licenses actually linked to a contact — either you set it in the CRM (**Customers** page), or the customer links it themselves the first time they sign in, by entering a key they already have. A vendor turning on stronger security without linking any licenses first will just lock customers out — link before you enforce.

White-label branding

The portal is fully **white-label** — your customers see your brand, PermitCore is invisible. Go to **Company → Portal Branding** tab and configure:

Setting	What it does	Plan required
Portal Display Name	Company name shown in the portal header instead of "Your License"	Starter+
Logo URL	Your logo displayed top-left — any direct image URL (HTTP/HTTPS)	Professional+
Brand Color	Hex color applied to buttons and accent elements (color picker included)	Professional+
Custom CSS	Full CSS control — fonts, backgrounds, layout, hide any element. No <script> tags. Max 10 000 chars.	Enterprise

All changes go live immediately — no rebuild, no cache invalidation needed.

White-label portal — Nexova Technologies logo, brand color (#10b981), company name. PermitCore branding is absent.

After entering the license key — product name, activation count, and list of activated devices with self-

service Remove buttons.

> **What "white-label" means in practice:** From your customer's perspective, they visit *your* portal, see *your* logo, and interact with *your* brand colors. PermitCore appears only as a small "Powered by" footnote — which you can hide entirely with one line of CSS: `footer { display: none; }`

Linking from your app

In your About screen or settings UI

```
// Open the portal directly to the customer's license
Process.Start(new ProcessStartInfo(
    "https://your-domain.com/portal/your-slug"
) { UseShellExecute = true });
```

> **Looking for the store's My Account portal?** This page documents the public license validation portal (`/portal/{slug}`) where anyone can look up a key. The store-specific customer account portal (`/store/{slug}/account`) — with order history, invoice download, and password management — is documented in the Customer Accounts section under License Store.

Reseller Portal

Licensing Features

The Reseller Portal lets you invite partner companies or distributors to **issue licenses on your behalf**. Each reseller gets their own login and can create licenses only for the products you authorize — with optional caps on how many they can issue.

> Available on the **Enterprise** plan.

The reseller invitation form before submission. Choose product access and configure the partner before sending the invitation.

How it works

1. Go to **Resellers** in the sidebar and click **Invite Reseller**
2. Enter the reseller's name, email, and a temporary password
3. Select which products they can sell and optionally set a max license count per product
4. The reseller logs in with their credentials and creates licenses from their own dashboard
5. New live activations of reseller-issued keys count against **your** total active device capacity; creating the key itself does not consume an activation

Reseller permissions

Can do	Cannot do
Create licenses for assigned products	Access products not assigned to them
View their own created licenses	See other users or company settings
View order/activation counts for their licenses	Access billing or payment settings
Generate up to their per-product cap	Exceed the license cap set by you

Managing resellers

From the Resellers page you can:

- **Deactivate** a reseller — immediately blocks their login without deleting their account or the licenses they've created
- **Adjust product access** — add or remove products, change the license cap
- **Monitor issuance** — see how many licenses each reseller has created per product

> **Use case:** You have a desktop app sold through 3 regional distributors. Each distributor gets a Reseller account with access to the relevant product. You cap each at 100 licenses. They log in, generate license keys for their customers, and you see all activations in your analytics dashboard.

Commission tracking

Sandbox orders are excluded from commission totals, lists, CSV exports and payment marking. Renewals do not create another commission. Partial refunds do not automatically adjust commission; fully refunded orders are excluded from unpaid earnings. Reconcile previously paid commissions manually after refunds.

When you invite a reseller, you can set a **commission percentage** per product (0–100%). Each initial order placed through a reseller link records the commission amount at the time of sale — so changing the rate later does not retroactively affect existing orders.

1. Go to **Resellers → Invite Reseller** and enter a commission % for each product.
2. When a customer purchases through a reseller link, the order records ResellerId, CommissionPercent, and the computed CommissionAmount.
3. Open the **Commissions** tab on the Resellers page to see all pending and paid commissions grouped by reseller.
4. Select one or more orders and click **Mark as Paid** to record payment. The system logs CommissionPaidAt and updates the reseller's running total.

Field	Description
CommissionPercent	Rate set per reseller × product pair (snapshot taken at sale time)
CommissionAmount	Pre-computed: OrderTotal × CommissionPercent / 100
TotalEarned	Running total of all commission amounts ever assigned
TotalPaid	Running total of commissions marked as paid
Unpaid balance	TotalEarned − TotalPaid — shown in the Commissions tab

> Commission percentages are **snapshots** — captured at the moment of purchase. You can update a reseller's rate at any time without affecting historical orders. Resellers can view their own earned and unpaid totals via the self-service summary endpoint.

Metered Billing

Licensing Features

Metered billing lets you charge customers based on **actual usage** rather than a flat seat count. Your software reports a usage event every time a customer performs a billable action — calling an API, exporting a file, generating a report. PermitCore records every event and, with Stripe connected, automatically charges customers at the end of each billing period. **No spreadsheets, no manual invoicing.**

Usage tracked per event | Stripe auto-charge | Explicit usage reporting | Fully automatic for Store customers

> **Two modes:** You can use metered billing *without Stripe* — just to track usage in the PermitCore dashboard and invoice manually. Or connect Stripe to auto-charge customers based on the usage data. Both modes are supported.

Recording a usage event — SDK call

POST /api/v1/meter — no authentication required. Rate limited to 60 requests/minute per IP.

curl

```
curl -X POST https://api.permitcore.dev/api/v1/meter \
  -H "Content-Type: application/json" \
  -d '{"licenseKey":"PERMIT-XXXX-XXXX-XXXX-XXXX","eventName":"api_call","quantity":1}'
```

JavaScript

```
await fetch('https://api.permitcore.dev/api/v1/meter', {
  method: 'POST',
  headers: { 'Content-Type': 'application/json' },
  body: JSON.stringify({
    licenseKey: 'PERMIT-XXXX-XXXX-XXXX-XXXX',
    eventName:  'api_call',
    quantity:   1
  })
});
```

Python

```
import requests
requests.post('https://api.permitcore.dev/api/v1/meter', json={
    'licenseKey': 'PERMIT-XXXX-XXXX-XXXX-XXXX',
    'eventName':  'api_call',
    'quantity':   1,
})
```

Request fields

Field	Required	Description
licenseKey	Yes	Plaintext license key — hashed and resolved via HMAC lookup server-side, never stored
eventName	Yes	Billable action slug, e.g. api_call, export, report_generated. Normalised to lowercase. Must be consistent — this same name is used to configure pricing.
quantity	No	Units consumed. Defaults to 1. Use for batch operations (e.g. 5 if processing 5 items at once).
meta	No	Arbitrary JSON stored with the event (e.g. file format, region). Visible in the admin event list.

Response

JSON — success

```
{ "recorded": true }
```

Handle metering failures. Await or durably queue usage reporting and inspect its result. A lost usage event can cause underbilling; blindly retrying an uncertain result can duplicate usage. The call is fast, but Stripe forwarding happens asynchronously in PermitCore — the SDK response is never delayed by Stripe latency.

Choosing event names

Event names are normalised to lowercase and matched exactly. Choose them carefully up front — they appear in your Stripe Dashboard, invoices, and customer billing pages.

Good	Why
api_call	Machine-readable, consistent, clear unit
export	One word, noun — matches what the customer did
report_generated	Verb past-tense, unambiguous
sms_sent	Describes the unit being billed

Do not change event names after launch. The event name is the key used to match usage to Stripe Billing Meters and to PermitCore pricing config. Renaming it mid-flight means old events stop being counted and auto-charge breaks.

Meter Usage dashboard

Go to **Meter Usage** in the admin sidebar. The dashboard shows:

- **Total events** — sum of all events in the selected period (7 / 30 / 90 days)
- **Unique event types** — count of distinct event names
- **Average events / day** — total ÷ days in period
- **Daily bar chart** — volume per calendar day, useful for identifying traffic spikes
- **Event breakdown table** — each event type, total count, and share percentage

Buttons at the top of the page:

- **Pricing** — configure per-event Stripe pricing (see below)
- **Stripe Setup** — retroactive backfill wizard for existing customers

Stripe Auto-Charge

Licensing Features

Stripe Auto-Charge for Metered Usage

Connect PermitCore to Stripe to have customers billed automatically based on their usage — no manual invoicing, no exports, no scripts. When a Store customer triggers their first meter event, PermitCore creates the Stripe mapping and subscription item automatically. You configure prices once and everything else runs itself.

> **For Store customers: zero vendor action after initial setup.** Once you set a price for an event name, every customer who bought via your store is automatically billed on their first meter event. No per-customer configuration required.

Prerequisites

- A Stripe account (live or test mode)
- Your **Stripe Secret Key** added to PermitCore under **Settings → Company → Payment Settings**
- Customers who bought via your PermitCore Store — their Stripe `cus_xxx` ID is captured automatically on checkout

Step 1 — Configure pricing for each event name

Go to **Meter Usage → Pricing**. This page shows all event names that have been recorded, plus any you configure in advance.

Click **Add Event Price** and fill in:

Field	Description	Example
Event Name	Must exactly match the `eventName` your SDK sends. Normalised to lowercase automatically.	`api_call`
Display Name	Human-readable name shown in Stripe Dashboard and customer invoices.	API Calls
Unit Label	The noun Stripe uses on invoices — e.g. "per API call" on the invoice line.	`api calls`
Price per Unit	Cost in the selected currency per single event unit. Supports up to 6 decimal places for micro-pricing.	`0.0100` ($0.01)
Currency	USD, EUR, or GBP. Must match the currency your Stripe subscription	USD

Field	Description	Example
	uses.	

Click **Save & Create in Stripe**. PermitCore immediately:

1. Calls the Stripe Billing Meters API to create a new **Billing Meter** with your event name, aggregation formula (sum), and value key (value).
2. Creates a recurring **metered Price** linked to that meter (monthly interval).
3. Stores the Stripe Price ID (price_xxx) in your database.

You will see a green **Stripe status** badge next to the event name showing the Price ID. This confirms Stripe is ready.

> **The Stripe Meter is created once per event name per Stripe account.** If you save the same event name again (e.g. to change the price), PermitCore reuses the existing Stripe Meter and creates a new Price linked to it. Old Price IDs are replaced.

Step 2 — What happens on the first meter event (fully automatic)

After Step 1, everything else is automatic. Here is exactly what happens when a customer's software calls POST /api/v1/meter for the first time for a given event:

1. **PermitCore records the event**
   The meter event is saved to the database immediately and { "recorded": true } is returned to your SDK. Everything else happens in the background.

2. **Customer mapping is auto-created**
   PermitCore checks if a Stripe customer mapping already exists for this license key + event name. If not, it looks up the Order record from when the customer bought via your Store — which contains their Stripe cus_xxx ID (captured automatically from the Stripe Checkout session). A LicenseMeterStripeConfig record is created linking the license key, event name, and Stripe customer ID.

3. **Metered price added to customer's subscription**
   PermitCore finds the customer's active Stripe subscription. It checks if the metered price (from Step 1 of setup) is already attached. If not, it calls SubscriptionItemService.CreateAsync to add the price to the subscription. From this moment, Stripe will aggregate all future meter events for this customer.

4. **Usage event pushed to Stripe Billing Meters**
   PermitCore calls the Stripe Billing Meters API (MeterEvent.create) with the event name, the customer ID, and the quantity. Stripe records this event against the meter.

5. **All subsequent events go straight to Stripe**
   Steps 2 and 3 only run once — when the mapping is first created. Every subsequent POST /api/v1/meter call for the same license key + event name skips directly to pushing the event to Stripe. The customer is invoiced at the end of each billing period by Stripe automatically.

> **Stripe Billing handles invoicing.** PermitCore never touches money directly. Stripe aggregates all usage events

during the billing period and auto-generates + charges the invoice. You see all of this in your Stripe Dashboard under Customers → Subscriptions → Usage.

For customers who didn't buy via the Store

If a customer's license was not sold through the PermitCore Store (e.g. sold offline, via a reseller, or via a custom flow), their Stripe customer ID (cus_xxx) is not automatically available. You need to configure the mapping manually.

Go to **Licenses → View** for the relevant license key. Scroll down to the **Stripe Metered Billing** card.

1. **Click "Add Mapping"**
   Opens a modal. Enter the event name (must match exactly what your SDK sends) and the customer's Stripe ID in the format cus_xxx.
2. **Test the connection**
   Click **Test Connection**. PermitCore calls the Stripe API to verify the customer exists and returns their name and email from Stripe — confirming the cus_xxx ID is correct before saving.
3. **Save Mapping**
   The mapping is saved. The next meter event from this license key for this event name will trigger the auto-subscription-item step (Step 3 above) and then push to Stripe.

You can add multiple mappings per license — one per event name if different events should go to different Stripe customers (rare, but supported).

Retroactive backfill — customers who used meter before setup

If you had customers using POST /api/v1/meter before you configured pricing, those events were already recorded in PermitCore but never forwarded to Stripe. To backfill:

1. Go to **Meter Usage → Stripe Setup**.
2. The wizard shows all event names with existing meter events, how many licenses are auto-mappable (have Store orders with cus_xxx), and how many need manual setup.
3. Click **Backfill Now** to auto-create LicenseMeterStripeConfig records for all historical Store-purchased licenses.
4. Any remaining licenses with no cus_xxx are listed in Step 3 of the wizard with direct links to their license detail pages for manual mapping.

Backfill creates mappings — it does not replay historical events to Stripe. Historical meter events already recorded in PermitCore before the backfill will not be charged. Only new events going forward are forwarded to Stripe. To charge for historical usage, create a manual Stripe invoice line item in the Stripe Dashboard.

End-to-end example — $0.01 per API call

A vendor sells an API gateway product. Customers pay a base subscription plus $0.01 per API call.

1 **One-time setup (vendor)**
Open **Meter Usage → Pricing**. Add event api_call, Display Name "API Calls", Unit Label "api calls", Price $0.01, Currency USD. Click **Save & Create in Stripe**. Done — Stripe Meter + Price created automatically.

2 **Customer purchases via Store**
Customer buys through the vendor's PermitCore Store. Stripe Checkout runs, the customer's Stripe cus_xxx ID is captured and saved on the order automatically.

3 **Customer integrates SDK**
The customer adds one line to their code per API request: Python client.meter(license_key, "api_call", quantity=1)

4 **First SDK call — full auto-setup in background**
On the very first api_call event from this customer: PermitCore creates the customer mapping, adds the metered price to their Stripe subscription, and pushes the event. The response confirms that the event was recorded locally; it does not confirm successful delivery to Stripe.

5 **End of billing period**
Customer made 5,000 API calls. Stripe aggregates all meter events and auto-generates an invoice for 5,000 × $0.01 = **$50**. The customer is charged automatically. The vendor sees the charge in their Stripe Dashboard. No manual work needed.

How to confirm it actually worked

After sending a test POST /api/v1/meter call, check these three places — in order, since the first one confirms the fastest and the mapping/Stripe steps only run once per customer:

Where	What confirms success
The API response	{ "recorded": true } only confirms PermitCore accepted the event locally — not that Stripe received it.
License detail page → Stripe Metered Billing card	The mapping row now shows the Stripe Customer ID this event was routed to.
Stripe Dashboard → Billing → Meters → your meter → Event summaries	The definitive proof — an aggregated usage value against that customer for the current period. If this shows the expected total, the full pipeline worked end to end.

What you see in Stripe Dashboard

After setup, in your Stripe account you can see:

- **Billing → Meters** — your meter(s) with event name, current period usage, and aggregated totals
- **Customers → [customer] → Subscriptions** — the subscription now has a metered line item showing usage
- **Billing → Invoices** — at period end, a draft invoice is generated then auto-paid

Stripe auto-charge API reference

Endpoint	Auth	Description
GET /api/meter/pricing	CompanyAdmin	List all configured event prices. Also returns known unconfigured event names from recorded usage.
PUT /api/meter/pricing	CompanyAdmin	Create or update a price config. Immediately creates the Stripe Billing Meter + metered Price.
DELETE /api/meter/pricing/{eventName}	CompanyAdmin	Remove the local price config. Does not delete the Stripe Meter or Price (Stripe resources are kept).
GET /api/meter/stripe/{licenseKeyId}	CompanyAdmin	List manual customer mappings for a license key.
PUT /api/meter/stripe/{licenseKeyId}	CompanyAdmin	Create or update a manual customer mapping (eventName + stripeCustomerId).
DELETE /api/meter/stripe/{licenseKeyId}/{eventName}	CompanyAdmin	Remove a manual customer mapping.
POST /api/meter/stripe/{licenseKeyId}/test	CompanyAdmin	Verify a cus_xxx exists in Stripe. Returns { valid, email, name }.

Metered-price setup failures return a fixed operation message instead of raw Stripe exception text. A provider failure returns HTTP 400 without saving the local price change; previously created provider resources can still exist. Check the configured Stripe account and mode, then review Stripe Dashboard request logs.

Dunning Management

Failed PermitCore subscription payments

These reminders concern your company's subscription to PermitCore. When its payment fails, company admins receive an initial email, followed by reminders scheduled for day 3 and day 7 while the payment remains unresolved.

1. Open **Billing** from your admin account.
2. Review the failed invoice and update your payment method through the billing portal.
3. Confirm the payment and subscription status. A successful payment resolves the reminder sequence.

Reminder delivery depends on the instance's mail configuration and payment webhooks. On the hosted service these are operated by PermitCore; self-hosted operators must configure and monitor them. Scheduled follow-ups may arrive after their nominal time because they are processed by a background worker.

For payments made by buyers in your own store, configure the vendor Stripe integration described in License Store.

Customer Notifications

Licensing Features

End-Customer Expiry Notifications

PermitCore can send expiry reminder emails **directly to your end customer** — not just to you as the admin. When a **Customer Email** is set on a license, that address receives branded reminder emails at **30, 7, and 1 day** before the license expires, prompting the customer to renew.

The Customer Email field in the Edit Metadata modal — set per license, completely optional.

How to set it up

1. Go to **Licenses → View → Edit Metadata**
2. Enter the customer's email in the **Customer Email** field
3. Save — the email will be used for all future expiry notifications for this license

Setting a Customer Email does **not** replace admin notifications — admins still receive their daily digest. This is an additional, per-license notification that goes directly to the end user.

The customer email

The email sent to the customer:

- Is delivered using the instance's configured mail provider
- Shows **your company name** as the sender — PermitCore is not mentioned
- Includes the product name, expiry date, and days remaining
- Asks the customer to contact *you* to renew

> **Use case:** You sell annual desktop app licenses. Instead of customers manually tracking expiry dates, PermitCore sends them a reminder 30 days before expiry — giving them time to renew before their software stops working.

GDPR Data Export

Export account data

A company admin with a verified email can download an account-data ZIP from Company Settings. This is available on every plan. The export is a selected set of account records, not a complete database backup or a collection of plaintext license keys.

What the ZIP contains

File	Contents
tenant.json	Company profile, activity status, billing tier, timestamps and the encryption salt needed with your own passphrase.
users.json	Team identity, role, email-verification and MFA status; no password hashes.
products.json	Product names, versions, descriptions and activity status.
licenses.json	Encrypted key values and IVs, lookup hashes, IDs, product references, activation counts, expiry, license controls, payment boundaries, notes and tags. No plaintext keys or passphrase.
activations.json	Activation references, timestamps, IP addresses, location, user agent, device identity, software version, user email and reactivation flag.
audit_logs.json	Audit events from the last 90 days.
api_keys.json	Names, prefixes, activity and expiry metadata; no raw API keys or hashes.
customers.json	Customer profiles, addresses and verification status; no passwords or sign-in tokens.
store_*.json, orders.json	Catalog, prices/history, design, slides/image references, orders, Stripe subscriptions and tax invoices. Includes original provider/payment IDs, money, tax, refund and commission values; no order plaintext license values.
provider_*.json	Provider connection metadata, product mappings, subscription/payment receipts and event status. No API/signing credentials or raw provider inbox bodies.
bundles.json, bundle_items.json, license_templates.json	License configuration and product relationships.
meter_events.json, meter_prices.json, license_meter_config.json	Usage history and provider metering configuration.

File	Contents
roles.json, role_permissions.json, reseller_access.json	Tenant-owned roles, permissions and reseller/commission configuration.
webhook_endpoints.json, *_metadata.json	Webhook configuration and delivery status, floating sessions, SSO, offline activation and self-host records. Session tokens, webhook secrets, payloads/errors and SSO secrets are excluded.
signing_public_keys.json	Historical public verification keys, key IDs and status. No private signing keys.
billing_dunning.json, activation_topups.json, erasure_requests.json	Payment recovery, purchased quota and privacy-request records.
platform_paddle_checkouts.json, platform_paddle_events.json	Own platform checkout snapshots and event-processing metadata, including original provider references. No global provider credentials, catalog or raw inbox bodies.
manifest.json	Versioned export format and explicit coverage/exclusions. It identifies this ZIP as a selected account-data export rather than a database backup.
README.txt	Export date and a description of the archive.

Download and use the export

A Store-issued license's EncryptedKey can be a store-order: reference, not passphrase-decryptable ciphertext. The ZIP excludes order delivery material and server encryption secrets, so it cannot recover these Store key values on its own. Preserve delivered keys and arrange authorized recovery while retained material and the service remain available.

1. Sign in as a company admin with a verified email and open **Company Settings**.
2. Choose **Export My Data (ZIP)** and save the archive in a protected location.
3. Inspect the included files before using the archive for an access request or migration. Retain your license-key passphrase separately: ciphertext, IVs and salt cannot recover it on their own. The archive contains personal data and does not include every application feature's records.

Format version 3 lists every archive file in manifest.json, including platform Paddle history. Uploaded file bytes are not included: image/download URLs are references, while inline avatars remain in team records. Keep original assets and provider records separately. Raw event/delivery payloads, authentication and provider secrets, private signing keys and platform-global configuration are excluded. Reads are sequential, not a transactionally consistent database snapshot. Signing private keys require the separate acknowledged export in **Settings → Security → Business Continuity**. Follow the exit plan and test recovery before an outage. Exported lookup hashes are identifiers, not license values or a replacement for the server's global secrets.

The API equivalent is GET /api/gdpr/export with the signed-in admin's bearer token. API keys are not accepted by this endpoint. The export action is recorded in the audit log.

Account Recovery Package

The separate **Export Recovery Package (ZIP)** action in Company Settings is available to a company admin with a verified email on every plan. It captures the account records together in a PostgreSQL repeatable-read transaction, adds available original Store license values, and packages the stored bytes of inline images. The ordinary account-data ZIP above keeps its existing scope.

Prepare and download

The current recovery helper accepts exact versions 2, 3, 4 and 5 account-data inventories. Restored platform checkouts are closed, inbox events are archived without replay, and operational billing connections are cleared. Version 4 also preserves recorded last-sign-in and device last-seen timestamps; legacy versions leave missing observations unknown. Recovery never starts a new introductory trial or changes an external provider subscription; review billing separately before reopening.

1. First export the active signing key from **Settings → Security → Business Continuity** and keep your original license-key encryption passphrase separately.
2. Open **Company Settings → Data & Privacy → Recovery package**. Read and select the confidentiality acknowledgement, then choose **Export Recovery Package (ZIP)**.
3. Store the ZIP, signing-key file and passphrase in protected backup storage. **The ZIP itself is not encrypted.** Anyone with it can read the included customer data and available Store license values.
4. Read manifest.json and recovery/README.txt. A warning identifies historical Store values that were never retained or whose stored delivery material is invalid. Export does not invent or replace a customer key.
5. Arrange a trial recovery with your operator in a new isolated installation before depending on the package.

Files and limits

Location	What it provides
data/	The 47 allowlisted record datasets plus manifest.json and README.txt (49 files), captured inside the recovery transaction.
recovery/store-keys.json	Available original Store key values with their license/order IDs, or an explicit missing/invalid-delivery status.
assets/, recovery/assets.json	Hashed binary copies of stored inline images. Uploaded avatars are the stored resized version, not the original pre-resize file.
recovery/external-assets.json	External image/download references. These URLs are not fetched or represented as backed-up files.
recovery/*-metadata.json, recovery/audit-full.json	Key-prefix and active-device-capacity details plus all

Location	What it provides
	available audit metadata; raw audit detail bodies remain excluded.
manifest.json	Format version, entry sizes, SHA-256 checksums, key coverage and recovery requirements. Checksums detect corruption; they are not an independent authenticity signature.

The package excludes platform master encryption/HMAC keys, passwords, MFA and sessions, provider/webhook credentials, raw event bodies and private signing keys. Keep the separately exported active signing key. Recovery is limited to currently retained records and original values that still exist. The maximum uncompressed package is 128 MiB; an over-limit snapshot fails rather than silently truncating records.

Restore in an isolated destination

Your operator initializes the same compatible PermitCore release in an empty PostgreSQL database named permitcore_recovery_*, with a separate Valkey and the target's own encryption/HMAC configuration. The API image includes the operator helper at /app/recovery/PermitCore.Recovery.dll. Backend self-hosting entitlement is separate from exporting your data.

Operator CLI

```
dotnet /app/recovery/PermitCore.Recovery.dll inspect --archive /private/recovery.zip
dotnet /app/recovery/PermitCore.Recovery.dll restore --archive /private/recovery.zip --source-passphrase-file /private/passphrase.txt --signing-key-file /private/signing-key.json
```

The helper uses PERMITCORE_RECOVERY_CONNECTION, PERMITCORE_RECOVERY_VALKEY, PERMITCORE_RECOVERY_FIELD_KEY and PERMITCORE_RECOVERY_HMAC_SECRET from the target's protected configuration. Secret values belong in protected files/environment, not command-line arguments or logs. It refuses an existing tenant or a normal application database.

Original Store values are encrypted again with the target's field key and rehashed with its HMAC key. Browser-created values require the original passphrase for authenticated decryption and rehashing. Active signing trust requires the matching separate signing-key file. A wrong passphrase, mismatched signing key, invalid relationship or checksum aborts the transaction.

Missing values make a default restore fail. An operator can explicitly use --allow-unavailable-values for an incomplete restore; those licenses stay disabled and the command returns exit code 3. This is not a complete recovery of those key values.

Review before reopening

- Sign in using the target installation's fresh operator account. Restored team members need new authentication setup; old passwords, MFA and sessions are not copied.
- Review the restore report. The Store, payment connections/mappings, SSO and outbound webhooks require reconfiguration. Old billing reminder and event replay schedules are stopped;

floating clients must check out new leases.

- Reconcile platform billing and activation add-ons. Recovery does not change or cancel subscriptions held by an external payment provider.
- A historical order may refer to a reseller outside the exported account. That nullable user link is disconnected on restore; its original ID remains in the ZIP and restore report. Commission amounts remain unchanged and require reconciliation. Recovery does not copy another account's user.
- Test original license values, trusted proof, product/customer/order relationships, monetary values and images before opening checkout. Keep external asset files with their own host's backup.

Stored image bytes remain in restored account records. An operator can also use `extract-assets --archive /private/recovery.zip --destination /private/new-assets` to save binary files into a new private directory without overwriting existing files.

Overview

Choose an integration path

Use an official SDK for license validation, activation, hardware-ID helpers where available, signed offline caching, and floating sessions. Use HTTP directly when your language is not supported or you need control over transport.

SDK constructors take the API origin, `https://api.permitcore.dev`, without `/api/v1`. Self-hosted deployments use their own API origin.

API Overview

API base and authentication

```
https://api.permitcore.dev/api/v1
```

Download Postman Collection

Operation	Credential	Use
POST /validate	License key in JSON body	Check entitlement without consuming a seat.
POST /activate	License key plus fresh nonce	Register a new installation; same-device node-locked reactivation reuses the seat.
GET /nonce	No credential	Obtain the single-use activation token.
/float/*	License key for checkout; session token for heartbeat/checkin	Acquire, maintain, and release a concurrent seat.
POST /meter	License key in body	Record usage. Keep event names and quantities consistent.
Management API (/api/*)	Bearer token or supported X-Api-Key authentication	Tenant administration from a trusted server.

Public licensing endpoints require no admin API key. The license key is still a sensitive credential: use HTTPS, keep it out of URLs, and exclude it from logs. Admin API keys have broad tenant-management access and must never be shipped inside customer applications. Review the reachable-endpoint list in the API Keys screen before using one.

Responses and limits

Require `isValid: true` before granting access; HTTP 200 can carry a license rejection. Verify that the product/bundle and features match your application. Per-IP and abuse controls apply in addition to your plan's tenant limit. Handle HTTP 429 and honor `Retry-After` when supplied. See error codes and bounded retries.

POST /validate

POST /api/v1/validate

Subscription renewals & past_due

For monthly and annual Store purchases, a paid Stripe subscription invoice extends the existing license to the end of its paid billing period. The customer keeps the same key. expiresAt is the paid-through date, not the grace deadline.

A failed renewal sets paymentStatus to past_due. Validation and activation continue through the configured payment grace window: 14 days by default, measured from the paid-through date. Vendors can set 0–365 days in Store → Settings; a license's explicit PostExpiryGraceDays takes precedence, including zero. unpaid keeps the same deadline. Once it is reached, access expires. A successful payment restores active and advances the paid-through date.

During payment grace, the response includes isInGracePeriod: true and paymentStatus: "past_due" or "unpaid". Show a card-update notice while isValid remains true. Non-subscription licenses return no payment state. Offline cache tokens and newly generated offline activation tokens for Store subscriptions are capped at paid-through plus grace. Reconnect to receive a refreshed signed cache token after payment; already-issued offline tokens retain their signed deadline and cannot learn about a cancellation while disconnected.

Scheduling cancellation at period end preserves access through that paid period. When Stripe ends the subscription, customer.subscription.deleted revokes the license. Duplicate invoices and webhook deliveries have no additional effect; older payments never shorten the paid period or reactivate a revoked key.

Register invoice.paid, invoice.payment_failed, and customer.subscription.updated alongside checkout.session.completed, payment_intent.payment_failed, checkout.session.expired, customer.subscription.deleted, and charge.refunded on the vendor's Store webhook endpoint. This is the customer's Store subscription; the vendor's own PermitCore billing is managed separately.

Checks whether a license key is valid. Returns license info but does **not** increment any counter. Use this on every app launch to confirm the key is still active. The key is sent in the POST body (never the URL) to keep it out of access logs and browser history.

Request fields

licenseKey is required. Optional version contains the running app version; send it consistently when using version restrictions, since omitting it skips the version check.

Validation checks license entitlement, not proof that the current machine owns a registered activation seat.

> **Check the product, or pass expectedProductId:** a valid, active key from your tenant validates successfully *even if it was issued for a completely different product* — /validate finds a key purely by the key itself. Either compare the response's productId against your own product's ID yourself, or pass an optional

> expectedProductId (GUID) in the request and let the server reject a mismatch for you with errorCode: "WrongProduct". Find your product's ID in the Admin panel under Products, or on a license's own detail page.

Optional deviceId supplies context for the signed offline proof; validation never allocates a seat. New grace tokens sign licenseKeySha256, the checked productId, deviceId, nodeLocked and deviceActivated from the actual activation records. The existing opaque licenseKeyHash remains a server HMAC identifier. An already-issued offline proof retains its signed deadline after later device removal. SDK 2 trust migration.

Examples

cURL

```
curl -X POST https://api.permitcore.dev/api/v1/validate \
  -H "Content-Type: application/json" \
  -d '{"licenseKey":"PERMIT-A3K7-MN2P-QR9X-
TZ5W","expectedProductId":"7f3e9a2b-1c4d-4e8f-9a6b-3d2c1e0f5a8b"}'
```

JavaScript

```
const res = await fetch('https://api.permitcore.dev/api/v1/validate', {
  method: 'POST',
  headers: { 'Content-Type': 'application/json' },
  body: JSON.stringify({
    licenseKey: 'PERMIT-A3K7-MN2P-QR9X-TZ5W',
    // Optional — reject a key issued for a different product instead of silently accepting it
    expectedProductId: '7f3e9a2b-1c4d-4e8f-9a6b-3d2c1e0f5a8b',
  }),
});
const data = await res.json();

if (data.isValid) {
  console.log('License OK for:', data.productName, data.productId);
  // data.remainingActivations tells you how many more activations are allowed
} else if (data.errorCode === 'WrongProduct') {
  console.error('This key is for a different product:', data.productId);
} else {
  console.error('Invalid license:', data.message);
  // Show error to user
}
```

Python

```
import requests

resp = requests.post(
    'https://api.permitcore.dev/api/v1/validate',
    json={
        'licenseKey': 'PERMIT-A3K7-MN2P-QR9X-TZ5W',
        # Optional — reject a key issued for a different product instead of silently accepting it
        'expectedProductId': '7f3e9a2b-1c4d-4e8f-9a6b-3d2c1e0f5a8b',
    }
```

```
)
data = resp.json()

if data['isValid']:
    print(f"Valid! Product: {data['productName']} ({data['productId']}), "
          f"Remaining activations: {data['remainingActivations']}")
elif data.get('errorCode') == 'WrongProduct':
    print(f"This key is for a different product: {data['productId']}")
else:
    print(f"Invalid: {data.get('message', 'unknown error')}")
```

C#

```
using var http = new HttpClient();
var resp = await http.PostAsJsonAsync(
    "https://api.permitcore.dev/api/v1/validate",
    new {
        licenseKey = "PERMIT-A3K7-MN2P-QR9X-TZ5W",
        // Optional — reject a key issued for a different product instead of silently accepting it
        expectedProductId = "7f3e9a2b-1c4d-4e8f-9a6b-3d2c1e0f5a8b",
    }
);
var result = await resp.Content.ReadFromJsonAsync<LicenseResult>();

if (result?.IsValid == true)
    Console.WriteLine($"Valid! Product: {result.ProductName} ({result.ProductId})");
else if (result?.ErrorCode == "WrongProduct")
    Console.WriteLine($"This key is for a different product: {result.ProductId}");
else
    Console.WriteLine($"Invalid: {result?.Message}");

record LicenseResult(
    bool IsValid,
    string? ProductName,
    Guid? ProductId,
    int? RemainingActivations,
    DateTime? ExpiresAt,
    string? Message,
    string? ErrorCode,
    Dictionary<string, string>? CustomFields
);
```

Success response (200 OK)

JSON


```
{
  "isValid": true,
  "productName": "Acme Video Editor Pro",
  "productId": "7f3e9a2b-1c4d-4e8f-9a6b-3d2c1e0f5a8b",
  "remainingActivations": 3,
  "expiresAt": null,
  "customFields": { "edition": "Pro", "seats": "5" },
  "features": ["export", "api", "sso"],
  "isTrial": false,
```

```
  "trialDaysRemaining": null,
  "nodeLocked": false,
  "offlineGraceDays": 7,
  "vendorWarning": null
}
```

Invalid / revoked key

```
JSON
{
  "isValid": false,
  "message": "License key not found",
  "errorCode": "NotFound",
  "productId": null
}
```

Additional fields include errorCode, minVersion, maxVersion, bundleProducts, isInGracePeriod, maintenanceExpiresAt, and signed offlineCacheToken. These may be null when not applicable. A maintenance date is metadata for your app's update policy, not an automatic license rejection.

Response fields

Field	Type	Description
isValid	boolean	true if the key exists, is active, not expired, and not revoked
productName	string	The product this key belongs to
productId	string (GUID) | null	Stable ID of the product this key belongs to. Always present when a license was found (even if you didn't pass expectedProductId) — compare it yourself, or use expectedProductId and let the server do it.
remainingActivations	number	maxActivations minus currentActivations — how many more devices can activate
expiresAt	string | null	ISO 8601 expiry datetime, or null for perpetual licenses
customFields	object | null	Arbitrary key-value pairs attached to the license
features	string[] | null	Feature flag strings e.g. ["export","api","sso"]. Use these to conditionally unlock functionality in your app without issuing separate

Field	Type	Description
		keys.
isTrial	boolean	true if this is a time-limited trial license
trialDaysRemaining	number | null	Days left on the trial period. Only present when isTrial is true. Show this to the customer to prompt them to purchase.
nodeLocked	boolean	true if the license is bound to specific devices (HWID mode)
offlineGraceDays	number | null	How many days your SDK may run without contacting the server. Use the signed offlineCacheToken with a trusted public key; do not trust an editable cached JSON response. null = no grace period.
message	string	Reason for invalidity — only present when isValid = false
vendorWarning	string | null	Present when the vendor's PermitCore subscription is nearing expiry. Surface this to the user. null during normal operation.

> **Always check vendorWarning in your app:** When this field is non-null, show it to the user (e.g. in a settings dialog or tooltip). It tells them the vendor's subscription is expiring and they may lose access soon — prompting them to contact the vendor to renew.

GET /nonce

GET /api/v1/nonce

Fetch a fresh nonce immediately before each raw HTTP activation. It expires after five minutes and is single-use. Official SDK activation methods obtain it for you.

```
curl https://api.permitcore.dev/api/v1/nonce
```

```
{ "nonce": "a3f82b1c9e4d7f06bc2e5a1d3c8f7e92" }
```

Include the returned value as `nonce` in the activation JSON. A missing nonce returns 400; an invalid, expired, or reused nonce returns 422. Never cache a nonce for future launches. After an uncertain activation timeout, remember the server may already have consumed the nonce and committed the activation.

POST /activate

POST /api/v1/activate

Store subscription payment state and grace use the same rules as validation, including paymentStatus and isInGracePeriod.

Validates the key and registers an installation. A new count-based activation consumes a seat; node-locked reactivation with the same device ID reuses its existing seat. Use this only when a customer "registers" your software on a specific machine for the first time. Once the key's max activations are reached, this endpoint returns an error.

> **Nonce required:** You must call GET /api/v1/nonce first and include the returned value as the nonce field. Nonces are single-use and expire in 5 minutes.

Request body fields

The examples use placeholder license keys, product IDs and email addresses. Supply your own application values; copying a sample does not issue a license. userEmail is optional and is only needed if you want named-user auditing.

Field	Type	Required	Description
licenseKey	string	Yes	The plaintext license key (PERMIT-XXXX-XXXX-XXXX-XXXX format)
nonce	string	Yes	Single-use token from GET /api/v1/nonce — prevents replay attacks
deviceId	string	No	SHA-256 hardware fingerprint — required for node-locked licenses
deviceName	string	No	Human-readable device label (e.g. "John-MacBook-Pro"), shown in admin panel
version	string	No	Your app version (e.g. "2.3.1") — validated against MinVersion/MaxVersion on the license
userEmail	string	No	End-user's email address — stored for B2B named-user audit trail, visible in admin panel

Field	Type	Required	Description
expectedProductId	string (GUID)	No	Reject (errorCode: "WrongProduct") unless this key belongs to this exact product. Without it, any active key for the caller's tenant activates successfully regardless of which product it was issued for. Find your product's ID in the Admin panel under Products.

JSON Body — full example

```
{
  "licenseKey":        "PERMIT-A3K7-MN2P-QR9X-TZ5W",
  "nonce":             "a3f82b1c9e4d7f06bc2e5a1d3c8f7e92",
  "deviceId":          "a3f8d2b1c9e4...",
  "deviceName":        "John-MacBook-Pro-M3",
  "version":           "2.3.1",
  "userEmail":         "john.smith@acme.com",
  "expectedProductId":  "7f3e9a2b-1c4d-4e8f-9a6b-3d2c1e0f5a8b"
}
```

cURL — two-step flow (fetch nonce then activate)

```
# Step 1: get a nonce
NONCE=$(curl -s https://api.permitcore.dev/api/v1/nonce | python3 -c "import sys,json; print(json.load(sys.stdin)
['nonce'])")

# Step 2: activate (expectedProductId is optional — see the field table above)
curl -X POST https://api.permitcore.dev/api/v1/activate \
  -H "Content-Type: application/json" \
  -d "{\"licenseKey\": \"PERMIT-A3K7-MN2P-QR9X-TZ5W\", \"nonce\": \"$NONCE\", \"expectedProductId\":
\"7f3e9a2b-1c4d-4e8f-9a6b-3d2c1e0f5a8b\"}"
```

Success response (200 OK)

JSON

```
{
  "isValid": true,
  "productName": "Acme Video Editor Pro",
  "productId": "7f3e9a2b-1c4d-4e8f-9a6b-3d2c1e0f5a8b",
  "remainingActivations": 2,
  "expiresAt": null,
  "features": ["export", "api"],
  "isTrial": false,
  "nodeLocked": true,
  "offlineGraceDays": 7
}
```

Max activations reached (200, isValid: false)

```
JSON
{
  "isValid": false,
  "message": "Maximum activations reached for this license",
  "errorCode": "SeatsExhausted"
}
```

Wrong product (200, isValid: false) — only when expectedProductId is sent

```
JSON
{
  "isValid": false,
  "message": "This license does not cover the requested product.",
  "errorCode": "WrongProduct",
  "productId": "a1b2c3d4-5678-90ab-cdef-1234567890ab"
}
```

> **Activation is atomic:** The counter is incremented using a database-level atomic operation. You don't need to worry about race conditions even if multiple machines try to activate the same key simultaneously.

Error codes

HTTP status and license status

Read the license-error UX guide. Use machine-readable decisions, a safe generic branch and a customer recovery path without turning rejection into access.

HTTP status tells you whether the request was processed. The isValid field tells you whether to allow licensed functionality. Check both. These rules describe public validate and activate requests; admin endpoints have separate authentication requirements.

Status	Meaning	Next step
200	License decision returned. Missing, revoked, expired, blocked, or exhausted licenses can return isValid: false.	Require isValid === true before unlocking features.
400	Invalid request or missing activation nonce.	Check fields and key format. Fetch a nonce before activation.
422	The supplied activation nonce is expired, invalid, or already used.	Fetch a fresh nonce.
429	An IP, tenant, or license abuse limit was reached.	Honor Retry-After when present and use bounded backoff.
5xx / timeout	No reliable license decision is available.	Retry validation a limited number of times. Offline access requires a verified signed token.

Error bodies are not always license response objects: request errors may be text or problem details, and a proxy may return HTML. Do not assume every non-200 response is JSON.

Machine-readable rejection reasons

Use errorCode when supplied. Keep message for display, not string matching. Some rejection paths return no error code, so retain a generic invalid-license branch.

Code	Meaning	Action
NotFound	No active matching license	Check the key or contact the vendor.
ProductInactive	The product is inactive	Vendor checks product status.
Expired	Expiry and any post-expiry grace have elapsed	Renew the license.
VersionBlocked	App version is outside the configured range	Use an entitled version or update the license.

Code	Meaning	Action
IpBlocked / RegionBlocked	Network restriction rejected the request	Check allowlists and the customer's network.
VendorSubscriptionLapsed	Vendor subscription needs attention	Vendor reviews Billing.
ActiveDeviceLimitReached	Total account active device capacity reached	Deactivate a device, add active top-up capacity or upgrade. Existing devices keep normal validity rules.
PoolExhausted	Deprecated	No longer returned; handle ActiveDeviceLimitReached.
SeatsExhausted	No seats available on the key	Remove an unused device or increase the key's limit.
WrongProduct	The key is valid but belongs to a different product than the expectedProductId you passed	Compare against the returned productId, or omit expectedProductId and check it yourself.

A bounded validation request

This JavaScript example makes at most three attempts and never grants access on connection failure. After a valid response, check your product and feature entitlements. For signed offline caching, use an official SDK.

```javascript
async function validateLicense(licenseKey) {
  const sleep = ms => new Promise(resolve => setTimeout(resolve, ms));
  for (let attempt = 0; attempt < 3; attempt++) {
    try {
      const res = await fetch('https://api.permitcore.dev/api/v1/validate', {
        method: 'POST',
        headers: { 'Content-Type': 'application/json' },
        body: JSON.stringify({ licenseKey, version: '2.3.1', expectedProductId:
'7f3e9a2b-1c4d-4e8f-9a6b-3d2c1e0f5a8b' }), // expectedProductId is optional — reject a key issued for a different
product instead of silently accepting it
        signal: AbortSignal.timeout(10000)
      });
      if ((res.status === 429 || res.status >= 500) && attempt < 2) {
        const header = res.headers.get('Retry-After');
        const seconds = header === null ? NaN : Number(header);
        const delay = Number.isFinite(seconds) ? seconds * 1000
          : header ? Date.parse(header) - Date.now() : NaN;
        if (Number.isFinite(delay) && delay > 30000)
          return { isValid: false, unavailable: true, retryAfter: header };
        await sleep(Number.isFinite(delay) ? Math.max(0, delay) : 1000 * 2 ** attempt);
        continue;
      }
      if (!res.ok) return { isValid: false, httpStatus: res.status };
      const data = await res.json();
```

```
      if (typeof data?.isValid !== 'boolean')
        return { isValid: false, unavailable: true };
      return data;
    } catch {
      if (attempt < 2) { await sleep(1000 * 2 ** attempt); continue; }
    }
  }
  return { isValid: false, unavailable: true, message: 'License service unavailable.' };
}
```

Activation retries need extra care

A timeout does not prove activation failed: the server may have committed it before the connection was lost. Blindly retrying count-based activation can consume another seat. Persist successful installation state, use a stable device ID for node-locked licenses, and fetch a new nonce for each new request. Reconcile uncertain results instead of looping indefinitely.

Offline access is a separate decision

A network failure is not a valid license. Offline access requires a previously issued signed token, a trusted public key, a valid expiry, and matching license/device binding where applicable. A cached JSON success or local “activated” flag alone is insufficient. Do not replace an explicit online rejection with an older cached success.

SDK Overview

Official SDKs

The SDK application and Store tutorials show current installation commands and application fragments. Supply inputs and UI/host control flow from your app. Online validation alone does not enable offline grace, register a device or reserve a floating session. Configure trusted public keys for offline use; C++ installation needs its documented vcpkg registry and pinned baseline.

SDK request settings

A slow or unavailable license server must not leave your application waiting indefinitely. All nine SDKs default to a 10-second timeout for each attempt and two additional attempts for safe requests. The usual backoff is 250 milliseconds, then 500 milliseconds. This is not a single 10-second deadline for the complete operation: activation also fetches a nonce.

- License validation, nonce/public-key retrieval and floating heartbeat may retry a network failure or HTTP 5xx response. Set the retry count to zero to disable these retries; the range is 0–5.
- Floating checkout, activation POST, usage reporting, device removal and checkin are sent once. A lost response can follow a successful server change, so retrying these blindly can duplicate work.
- HTTP 3xx/4xx responses and malformed successful responses do not retry. SDK-owned transports do not follow redirects, so configure the final HTTPS API address directly.

For validate/activate, branch on the server's error code when available. Otherwise HttpError means an HTTP rejection, InvalidResponse means malformed successful license/nonce evidence and RequestFailed means the network/5xx attempts were exhausted without a usable signed cache. Rejection or malformed successful evidence never becomes cached authorization. A network outage can use only a separately trusted, unexpired proof.

SDK	Configuration
.NET	PermitCoreOptions.Timeout = TimeSpan.FromSeconds(10) and MaxRetries = 2
Node.js	Use the options-object constructor with timeoutMs: 10000, maxRetries: 2.
Python	PermitCoreClient(url, timeout=10, max_retries=2)
Java	The extended constructor takes trusted keys, cache path, device ID, Duration.ofSeconds(10), then 2.
C++	The constructor takes URL, 10 timeout seconds, cache toggle, trusted keys, cache path, device ID, then 2 retries.
PHP	The constructor takes URL, cache toggle, 10 timeout seconds, trusted keys, cache path, device ID, then 2 retries.
Go	Options.Timeout uses time.Duration. MaxRetries is an

SDK	Configuration
	optional integer pointer: nil defaults to two; a pointer to zero disables retries.
Unity	Named constructor arguments timeoutSeconds: 10, maxRetries: 2; custom transports must honor the supplied timeout and preserve any received status.
Godot	Constructor options "timeout_seconds": 10.0, "max_retries": 2. Native frame-polled HTTP keeps the game loop responsive, including while game time is paused.

Timeout mechanisms follow each runtime: Python uses socket I/O timeouts; Unity delegates to its transport. Caller cancellation is separate from retry policy. Unity's default adapter disables redirects; WebGL needs Unity 2021.3 or later for that protection and fails closed on older versions.
Portable .NET tests do not certify an Editor, IL2CPP or browser build. Verify your actual target before shipping.

Install the published version for your language using the SDK installation references and follow the offline migration guide when upgrading from 1.x.

PermitCore provides official SDK packages for .NET, Node.js, Python, Java, C++, PHP, Go, Unity and Godot. Each SDK wraps the public REST API with idiomatic language bindings, automatic HWID fingerprinting and signed offline grace period caching. Floating checkout, heartbeat and checkin are available in all nine clients. Earlier Java/C++ releases use the REST endpoints for floating sessions.

SDK 2: All nine SDKs share the operations below. Their validate/activate calls treat HTTP rejections as authoritative: a 4xx response cannot become successful offline validation. Network/5xx outages can still use a verified signed cache within its deadline. Use version 2.1.1 for these examples and follow the offline migration guide when upgrading from 1.x.

In SDK 2.1.1, floating checkout/heartbeat failures return success=false with no token or expiry: RequestFailed for network/HTTP/invalid responses, or the server denial code with FloatingDenied as the default. Floating requests never use offline cache. TryCheckin (idiomatic spelling varies by language) confirms only HTTP 204; legacy checkin is best effort and relies on server expiry if release is unconfirmed. Only heartbeat uses safe bounded network/5xx retries; checkout and checkin are sent once.

SDK 2.1.1 provides device removal in every client: supply tenant slug, license key and activation ID. Only HTTP 204 confirms removal. OTP/account Portal protection still requires the customer Portal flow. Each client attempts to discard that key's grace cache; separately issued offline tokens retain their signed bounds. Usage reporting accepts an integer quantity (default 1; values below 1 normalize to 1) and optional nested metadata object. Only JSON recorded: true confirms recording. Neither operation uses offline fallback or automatic retries. A false result after a lost response does not prove no server-side change occurred.

Current capability matrix

All nine clients support validate, activate, product binding, feature checks, offline token verification and signed grace caches. The table covers the eight SDK 2.1.1 packages and Godot 2.2.0. Device removal and boolean usage reporting have wrappers in all nine. Shared transport defaults, safe retries and core error handling are covered below; optional DTO edges remain runtime-specific.

SDK	Floating seats	Meter reporting	Device deactivation
.NET	SDK	SDK	SDK
Node.js	SDK	SDK	SDK
Python	SDK	SDK	SDK
Java	SDK	SDK	SDK
C++	SDK	SDK	SDK
PHP	SDK	SDK	SDK
Go	SDK	SDK	SDK
Unity	SDK	SDK	SDK
Godot	SDK	SDK	SDK

SDK means the operation has a wrapper in the published version listed above. Java/C++ retain legacy detailed meter results and add canonical boolean tryMeter/try_meter calls; other clients use MeterAsync/meter/Meter. All nine use 10-second defaults and two safe retries; follow the configuration guide for idiomatic options. Source ZIPs include test fixtures. .NET source/tests require the .NET 10 SDK even when your application targets .NET 6–9. Unity portable-core tests do not certify Editor or IL2CPP builds.

Godot
GDScript · Godot 4.7+, native HTTP/TLS and signed offline verification
.NET
PermitCore.Client
— NuGet · .NET 6–10, zero dependencies
Node.js
permitcore
— npm · TypeScript-first, native fetch
Python
permitcore
— pip · Python 3.8+, standard library online; cryptography for offline verification
Java
Maven · Java 11+, built-in
HttpClient
C++
CMake 3.14+ · C++17, libcurl, OpenSSL; nlohmann/json 3.11.3+
PHP
Composer · PHP 8.0+, ext-curl, ext-json, and ext-openssl for offline verification

Go
Go 1.21+ · standard library only, crypto/ecdsa for offline verification
Unity
Unity 2021.2+ · zero dependencies, Desktop/Mobile/Console/WebGL

> **All SDKs are built on top of the public REST API.** If your language isn't listed, use the REST API directly — the Admin snippet generator includes HTTP examples and a native Godot SDK example.

> **Want to see it running before you write any code?** Download a full working demo app (.NET, Node.js, Python, PHP, Java, C++, Go, Unity or Godot) that gates itself behind a real license — try it in the Sandbox.

.NET SDK

Device and usage operations in SDK 2.1.1: all clients can request key-only device removal using tenant slug, key and activation ID. HTTP 204 confirms removal; protected portals still require OTP/ account customer flows. Usage reporting returns boolean confirmation, supports nested/empty metadata objects and normalizes quantities below 1 to 1. Java/C++ use tryMeter/try_meter for that boolean contract and retain legacy detailed meter results. Neither operation retries automatically or uses offline cache. See this SDK's README for signatures and failure handling.

Floating sessions in SDK 2.1.1: checkout and heartbeat return a failed session without credentials on network/HTTP/parse failure (RequestFailed). Business denial preserves its error code or uses FloatingDenied. Floating never uses offline cache. The new boolean TryCheckin equivalent confirms HTTP 204; legacy checkin is best effort. Method names follow this language's conventions. See the source README and SDK availability matrix before choosing a registry version.

Official SDKs

SDK request settings

A slow or unavailable license server must not leave your application waiting indefinitely. All nine SDKs default to a 10-second timeout for each attempt and two additional attempts for safe requests. The usual backoff is 250 milliseconds, then 500 milliseconds. This is not a single 10-second deadline for the complete operation: activation also fetches a nonce.

- License validation, nonce/public-key retrieval and floating heartbeat may retry a network failure or HTTP 5xx response. Set the retry count to zero to disable these retries; the range is 0–5.
- Floating checkout, activation POST, usage reporting, device removal and checkin are sent once. A lost response can follow a successful server change, so retrying these blindly can duplicate work.
- HTTP 3xx/4xx responses and malformed successful responses do not retry. SDK-owned transports do not follow redirects, so configure the final HTTPS API address directly.

For validate/activate, branch on the server's error code when available. Otherwise HttpError means an HTTP rejection, InvalidResponse means malformed successful license/nonce evidence and RequestFailed means the network/5xx attempts were exhausted without a usable signed cache. Rejection or malformed successful evidence never becomes cached authorization. A network outage can use only a separately trusted, unexpired proof.

SDK	Configuration
.NET	PermitCoreOptions.Timeout = TimeSpan.FromSeconds(10) and MaxRetries = 2

Timeout mechanisms follow each runtime: Python uses socket I/O timeouts; Unity delegates to its transport. Caller cancellation is separate from retry policy. Unity's default adapter disables redirects; WebGL needs Unity 2021.3 or later for that protection and fails closed on older versions.
Portable .NET tests do not certify an Editor, IL2CPP or browser build. Verify your actual target before shipping.

Install the published version for your language using the SDK installation references and follow the

offline migration guide when upgrading from 1.x.

The official .NET client targets .NET 6, 7, 8, 9, and 10. Zero dependencies.

Installation

Package Manager

```
dotnet add package PermitCore.Client --version 2.1.1
```

Quick start — validate on every launch

C#

```
using PermitCore;

var client = new PermitCoreClient("https://api.permitcore.dev");
var result = await client.ValidateAsync(
    "PERMIT-XXXX-XXXX-XXXX-XXXX",
    expectedProductId: Guid.Parse("7f3e9a2b-1c4d-4e8f-9a6b-3d2c1e0f5a8b")); // Optional — reject a key issued for a different product instead of silently accepting it

if (result.ErrorCode == "WrongProduct")
{
    MessageBox.Show($"This key is for a different product: {result.ProductId}");
    Application.Exit();
    return;
}
if (!result.IsValid)
{
    MessageBox.Show($"License invalid: {result.Message}");
    Application.Exit();
    return;
}

// Unlock features based on flags
if (result.HasFeature("export")) EnableExportMenu();
if (result.IsTrial)
    ShowTrialBanner($"{result.TrialDaysRemaining} days remaining");

// Warn if vendor subscription is expiring
if (result.VendorWarning is not null)
    ShowStatusWarning(result.VendorWarning);
```

Activation (first install)

C#

```
// Call once when the user first installs on a new machine
var result = await client.ActivateAsync(
    licenseKey:  "PERMIT-XXXX-XXXX-XXXX-XXXX",
    deviceId:    PermitCoreClient.GetHardwareId(), // auto-generated HWID
    deviceName:  Environment.MachineName,
    expectedProductId: Guid.Parse("7f3e9a2b-1c4d-4e8f-9a6b-3d2c1e0f5a8b") // Optional — reject a key issued for a different product
```

```
);

if (result.IsValid)
    SaveActivationSuccess();
else if (result.ErrorCode == "WrongProduct")
    ShowError($"Wrong product: {result.ProductId}");
else
    ShowError($"Could not activate: {result.Message}");
```

Product scoping: by default, any active key belonging to your tenant validates/activates successfully regardless of which product it was issued for. If you sell more than one product, either check `result.ProductId` yourself, or pass the optional `expectedProductId` parameter and let the server reject a mismatch (`result.ErrorCode == "WrongProduct"`). Find a product's ID in the Admin panel under Products.

Offline mode

When the server is unreachable, the SDK automatically returns the last cached response for `offlineGraceDays` days. Enable/disable in options:

```
C#
var client = new PermitCoreClient(new PermitCoreOptions
{
    BaseUrl           = "https://api.permitcore.dev",
    EnableOfflineCache = true,  // default
});

var result = await client.ValidateAsync(key);
if (result.IsOffline)
    ShowStatusBar("Running in offline mode");
```

Floating licenses

Check out a seat when the session starts. While running, heartbeat about every four minutes and handle failed heartbeats. Stop using a rejected session. Cancel the heartbeat loop and await checkin during normal shutdown; process-exit callbacks may not finish network calls. See floating-session lifecycle.

Node.js SDK

Device and usage operations in SDK 2.1.1: all clients can request key-only device removal using tenant slug, key and activation ID. HTTP 204 confirms removal; protected portals still require OTP/account customer flows. Usage reporting returns boolean confirmation, supports nested/empty metadata objects and normalizes quantities below 1 to 1. Java/C++ use tryMeter/try_meter for that boolean contract and retain legacy detailed meter results. Neither operation retries automatically or uses offline cache. See this SDK's README for signatures and failure handling.

Floating sessions in SDK 2.1.1: checkout and heartbeat return a failed session without credentials on network/HTTP/parse failure (RequestFailed). Business denial preserves its error code or uses FloatingDenied. Floating never uses offline cache. The new boolean TryCheckin equivalent confirms HTTP 204; legacy checkin is best effort. Method names follow this language's conventions. See the source README and SDK availability matrix before choosing a registry version.

Official SDKs

SDK request settings

A slow or unavailable license server must not leave your application waiting indefinitely. All nine SDKs default to a 10-second timeout for each attempt and two additional attempts for safe requests. The usual backoff is 250 milliseconds, then 500 milliseconds. This is not a single 10-second deadline for the complete operation: activation also fetches a nonce.

- License validation, nonce/public-key retrieval and floating heartbeat may retry a network failure or HTTP 5xx response. Set the retry count to zero to disable these retries; the range is 0–5.
- Floating checkout, activation POST, usage reporting, device removal and checkin are sent once. A lost response can follow a successful server change, so retrying these blindly can duplicate work.
- HTTP 3xx/4xx responses and malformed successful responses do not retry. SDK-owned transports do not follow redirects, so configure the final HTTPS API address directly.

For validate/activate, branch on the server's error code when available. Otherwise HttpError means an HTTP rejection, InvalidResponse means malformed successful license/nonce evidence and RequestFailed means the network/5xx attempts were exhausted without a usable signed cache. Rejection or malformed successful evidence never becomes cached authorization. A network outage can use only a separately trusted, unexpired proof.

SDK	Configuration
Node.js	Use the options-object constructor with timeoutMs: 10000, maxRetries: 2.

Timeout mechanisms follow each runtime: Python uses socket I/O timeouts; Unity delegates to its transport. Caller cancellation is separate from retry policy. Unity's default adapter disables redirects; WebGL needs Unity 2021.3 or later for that protection and fails closed on older versions.
Portable .NET tests do not certify an Editor, IL2CPP or browser build. Verify your actual target before shipping.

Install the published version for your language using the SDK installation references and follow the

offline migration guide when upgrading from 1.x.

TypeScript-first. Works in a Node.js runtime with native fetch (Node.js 18+; use a supported LTS release). Uses the native fetch API — no dependencies.

Installation

npm

```
npm install @permitcore/permitcore@2.1.1
```

Quick start

TypeScript

```
import { PermitCoreClient } from '@permitcore/permitcore';

const client = new PermitCoreClient('https://api.permitcore.dev');
const result = await client.validate(
  'PERMIT-XXXX-XXXX-XXXX-XXXX',
  undefined, // version
  '7f3e9a2b-1c4d-4e8f-9a6b-3d2c1e0f5a8b' // expectedProductId — optional, reject a key issued for a different product
);

if (result.errorCode === 'WrongProduct') {
  console.error(`Wrong product: ${result.productId}`);
  process.exit(1);
}
if (!result.isValid) {
  console.error(`License invalid: ${result.message}`);
  process.exit(1);
}

// Feature flags
const features = new Set(result.features ?? []);
if (features.has('export')) enableExport();
if (features.has('api'))    enableApiAccess();

// Or use the static helper:
if (PermitCoreClient.hasFeature(result, 'export')) enableExport();
```

Activation

TypeScript

```
// Auto-collects HWID when deviceId is omitted
const result = await client.activate(
    'PERMIT-XXXX-XXXX-XXXX-XXXX',
    PermitCoreClient.getHardwareId(), // optional — auto-used if omitted
    'My workstation',                 // optional display name
    undefined,                        // version
    '7f3e9a2b-1c4d-4e8f-9a6b-3d2c1e0f5a8b' // expectedProductId — optional
);
```

```
if (result.errorCode === 'WrongProduct') {
  console.error(`Wrong product: ${result.productId}`);
}
```

Product scoping: by default, any active key belonging to your tenant validates/activates successfully regardless of which product it was issued for. If you sell more than one product, either check `result.productId` yourself, or pass the optional `expectedProductId` parameter and let the server reject a mismatch (`result.errorCode === 'WrongProduct'`). Find a product's ID in the Admin panel under Products.

Floating licenses

Check out a seat when the session starts. While running, heartbeat about every four minutes and handle failed heartbeats. Stop using a rejected session. Cancel the heartbeat loop and await checkin during normal shutdown; process-exit callbacks may not finish network calls. See floating-session lifecycle.

Hardware ID

```TypeScript
import { PermitCoreClient } from '@permitcore/permitcore';

const hwid = PermitCoreClient.getHardwareId();
// "a3f82c91b44e..." — stable SHA-256 of machine identifiers
```

Python SDK

Read the Python desktop release guide. Plan packaged application state, GUI responsiveness, offline trust and clean-target release checks. A source fragment or SDK check is not a packaged-app acceptance test.

Device and usage operations in SDK 2.1.1: all clients can request key-only device removal using tenant slug, key and activation ID. HTTP 204 confirms removal; protected portals still require OTP/account customer flows. Usage reporting returns boolean confirmation, supports nested/empty metadata objects and normalizes quantities below 1 to 1. Java/C++ use tryMeter/try_meter for that boolean contract and retain legacy detailed meter results. Neither operation retries automatically or uses offline cache. See this SDK's README for signatures and failure handling.

Floating sessions in SDK 2.1.1: checkout and heartbeat return a failed session without credentials on network/HTTP/parse failure (RequestFailed). Business denial preserves its error code or uses FloatingDenied. Floating never uses offline cache. The new boolean TryCheckin equivalent confirms HTTP 204; legacy checkin is best effort. Method names follow this language's conventions. See the source README and SDK availability matrix before choosing a registry version.

Official SDKs

SDK request settings

A slow or unavailable license server must not leave your application waiting indefinitely. All nine SDKs default to a 10-second timeout for each attempt and two additional attempts for safe requests. The usual backoff is 250 milliseconds, then 500 milliseconds. This is not a single 10-second deadline for the complete operation: activation also fetches a nonce.

- License validation, nonce/public-key retrieval and floating heartbeat may retry a network failure or HTTP 5xx response. Set the retry count to zero to disable these retries; the range is 0–5.
- Floating checkout, activation POST, usage reporting, device removal and checkin are sent once. A lost response can follow a successful server change, so retrying these blindly can duplicate work.
- HTTP 3xx/4xx responses and malformed successful responses do not retry. SDK-owned transports do not follow redirects, so configure the final HTTPS API address directly.

For validate/activate, branch on the server's error code when available. Otherwise HttpError means an HTTP rejection, InvalidResponse means malformed successful license/nonce evidence and RequestFailed means the network/5xx attempts were exhausted without a usable signed cache. Rejection or malformed successful evidence never becomes cached authorization. A network outage can use only a separately trusted, unexpired proof.

SDK	Configuration
Python	PermitCoreClient(url, timeout=10, max_retries=2)

Timeout mechanisms follow each runtime: Python uses socket I/O timeouts; Unity delegates to its transport. Caller cancellation is separate from retry policy. Unity's default adapter disables redirects; WebGL needs Unity 2021.3 or later for that protection and fails closed on older versions.

Portable .NET tests do not certify an Editor, IL2CPP or browser build. Verify your actual target before shipping.

Install the published version for your language using the SDK installation references and follow the offline migration guide when upgrading from 1.x.

Pure Python 3.8+. Basic online requests use the standard library. Install `permitcore[offline]==2.1.1` for signed offline/grace verification.

Installation

pip

```
pip install "permitcore[offline]==2.1.1"
```

Quick start

Python

```python
from permitcore import PermitCoreClient

client = PermitCoreClient("https://api.permitcore.dev")
result = client.validate(
    "PERMIT-XXXX-XXXX-XXXX-XXXX",
    expected_product_id="7f3e9a2b-1c4d-4e8f-9a6b-3d2c1e0f5a8b"  # Optional — reject a key issued for a different product instead of silently accepting it
)

if result.error_code == "WrongProduct":
    print(f"This key is for a different product: {result.product_id}")
    raise SystemExit(1)
if not result.is_valid:
    print(f"License invalid: {result.message}")
    raise SystemExit(1)

# Feature flags
if result.has_feature("export"):
    enable_export()
if result.is_trial:
    print(f"Trial — {result.trial_days_remaining} days remaining")

# Warn if vendor subscription expiring
if result.vendor_warning:
    show_warning(result.vendor_warning)
```

Activation

Python

```python
import socket
result = client.activate(
    "PERMIT-XXXX-XXXX-XXXX-XXXX",
    device_id   = PermitCoreClient.get_hardware_id(),  # auto if omitted
    device_name = socket.gethostname(),
```

```
    expected_product_id = "7f3e9a2b-1c4d-4e8f-9a6b-3d2c1e0f5a8b"  # optional
)
if result.is_valid:
    save_activated()
elif result.error_code == "WrongProduct":
    print(f"Wrong product: {result.product_id}")
```

Product scoping: by default, any active key belonging to your tenant validates/activates successfully regardless of which product it was issued for. If you sell more than one product, either check `result.product_id` yourself, or pass the optional `expected_product_id` parameter and let the server reject a mismatch (`result.error_code == "WrongProduct"`). Find a product's ID in the Admin panel under Products.

Offline mode

```
Python
# Cache is enabled by default — stored in ~/.permitcore_cache_{hash}
client = PermitCoreClient(
    "https://api.permitcore.dev",
    enable_offline_cache=True
)
result = client.validate(key)
if result.is_offline:
    print("Running in offline mode")
```

Floating licenses

Check out a seat when the session starts. While running, heartbeat about every four minutes and handle failed heartbeats. Stop using a rejected session. Cancel the heartbeat loop and await checkin during normal shutdown; process-exit callbacks may not finish network calls. See floating-session lifecycle.

Hardware ID

```
Python
hwid = PermitCoreClient.get_hardware_id()
# "a3f82c91b44e..." — stable SHA-256 of machine identifiers
```

Java SDK

Device and usage operations in SDK 2.1.1: all clients can request key-only device removal using tenant slug, key and activation ID. HTTP 204 confirms removal; protected portals still require OTP/ account customer flows. Usage reporting returns boolean confirmation, supports nested/empty metadata objects and normalizes quantities below 1 to 1. Java/C++ use tryMeter/try_meter for that boolean contract and retain legacy detailed meter results. Neither operation retries automatically or uses offline cache. See this SDK's README for signatures and failure handling.

Floating sessions in SDK 2.1.1: checkout and heartbeat return a failed session without credentials on network/HTTP/parse failure (RequestFailed). Business denial preserves its error code or uses FloatingDenied. Floating never uses offline cache. The new boolean TryCheckin equivalent confirms HTTP 204; legacy checkin is best effort. Method names follow this language's conventions. See the source README and SDK availability matrix before choosing a registry version.

Official SDKs

SDK request settings

A slow or unavailable license server must not leave your application waiting indefinitely. All nine SDKs default to a 10-second timeout for each attempt and two additional attempts for safe requests. The usual backoff is 250 milliseconds, then 500 milliseconds. This is not a single 10-second deadline for the complete operation: activation also fetches a nonce.

- License validation, nonce/public-key retrieval and floating heartbeat may retry a network failure or HTTP 5xx response. Set the retry count to zero to disable these retries; the range is 0–5.
- Floating checkout, activation POST, usage reporting, device removal and checkin are sent once. A lost response can follow a successful server change, so retrying these blindly can duplicate work.
- HTTP 3xx/4xx responses and malformed successful responses do not retry. SDK-owned transports do not follow redirects, so configure the final HTTPS API address directly.

For validate/activate, branch on the server's error code when available. Otherwise HttpError means an HTTP rejection, InvalidResponse means malformed successful license/nonce evidence and RequestFailed means the network/5xx attempts were exhausted without a usable signed cache. Rejection or malformed successful evidence never becomes cached authorization. A network outage can use only a separately trusted, unexpired proof.

SDK	Configuration
Java	The extended constructor takes trusted keys, cache path, device ID, Duration.ofSeconds(10), then 2.

Timeout mechanisms follow each runtime: Python uses socket I/O timeouts; Unity delegates to its transport. Caller cancellation is separate from retry policy. Unity's default adapter disables redirects; WebGL needs Unity 2021.3 or later for that protection and fails closed on older versions.
Portable .NET tests do not certify an Editor, IL2CPP or browser build. Verify your actual target before shipping.

Install the published version for your language using the SDK installation references and follow the

offline migration guide when upgrading from 1.x.

Pure Java 11+. Uses Java's built-in HttpClient — no Apache HttpClient or OkHttp required. JSON handled by Jackson (pulled automatically by Maven).

Download Java SDK (ZIP)

Installation

The package for this release is dev.permitcore:permitcore-sdk:2.1.1. Add the dependency below to use Maven Central. To build from the optional source ZIP instead:

```
Terminal
cd SDKs/java
mvn install
```

Then add to your pom.xml:

```
pom.xml
<dependency>
  <groupId>dev.permitcore</groupId>
  <artifactId>permitcore-sdk</artifactId>
  <version>2.1.1</version>
</dependency>
```

Quick start

```
Java
import dev.permitcore.PermitClient;

PermitClient client = new PermitClient("https://api.permitcore.dev");

PermitClient.ValidateResult result = client.validate("PERMIT-XXXX-XXXX-XXXX-XXXX", null,
    "7f3e9a2b-1c4d-4e8f-9a6b-3d2c1e0f5a8b"); // expectedProductId — optional, reject a key issued for a different product
if (result.isValid) {
    System.out.println("License OK — product: " + result.productName + " (" + result.productId + ")");
    // Unlock features
    if (result.features != null && result.features.contains("export"))
        enableExportMenu();
    if (result.isTrial)
        showTrialBanner(result.trialDaysRemaining + " days remaining");
    if (result.isOffline)
        System.out.println("Running on a cached result — server was unreachable.");
} else if ("WrongProduct".equals(result.errorCode)) {
    showLicenseError("This key is for a different product: " + result.productId);
} else {
    showLicenseError(result.message);
}
```

validate() and activate() never throw for a network failure — they fall back to the local offline-grace cache automatically and, failing that, return isValid: false with isOffline: true. Only meter() throws PermitException.

Validate with version check

```java
PermitClient.ValidateResult result = client.validate("PERMIT-XXXX-XXXX-XXXX-XXXX", "2.3.1");
if (!result.isValid && result.minVersion != null)
    showError("Please upgrade to v" + result.minVersion + " or newer.");
```

Activation

```java
var req = new PermitClient.ActivateRequest(
    "PERMIT-XXXX-XXXX-XXXX-XXXX",
    "sha256-of-hardware-fingerprint",  // deviceId
    InetAddress.getLocalHost().getHostName(), // deviceName
    "2.3.1",                          // version
    "7f3e9a2b-1c4d-4e8f-9a6b-3d2c1e0f5a8b" // expectedProductId — optional, reject a key issued for a different product
);
PermitClient.ValidateResult result = client.activate(req);
if (result.isValid)
    saveActivationSuccess();
else if ("WrongProduct".equals(result.errorCode))
    showError("Wrong product: " + result.productId);
```

Metered billing

```java
// Single event
PermitClient.MeterResult m = client.meter("PERMIT-XXXX-XXXX-XXXX-XXXX", "export");

// Batch — 5 API calls at once
var req = new PermitClient.MeterRequest("PERMIT-XXXX-XXXX-XXXX-XXXX", "api_call", 5, null);
PermitClient.MeterResult m = client.meter(req);
```

ValidateResult field reference

Field	Type	Description
isValid	boolean	true if the license is active and all checks pass
productName	String	Name of the product this license belongs to
remainingActivations	Integer	How many more activations are allowed (null = unlimited)
expiresAt	OffsetDateTime	License expiry in UTC (null = perpetual)

Field	Type	Description
message	String	Human-readable status message
customFields	Map<String, String>	Arbitrary key/value metadata set by the vendor on this license
vendorWarning	String	Non-null if the vendor's PermitCore subscription is lapsing — display a warning to your user
features	List<String>	Feature flags enabled on this license (e.g. ["export","api","premium"])
isTrial	boolean	true if this is a trial license
trialDaysRemaining	Integer	Days remaining in the trial (0 when not a trial)
nodeLocked	boolean	true if this license uses HWID node-locking
offlineGraceDays	Integer	Days the software may run without reaching the server
minVersion	String	Minimum allowed software version (semver), null if unrestricted
maxVersion	String	Maximum allowed software version (semver), null if unrestricted
isOffline	boolean	true when this result was served from the local offline-grace cache (server unreachable)
productId	String	Stable ID of the product this license belongs to. Always present when a license was found, even without passing expectedProductId on the request.

Product scoping: by default, any active key belonging to your tenant validates/activates successfully regardless of which product it was issued for. If you sell more than one product, either check result.getProductId() yourself, or pass expectedProductId (a 5th ActivateRequest constructor arg, or a 3rd validate(licenseKey, version, expectedProductId) overload) and let the server reject a mismatch (result.getErrorCode().equals("WrongProduct")). Find a product's ID in the Admin panel under Products.

Offline grace period (built in)

When offlineGraceDays is set on the license, validate()/activate() cache every successful result to the OS temp directory (java.io.tmpdir). If a later call can't reach the server, the cached result is returned automatically — check isOffline to detect it. Disable with new PermitClient(baseUrl, false).

Error handling

validate() and activate() never throw for a network failure — they fall back to the cache and, failing that, return isValid: false with message: "Cannot reach license server." and isOffline: true. Only meter() throws PermitClient.PermitException (checked exception) on a network error, timeout, or HTTP 429 (rate limited) — usage events aren't cached, since replaying a stale count on reconnect could double-count.

```java
var result = client.validate(key);
if (result.isValid) {
    // proceed — result.isOffline tells you if this came from cache
} else {
    showLicenseDialog(result.message); // key expired, revoked, or unreachable
}

try {
    client.meter(key, "export");
} catch (PermitClient.PermitException e) {
    log.warn("Could not record usage event: " + e.getMessage());
}
```

C++ SDK

Device and usage operations in SDK 2.1.1: all clients can request key-only device removal using tenant slug, key and activation ID. HTTP 204 confirms removal; protected portals still require OTP/account customer flows. Usage reporting returns boolean confirmation, supports nested/empty metadata objects and normalizes quantities below 1 to 1. Java/C++ use tryMeter/try_meter for that boolean contract and retain legacy detailed meter results. Neither operation retries automatically or uses offline cache. See this SDK's README for signatures and failure handling.

Floating sessions in SDK 2.1.1: checkout and heartbeat return a failed session without credentials on network/HTTP/parse failure (RequestFailed). Business denial preserves its error code or uses FloatingDenied. Floating never uses offline cache. The new boolean TryCheckin equivalent confirms HTTP 204; legacy checkin is best effort. Method names follow this language's conventions. See the source README and SDK availability matrix before choosing a registry version.

Official SDKs

SDK request settings

A slow or unavailable license server must not leave your application waiting indefinitely. All nine SDKs default to a 10-second timeout for each attempt and two additional attempts for safe requests. The usual backoff is 250 milliseconds, then 500 milliseconds. This is not a single 10-second deadline for the complete operation: activation also fetches a nonce.

- License validation, nonce/public-key retrieval and floating heartbeat may retry a network failure or HTTP 5xx response. Set the retry count to zero to disable these retries; the range is 0–5.
- Floating checkout, activation POST, usage reporting, device removal and checkin are sent once. A lost response can follow a successful server change, so retrying these blindly can duplicate work.
- HTTP 3xx/4xx responses and malformed successful responses do not retry. SDK-owned transports do not follow redirects, so configure the final HTTPS API address directly.

For validate/activate, branch on the server's error code when available. Otherwise HttpError means an HTTP rejection, InvalidResponse means malformed successful license/nonce evidence and RequestFailed means the network/5xx attempts were exhausted without a usable signed cache. Rejection or malformed successful evidence never becomes cached authorization. A network outage can use only a separately trusted, unexpired proof.

SDK	Configuration
C++	The constructor takes URL, 10 timeout seconds, cache toggle, trusted keys, cache path, device ID, then 2 retries.

Timeout mechanisms follow each runtime: Python uses socket I/O timeouts; Unity delegates to its transport. Caller cancellation is separate from retry policy. Unity's default adapter disables redirects; WebGL needs Unity 2021.3 or later for that protection and fails closed on older versions.
Portable .NET tests do not certify an Editor, IL2CPP or browser build. Verify your actual target before shipping.

Install the published version for your language using the SDK installation references and follow the

offline migration guide when upgrading from 1.x.

C++17 static library. SDK 2.1.1 requires **libcurl**, **OpenSSL** and **nlohmann/json 3.11.3+** (header-only, build time). Public SDK headers use standard-library types. Check your package registry for the released version. JSON fields are extracted using targeted string search so the dependency tree stays minimal. Built with CMake 3.14+.

Download C++ SDK (ZIP)

Building — Linux / macOS

```bash
# Ubuntu / Debian
sudo apt-get install libcurl4-openssl-dev libssl-dev nlohmann-json3-dev

# macOS (Homebrew)
brew install curl openssl

cd SDKs/cpp
cmake -B build -DCMAKE_BUILD_TYPE=Release
cmake --build build
```

Building — Windows (vcpkg + Visual Studio)

```powershell
vcpkg install curl:x64-windows openssl:x64-windows
cmake -B build `
  -DCMAKE_TOOLCHAIN_FILE=C:/vcpkg/scripts/buildsystems/vcpkg.cmake
cmake --build build --config Release
```

Integrating into your CMake project

```CMakeLists.txt
add_subdirectory(SDKs/cpp)
target_link_libraries(your_target PRIVATE permitcore)
```

Quick start

```C++
#include <permitcore/permit_client.hpp>
#include <iostream>

int main() {
  permitcore::PermitClient client("https://api.permitcore.dev");

  auto result = client.validate("PERMIT-XXXX-XXXX-XXXX-XXXX", "",
    "7f3e9a2b-1c4d-4e8f-9a6b-3d2c1e0f5a8b"); // expected_product_id — optional, reject a key issued for a different product
```

```
    if (result.is_valid) {
        std::cout << "License OK — product: " << result.product_name << " (" << result.product_id << ")\n";
        if (result.is_offline) std::cout << "Running on a cached result — server was unreachable.\n";
    } else if (result.error_code == "WrongProduct") {
        std::cerr << "This key is for a different product: " << result.product_id << "\n";
    } else {
        std::cerr << "License rejected: " << result.message << "\n";
    }
}
```

validate() and activate() never throw for a network failure — they fall back to the local offline-grace cache automatically and, failing that, return is_valid: false with is_offline: true. Only meter() throws PermitException.

Validate with version check

```
C++
auto result = client.validate("PERMIT-XXXX-XXXX-XXXX-XXXX", "2.3.1");
if (!result.is_valid && !result.min_version.empty())
    show_error("Upgrade required: minimum version is " + result.min_version);
```

Activation

```
C++
permitcore::ActivateOptions opts;
opts.device_id          = "sha256-of-mac-plus-board-serial";
opts.device_name        = "My-Linux-Server";
opts.version            = "2.3.1";
opts.expected_product_id = "7f3e9a2b-1c4d-4e8f-9a6b-3d2c1e0f5a8b"; // optional

auto result = client.activate("PERMIT-XXXX-XXXX-XXXX-XXXX", opts);
if (result.is_valid) {
    std::cout << "Activated successfully.\n";
} else if (result.error_code == "WrongProduct") {
    std::cerr << "Wrong product: " << result.product_id << "\n";
} else {
    std::cerr << "Activation failed: " << result.message << "\n";
}
```

Metered billing

```
C++
auto m = client.meter("PERMIT-XXXX-XXXX-XXXX-XXXX", "export", 1);
if (m.recorded) std::cout << "Usage recorded.\n";

// Batch — 5 API calls at once
auto m2 = client.meter("PERMIT-XXXX-XXXX-XXXX-XXXX", "api_call", 5);
```

ValidateResult struct fields

Field	Type	Description
is_valid	bool	true if the license is active and all checks pass
product_name	std::string	Product name
remaining_activations	int	Remaining activation seats (0 = at limit)
expires_at	std::string	ISO 8601 expiry timestamp; empty if perpetual
message	std::string	Status message from the server
vendor_warning	std::string	Non-empty if the vendor's PermitCore plan is lapsing
features	std::vector<std::string>	Enabled feature flags
custom_fields	std::map<std::string, std::string>	Vendor-defined metadata
is_trial	bool	true if this is a trial license
trial_days_remaining	int	Days remaining in the trial
node_locked	bool	true if HWID node-locking is enabled
offline_grace_days	int	Days the software may run without a server check
min_version	std::string	Minimum required software version; empty if unrestricted
max_version	std::string	Maximum allowed software version; empty if unrestricted
is_offline	bool	true when this result was served from the local offline-grace cache (server unreachable)
product_id	std::string	Stable ID of the product this license belongs to. Always present when a license was found, even without passing expected_product_id.

ActivateOptions struct

Field	Description
device_id	Hardware fingerprint — SHA-256 of MAC address, motherboard serial, or similar stable identifier

Field	Description
device_name	Human-readable label shown in the admin activation history
version	Current software version string, e.g. "2.3.1"
expected_product_id	Optional — reject (error_code == "WrongProduct") unless this key belongs to this exact product. Also accepted as validate()'s 3rd argument.

Product scoping: by default, any active key belonging to your tenant validates/activates successfully regardless of which product it was issued for. Find a product's ID in the Admin panel under Products.

Offline grace period (built in)

When offline_grace_days is set on the license, validate()/activate() cache every successful result to the OS temp directory (%TEMP% on Windows, $TMPDIR//tmp on POSIX). If a later call can't reach the server, the cached result is returned automatically — check is_offline to detect it. Disable via the third constructor argument: permitcore::PermitClient client(base_url, 10, false);.

Implementation notes

- **No JSON library.** Fields extracted from responses (including the cache file) using std::string::find — the client also links OpenSSL for signature verification.
- **Thread safety.** Each method call creates and destroys its own curl easy handle. Safe to call from multiple threads on a shared PermitClient instance.
- **Custom timeout.** Default is 10 seconds. Pass a second argument: permitcore::PermitClient client("https://...", 5); for 5 seconds.
- **HTTPS on Windows.** Build against the WinSSL backend (CURL_USE_SCHANNEL) for zero-config HTTPS without deploying a CA bundle.

PHP SDK

Device and usage operations in SDK 2.1.1: all clients can request key-only device removal using tenant slug, key and activation ID. HTTP 204 confirms removal; protected portals still require OTP/account customer flows. Usage reporting returns boolean confirmation, supports nested/empty metadata objects and normalizes quantities below 1 to 1. Java/C++ use tryMeter/try_meter for that boolean contract and retain legacy detailed meter results. Neither operation retries automatically or uses offline cache. See this SDK's README for signatures and failure handling.

Floating sessions in SDK 2.1.1: checkout and heartbeat return a failed session without credentials on network/HTTP/parse failure (RequestFailed). Business denial preserves its error code or uses FloatingDenied. Floating never uses offline cache. The new boolean TryCheckin equivalent confirms HTTP 204; legacy checkin is best effort. Method names follow this language's conventions. See the source README and SDK availability matrix before choosing a registry version.

Official SDKs

SDK request settings

A slow or unavailable license server must not leave your application waiting indefinitely. All nine SDKs default to a 10-second timeout for each attempt and two additional attempts for safe requests. The usual backoff is 250 milliseconds, then 500 milliseconds. This is not a single 10-second deadline for the complete operation: activation also fetches a nonce.

- License validation, nonce/public-key retrieval and floating heartbeat may retry a network failure or HTTP 5xx response. Set the retry count to zero to disable these retries; the range is 0–5.
- Floating checkout, activation POST, usage reporting, device removal and checkin are sent once. A lost response can follow a successful server change, so retrying these blindly can duplicate work.
- HTTP 3xx/4xx responses and malformed successful responses do not retry. SDK-owned transports do not follow redirects, so configure the final HTTPS API address directly.

For validate/activate, branch on the server's error code when available. Otherwise HttpError means an HTTP rejection, InvalidResponse means malformed successful license/nonce evidence and RequestFailed means the network/5xx attempts were exhausted without a usable signed cache. Rejection or malformed successful evidence never becomes cached authorization. A network outage can use only a separately trusted, unexpired proof.

SDK	Configuration
PHP	The constructor takes URL, cache toggle, 10 timeout seconds, trusted keys, cache path, device ID, then 2 retries.

Timeout mechanisms follow each runtime: Python uses socket I/O timeouts; Unity delegates to its transport. Caller cancellation is separate from retry policy. Unity's default adapter disables redirects; WebGL needs Unity 2021.3 or later for that protection and fails closed on older versions.
Portable .NET tests do not certify an Editor, IL2CPP or browser build. Verify your actual target before shipping.

Install the published version for your language using the SDK installation references and follow the offline migration guide when upgrading from 1.x.

Pure PHP 8.0+ client. Enable **ext-curl** and **ext-json** for online calls, and **ext-openssl** for offline verification. No Guzzle, no additional packages. Works with Laravel, Symfony, WordPress plugins, raw PHP scripts.

Download PHP SDK (ZIP)

Installation via Composer

SDK 2.1 Composer packaging maps the client and public result types independently. Require your application's vendor/autoload.php; no manual SDK source-file include or client construction is needed before using a result type.

composer require permitcore/sdk:2.1.1
Or install from the downloaded source ZIP:

unzip permitcore-sdk-php.zip -d permitcore-sdk-php # Then in composer.json add the path repository: composer config repositories.local path ../permitcore-sdk-php composer require permitcore/sdk:@dev

Quick start — validate

<?php require 'vendor/autoload.php'; use PermitCore\PermitCoreClient; $client = new PermitCoreClient('https://api.permitcore.dev'); $result = $client->validate('PERMIT-XXXX-XXXX-XXXX-XXXX', expectedProductId: '7f3e9a2b-1c4d-4e8f-9a6b-3d2c1e0f5a8b' // Optional — reject a key issued for a different product instead of silently accepting it); if ($result->isValid) { echo 'Valid! Product: ' . $result->productName . ' (' . $result->productId . ')' . PHP_EOL; if ($result->hasFeature('export')) { enableExport(); } } elseif ($result->errorCode === 'WrongProduct') { exit('This key is for a different product: ' . $result->productId); } else { exit('License invalid: ' . $result->message); }

Activate (call once per installation)

$result = $client->activate(licenseKey: 'PERMIT-XXXX-XXXX-XXXX-XXXX', deviceId: null, // auto-generated HWID deviceName: 'Production Server', expectedProductId: '7f3e9a2b-1c4d-4e8f-9a6b-3d2c1e0f5a8b' // optional); if ($result->errorCode === 'WrongProduct') { exit('This key is for a different product: ' . $result->productId); } if (!$result->isValid) { exit('Activation failed: ' . $result->message); } // Save $result->isValid to local storage — use validate() on every subsequent launch.

Metered billing

// Record a single event $client->meter('PERMIT-XXXX-XXXX-XXXX-XXXX', 'api_call'); // Record bulk usage with metadata $client->meter(licenseKey: 'PERMIT-XXXX-XXXX-XXXX-XXXX', eventName: 'export', quantity: 5, meta: ['format' => 'pdf', 'pages' => 12]);

Floating licenses

// Check out a seat at session start $session = $client->checkout('PERMIT-XXXX-XXXX-XXXX-XXXX'); if (!$session->success) { exit('No seats available: ' . $session->message); } $token = $session->sessionToken; // Heartbeat every 4–5 minutes (use a cron job or event loop) $client-

>heartbeat($token); // Release when the user logs out or the session ends $client->checkin($token);

Offline grace period

$result = $client->validate($licenseKey); // falls back to disk cache automatically if ($result->isOffline) { // Running on cached result — server unreachable showNotice('Running offline. Connect to refresh your license.'); }

LicenseResult properties

Property	Type	Description
isValid	bool	True if the license is active and valid
productName	?string	Product the license belongs to
remainingActivations	?int	Activation slots remaining
expiresAt	?string	ISO 8601 expiry date, null = perpetual
features	?array	Feature flag list, e.g. ['export','api']
isTrial	bool	True for trial licenses
trialDaysRemaining	?int	Days until trial expires
nodeLocked	bool	True if bound to a specific device
offlineGraceDays	?int	How many days the cache remains valid
minVersion	?string	Minimum app version the license allows
maxVersion	?string	Maximum app version the license allows
vendorWarning	?string	Non-fatal message from the vendor
message	?string	Reason when isValid = false
isOffline	bool	True when result came from local cache
productId	?string	Stable ID of the product this license belongs to. Always present when a license was found, even without passing expectedProductId.

hasFeature(string $feature): bool — case-insensitive check whether a feature flag is present on the license.

Product scoping: by default, any active key belonging to your tenant validates/activates successfully regardless of which product it was issued for. If you sell more than one product, either check $result->productId yourself, or pass the optional $expectedProductId parameter and let the server reject a

mismatch (`$result->errorCode === 'WrongProduct'`). Find a product's ID in the Admin panel under Products.

Go SDK

Device and usage operations in SDK 2.1.1: all clients can request key-only device removal using tenant slug, key and activation ID. HTTP 204 confirms removal; protected portals still require OTP/account customer flows. Usage reporting returns boolean confirmation, supports nested/empty metadata objects and normalizes quantities below 1 to 1. Java/C++ use tryMeter/try_meter for that boolean contract and retain legacy detailed meter results. Neither operation retries automatically or uses offline cache. See this SDK's README for signatures and failure handling.

Floating sessions in SDK 2.1.1: checkout and heartbeat return a failed session without credentials on network/HTTP/parse failure (RequestFailed). Business denial preserves its error code or uses FloatingDenied. Floating never uses offline cache. The new boolean TryCheckin equivalent confirms HTTP 204; legacy checkin is best effort. Method names follow this language's conventions. See the source README and SDK availability matrix before choosing a registry version.

Official SDKs

SDK request settings

A slow or unavailable license server must not leave your application waiting indefinitely. All nine SDKs default to a 10-second timeout for each attempt and two additional attempts for safe requests. The usual backoff is 250 milliseconds, then 500 milliseconds. This is not a single 10-second deadline for the complete operation: activation also fetches a nonce.

- License validation, nonce/public-key retrieval and floating heartbeat may retry a network failure or HTTP 5xx response. Set the retry count to zero to disable these retries; the range is 0–5.
- Floating checkout, activation POST, usage reporting, device removal and checkin are sent once. A lost response can follow a successful server change, so retrying these blindly can duplicate work.
- HTTP 3xx/4xx responses and malformed successful responses do not retry. SDK-owned transports do not follow redirects, so configure the final HTTPS API address directly.

For validate/activate, branch on the server's error code when available. Otherwise HttpError means an HTTP rejection, InvalidResponse means malformed successful license/nonce evidence and RequestFailed means the network/5xx attempts were exhausted without a usable signed cache. Rejection or malformed successful evidence never becomes cached authorization. A network outage can use only a separately trusted, unexpired proof.

SDK	Configuration
Go	Options.Timeout uses time.Duration. MaxRetries is an optional integer pointer: nil defaults to two; a pointer to zero disables retries.

Timeout mechanisms follow each runtime: Python uses socket I/O timeouts; Unity delegates to its transport. Caller cancellation is separate from retry policy. Unity's default adapter disables redirects; WebGL needs Unity 2021.3 or later for that protection and fails closed on older versions.
Portable .NET tests do not certify an Editor, IL2CPP or browser build. Verify your actual target before shipping.

Install the published version for your language using the SDK installation references and follow the offline migration guide when upgrading from 1.x.

Pure Go 1.21+ client, standard library only — crypto/ecdsa handles offline token verification natively, so unlike SDKs built on OpenSSL/PHP-openssl/Python cryptography this needs no raw-signature-to-DER conversion step. No third-party packages at all.

Download Go SDK (ZIP)
License: the Go SDK source is MIT-licensed by permitcore. Keep its notice when redistributing. See the Go SDK license. Existing 2.1.1 module archives are immutable; an older archive without the file can use the repository notice. This does not change SDK behavior or license the hosted service.

Installation

Published as a real, tagged Go module — go get works directly, no manual setup:

SDK 2.1.1 uses the /v2 module path. Install it with this command.

go get github.com/permitCore-spec/permitcore-sdk-go/v2@v2.1.1
Or, to work from the source ZIP above instead (e.g. for local edits before a release), add a replace directive to your own go.mod pointing at the extracted folder:

require github.com/permitCore-spec/permitcore-sdk-go/v2 v2.1.1 replace github.com/permitCore-spec/permitcore-sdk-go/v2 => ./permitcore-sdk-go

Quick start — validate

import
"github.com/permitCore-spec/permitcore-sdk-go/v2" client := permitcore.New(
"https://api.permitcore.dev"
) result := client.Validate(
"PERMIT-XXXX-XXXX-XXXX-XXXX"
,
""
,
"7f3e9a2b-1c4d-4e8f-9a6b-3d2c1e0f5a8b"
)
// expectedProductID — required trailing arg, pass "" to omit
if
result.ErrorCode ==
"WrongProduct"
{ log.Fatalf(
"This key is for a different product: %s"
, result.ProductID) }
if
!result.IsValid { log.Fatalf(
"License invalid: %s"
, result.Message) }
if

```
result.HasFeature(
"export"
) { enableExport() }
```

Activate (call once per installation)

```
result := client.Activate(
"PERMIT-XXXX-XXXX-XXXX-XXXX"
,
""
,
// deviceId — auto-generated HWID when empty
"Production Server #1"
,
// deviceName
""
,
// version, optional
"7f3e9a2b-1c4d-4e8f-9a6b-3d2c1e0f5a8b"
,
// expectedProductID — required trailing arg, pass "" to omit
)
if
result.ErrorCode ==
"WrongProduct"
{ log.Fatalf(
"Wrong product: %s"
, result.ProductID) }
if
!result.IsValid { log.Fatalf(
"Activation failed: %s"
, result.Message) }
// Save result.IsValid to local storage — use Validate() on every subsequent launch.
```

Metered billing

```
// Record a single event
client.Meter(licenseKey,
"api_call"
,
1
,
nil
)
// Record bulk usage with metadata
client.Meter(licenseKey,
```

"export"
,
5
, map[string]any{
"format"
:
"pdf"
,
"pages"
:
12
})

Floating licenses

// Check out a seat at session start
session := client.Checkout(licenseKey,
""
,
""
)
if
!session.Success { log.Fatalf(
"No seats available: %s"
, session.Message) } token := session.SessionToken
// Heartbeat every 4-5 minutes to keep the seat alive
client.Heartbeat(token)
// Release the seat when done
client.Checkin(token)

Offline grace period

result := client.Validate(licenseKey,
""
,
""
)
// falls back to disk cache automatically (expectedProductID omitted here)
if
result.IsOffline {
// Running on cached result — server unreachable
showNotice(
"Running offline. Connect to refresh your license."
) }

Offline license tokens

No extra package needed — crypto/ecdsa is standard library. Verification is pure local ECDSA P-256/SHA-256, zero network calls:

```
result := permitcore.VerifyOfflineToken(token, publicKeyBase64)
if
result.IsValid { fmt.Println(
"Valid! Product:"
, result.Payload.ProductName) }
// Bind to this device + persist locally (call once, e.g. at install time)
result = client.ActivateOffline(token, deviceID, productID)
// On every later launch — no token needed, reads the local cache
result = client.ValidateOffline(deviceID, productID)
```

LicenseResult reference

Field	Type	Description
IsValid	bool	True if the license is active and valid
ProductName	string	Product the license belongs to
RemainingActivations	*int	Activation slots remaining
ExpiresAt	string	ISO 8601 expiry date, empty = perpetual
Features	[]string	Feature flag list, e.g. ["export","api"]
IsTrial	bool	True for trial licenses
TrialDaysRemaining	*int	Days until trial expires
NodeLocked	bool	True if bound to a specific device
OfflineGraceDays	*int	How many days the cache remains valid
MinVersion / MaxVersion	string	Version enforcement bounds
VendorWarning	string	Non-fatal message from the vendor
Message	string	Reason when IsValid == false
IsOffline	bool	True when result came from local cache
ErrorCode	string	Stable, machine-readable failure reason
ProductID	string	Stable ID of the product this license belongs to. Always present when a license was found, even without passing expectedProductID.

`HasFeature(feature string) bool` — case-insensitive check whether a feature flag is present on the license.

Product scoping: by default, any active key belonging to your tenant validates/activates successfully regardless of which product it was issued for. Both `Validate` and `Activate` take a trailing `expectedProductID string` argument (pass `""` to omit) — set it and the server rejects a mismatch with `ErrorCode == "WrongProduct"`. Find a product's ID in the Admin panel under Products.

> **Try it live:** the downloadable Task Manager Pro demo includes a working Go build gated behind a real license, plus a developer console panel that exercises every SDK call (validate, activate, offline tokens, floating checkout) against your own PermitCore instance.

Unity SDK

Device and usage operations in SDK 2.1.1: all clients can request key-only device removal using tenant slug, key and activation ID. HTTP 204 confirms removal; protected portals still require OTP/account customer flows. Usage reporting returns boolean confirmation, supports nested/empty metadata objects and normalizes quantities below 1 to 1. Java/C++ use tryMeter/try_meter for that boolean contract and retain legacy detailed meter results. Neither operation retries automatically or uses offline cache. See this SDK's README for signatures and failure handling.

Floating sessions in SDK 2.1.1: checkout and heartbeat return a failed session without credentials on network/HTTP/parse failure (RequestFailed). Business denial preserves its error code or uses FloatingDenied. Floating never uses offline cache. The new boolean TryCheckin equivalent confirms HTTP 204; legacy checkin is best effort. Method names follow this language's conventions. See the source README and SDK availability matrix before choosing a registry version.

Official SDKs

SDK request settings

A slow or unavailable license server must not leave your application waiting indefinitely. All nine SDKs default to a 10-second timeout for each attempt and two additional attempts for safe requests. The usual backoff is 250 milliseconds, then 500 milliseconds. This is not a single 10-second deadline for the complete operation: activation also fetches a nonce.

- License validation, nonce/public-key retrieval and floating heartbeat may retry a network failure or HTTP 5xx response. Set the retry count to zero to disable these retries; the range is 0–5.
- Floating checkout, activation POST, usage reporting, device removal and checkin are sent once. A lost response can follow a successful server change, so retrying these blindly can duplicate work.
- HTTP 3xx/4xx responses and malformed successful responses do not retry. SDK-owned transports do not follow redirects, so configure the final HTTPS API address directly.

For validate/activate, branch on the server's error code when available. Otherwise HttpError means an HTTP rejection, InvalidResponse means malformed successful license/nonce evidence and RequestFailed means the network/5xx attempts were exhausted without a usable signed cache. Rejection or malformed successful evidence never becomes cached authorization. A network outage can use only a separately trusted, unexpired proof.

SDK	Configuration
Unity	Named constructor arguments timeoutSeconds: 10, maxRetries: 2; custom transports must honor the supplied timeout and preserve any received status.

Timeout mechanisms follow each runtime: Python uses socket I/O timeouts; Unity delegates to its transport. Caller cancellation is separate from retry policy. Unity's default adapter disables redirects; WebGL needs Unity 2021.3 or later for that protection and fails closed on older versions.
Portable .NET tests do not certify an Editor, IL2CPP or browser build. Verify your actual target before shipping.

Install the published version for your language using the SDK installation references and follow the offline migration guide when upgrading from 1.x.

Unity 2021.2+ client (.NET Standard 2.1 API Compatibility Level), zero external package dependencies — no Newtonsoft.Json, no additional UPM packages. Works on Desktop, Mobile, and Console builds. SDK 2.1 online WebGL calls require Unity 2021.3+ for redirect protection; older WebGL fails closed. The offline-crypto caveat is noted below.

> **New to PermitCore?** Follow the Unity software licensing quickstart to create a product, validate a real key in Unity, and publish that product in your built-in Store.

Download Unity SDK (ZIP)

Installation

Listed on OpenUPM (com.permitcore.sdk) — the simplest option if you have the openupm-cli installed:

openupm add com.permitcore.sdk@2.1.1

Or install directly from the tagged git repo — Unity Package Manager's git-URL install works with no manual setup: **Window → Package Manager → + → Install package from git URL...**

https://github.com/permitCore-spec/permitcore-sdk-unity.git#v2.1.1

Or work from the source ZIP above: extract it and in Unity: **Window → Package Manager → + → Install package from disk...** and select the extracted folder's package.json. Or copy Runtime/'s contents straight into your project's Assets/ folder.

Quick start — validate

```
using
PermitCore;
using
UnityEngine;
public class
LicenseGate : MonoBehaviour {
private
PermitCoreClient _client;
async void
Start() { _client =
new
PermitCoreClient(
"https://api.permitcore.dev"
);
var
result =
await
_client.ValidateAsync(
```

```
"PERMIT-XXXX-XXXX-XXXX-XXXX"
, expectedProductId:
"7f3e9a2b-1c4d-4e8f-9a6b-3d2c1e0f5a8b"
);
// optional — reject a key issued for a different product
if
(result.ErrorCode ==
"WrongProduct"
) { Debug.LogError(
"This key is for a different product: "
+ result.ProductId);
return
; }
if
(!result.IsValid) { Debug.LogError(
"License invalid: "
+ result.Message);
return
; }
if
(result.HasFeature(
"pro"
)) EnableProFeatures(); } }
```

Activate (call once per installation)

```
var
result =
await
_client.ActivateAsync(
"PERMIT-XXXX-XXXX-XXXX-XXXX"
, deviceId:
null
,
// auto-generated when omitted
deviceName: SystemInfo.deviceName, version: Application.version,
// optional — enforces min/maxVersion
expectedProductId:
"7f3e9a2b-1c4d-4e8f-9a6b-3d2c1e0f5a8b"
);
// optional
if
(result.ErrorCode ==
"WrongProduct"
) Debug.LogError(
```

```
"Wrong product: "
+ result.ProductId);
else if
(!result.IsValid) Debug.LogError(
"Activation failed: "
+ result.Message);
```

Metered billing

```
await
_client.MeterAsync(licenseKey,
"level_completed"
);
await
_client.MeterAsync(licenseKey,
"export"
, quantity:
5
, meta:
new
Dictionary<
string
,
object
> { [
"format"
] =
"png"
});
```

Floating licenses

```
var
session =
await
_client.CheckoutAsync(licenseKey);
if
(!session.Success) { Debug.LogError(
"No seats available: "
+ session.Message);
return
; }
// Heartbeat every 4-5 minutes (e.g. from a coroutine or InvokeRepeating)
await
_client.HeartbeatAsync(session.SessionToken);
// On quit / logout
```

```
await
_client.CheckinAsync(session.SessionToken);
```

Offline license tokens

No extra package needed — System.Security.Cryptography.ECDsa is part of Unity's .NET Standard 2.1 surface on Desktop, Mobile, and Console targets.

```
// Pure local verification — no network call, never throws on Desktop/Mobile/Console.
var
result = PermitCoreClient.VerifyOfflineToken(token, publicKeyBase64);
if
(result.IsValid) Debug.Log(
"Valid! Product: "
+ result.Payload.ProductName);
// Bind to this device + persist locally (call once, e.g. at install time)
var
activated = _client.ActivateOffline(token, deviceId, productId);
// On every later launch — no token needed, reads the local cache
var
cached = _client.ValidateOffline(deviceId, productId);
```

> **Not supported on WebGL builds specifically** — the browser sandbox WebGL runs in has no System.Security.Cryptography implementation, so the three offline-token methods above will throw on that one platform. Online ValidateAsync/ActivateAsync use the default UnityWebRequest transport on supported WebGL versions (Unity 2021.3+ for SDK 2.1). Fully-offline crypto remains unavailable; that's a platform constraint of WebGL itself, not something this SDK can work around.

Hardware ID

```
// Default: a random id persisted to a local file — works identically on every platform.
string
hwid = _client.GetHardwareId();
// Unity-idiomatic alternative, recommended on mobile/console:
string
hwid = UnityHardwareId.Get();
// SHA-256 of SystemInfo.deviceUniqueIdentifier
```

LicenseResult reference

Field	Type	Description
IsValid	bool	True if the license is active and valid
ProductName	string	Product the license belongs to
RemainingActivations	int?	Activation slots remaining

Field	Type	Description
ExpiresAt	string	ISO 8601 expiry date, null = perpetual
Features	List<string>	Feature flag list, e.g. ["export","api"]
IsTrial	bool	True for trial licenses
TrialDaysRemaining	int?	Days until trial expires
NodeLocked	bool	True if bound to a specific device
OfflineGraceDays	int?	How many days the cache remains valid
MinVersion / MaxVersion	string	Version enforcement bounds
VendorWarning	string	Non-fatal message from the vendor
Message	string	Reason when IsValid == false
IsOffline	bool	True when result came from local cache
ErrorCode	string	Stable, machine-readable failure reason
ProductId	string	Stable ID of the product this license belongs to. Always present when a license was found, even without passing expectedProductId.

HasFeature(string feature) — case-insensitive feature check.

Product scoping: by default, any active key belonging to your tenant validates/activates successfully regardless of which product it was issued for — this is exactly the gap this SDK's own demo surfaced while testing it in a real Unity Editor. Both ValidateAsync/ActivateAsync take an optional trailing expectedProductId parameter — set it and the server rejects a mismatch with ErrorCode == "WrongProduct". Find a product's ID in the Admin panel under Products.

> **Try it live:** the downloadable Task Manager Pro demo is a complete Unity project (attach one script, press Play — no Canvas/prefab setup) gated behind a real license, with a built-in developer console panel exercising every SDK call against your own PermitCore instance.

Godot SDK

Native GDScript

Godot SDK 2.2.0 provides the same licensing operations as the other eight official clients, using Godot's native HTTP/TLS and ECDSA implementation. The game needs no .NET edition, GDExtension binary, Python installation or external cryptography package. Use Godot 4.7+; the release evidence identifies the actual editor and export versions tested.

Install the add-on

Download the Godot source ZIP. Copy addons/permitcore/ into your project's res://addons/permitcore/ and import the project in Godot. Enabling the editor plugin is optional. The self-contained Godot demo already contains the add-on: open project.godot and run its main scene.

Validate and activate

Supply your vendor's verified API origin and the expected product ID. Configure public-key trust through your application's trusted release channel. Keep keys out of logs, exported source and URL query strings. Call network methods with await from Godot's SceneTree thread; native HTTP polling yields frames while the request is in progress.

```gdscript
extends Node
# Copy addons/permitcore into res://addons/permitcore first.
const PermitClient = preload("res://addons/permitcore/permitcore_client.gd")
var client = PermitClient.new("https://api.permitcore.dev", {
    "trusted_public_keys": ["YOUR_TENANT_SPKI_BASE64_PUBLIC_KEY"],
    "cache_directory": "user://permitcore"
})

func can_use_pro_features(key: String, product_id: String) -> bool:
    var result = await client.validate(key, "1.0.0", product_id)
    return result.is_valid and result.has_feature("pro")

func register_installation(key: String, product_id: String) -> bool:
    var result = await client.activate(key, "", "My Godot app", "1.0.0", product_id)
    return result.is_valid
```

Validation does not consume a device seat. Explicit activation fetches a nonce and registers the installation. Reuse the same stable device identity later. has_feature checks feature names without case sensitivity; your application enforces the result at the protected operation. No client-side check provides tamper-proof DRM.

Complete operation reference

Method	Result and boundary
await validate(key, version, product)	LicenseResult; product-scoped status without activation.
await activate(key, device, name, version, product)	LicenseResult; fresh nonce, single activation POST.

Method	Result and boundary
await deactivate(tenant, key, activation_id)	Only HTTP 204 confirms removal. Portal OTP/account protection still applies.
await meter(key, event, quantity, meta)	Only JSON recorded:true confirms recording; quantity below one becomes one.
await checkout(key, device, name)	FloatingSession; no offline fallback.
await heartbeat(session_token)	Renewed FloatingSession or failure with no returned credentials.
await try_checkin(session_token)	Boolean confirmation of HTTP 204. checkin is the best-effort variant.
get_hardware_id()	Private hashed installation identifier using a persisted seed; not clone attestation.
verify_offline_token(token, public_key)	Local signature/deadline primitive, with explicit caller-managed trust.
verify_grace_cache_token(token, public_key)	Local grace-token signature/deadline primitive.
activate_offline(token, device, product)	Configured trust and context checks; stores only original signed proof.
validate_offline(device, product)	Reverifies saved original proof on every read.
await verify_offline_online(token)	Server token verification and revocation status; requires connectivity.
result.has_feature(name)	Case-insensitive feature membership.

Results, offline trust and recovery

LicenseResult exposes validity, product identity, features, custom fields, device capacity, expiry/maintenance, trial/version bounds, vendor warning and payment/grace metadata. Use error_code for branching and message for display. TokenResult exposes is_valid and a signed payload dictionary; signed dictionary fields retain their camelCase wire names.

Automatic grace fallback requires independently configured trusted_public_keys and matching license hash, product, device and signed deadline. Activation fallback also needs signed existing-activation evidence. No pins means no trusted cache. Offline proof cannot discover later revocation, device removal or payment changes; its signed deadline remains binding. Removing a device attempts to discard that key's local grace cache.

The default cache is user://permitcore. Set an application-specific cache_directory and, where appropriate, a stable device_id. Invalid constructor options populate configuration_error and make network operations fail without a request. An optional tls_ca_file loads your own independently trusted PEM CA for self-hosting; hostname verification stays enabled.

SDK request settings

A slow or unavailable license server must not leave your application waiting indefinitely. All nine SDKs default to a 10-second timeout for each attempt and two additional attempts for safe requests. The usual backoff is 250 milliseconds, then 500 milliseconds. This is not a single 10-second deadline for the complete operation: activation also fetches a nonce.

- License validation, nonce/public-key retrieval and floating heartbeat may retry a network failure or HTTP 5xx response. Set the retry count to zero to disable these retries; the range is 0–5.
- Floating checkout, activation POST, usage reporting, device removal and checkin are sent once. A lost response can follow a successful server change, so retrying these blindly can duplicate work.
- HTTP 3xx/4xx responses and malformed successful responses do not retry. SDK-owned transports do not follow redirects, so configure the final HTTPS API address directly.

For validate/activate, branch on the server's error code when available. Otherwise `HttpError` means an HTTP rejection, `InvalidResponse` means malformed successful license/nonce evidence and `RequestFailed` means the network/5xx attempts were exhausted without a usable signed cache. Rejection or malformed successful evidence never becomes cached authorization. A network outage can use only a separately trusted, unexpired proof.

SDK	Configuration
Godot	Constructor options `"timeout_seconds": 10.0, "max_retries": 2`. Native frame-polled HTTP keeps the game loop responsive, including while game time is paused.

Timeout mechanisms follow each runtime: Python uses socket I/O timeouts; Unity delegates to its transport. Caller cancellation is separate from retry policy. Unity's default adapter disables redirects; WebGL needs Unity 2021.3 or later for that protection and fails closed on older versions.
Portable .NET tests do not certify an Editor, IL2CPP or browser build. Verify your actual target before shipping.

Install the published version for your language using the SDK installation references and follow the offline migration guide when upgrading from 1.x.

Test with the actual engine

The source distribution contains shared vectors and native-engine tests. Run `python -m pip install -r tests/requirements.txt`, then `python tests/run_tests.py --godot /absolute/path/to/godot`. Python is test tooling, not a game dependency. Tests exercise real Godot HTTP/TLS, signatures, signed-context attacks, response failures, retries and game-loop responsiveness. Actual editor/export and hosted acceptance remain distinct from loopback fixture evidence.

Verify your own exported target before shipping. Web, mobile and console limitations are platform-specific; do not infer support from a desktop run. Never distribute test signing keys or a development-only success response as production authorization.

Webhooks

Receive events on your server

Webhooks notify a trusted server when subscribed events occur, such as a license activation or revocation. Configure them under **Webhooks → Add Endpoint** (Starter and above).

The Webhooks page before setup. Add an endpoint to start receiving events; delivery attempts then appear below.

1. Enter a publicly reachable HTTPS endpoint; private and loopback destinations are blocked.
2. Select the events your integration handles.
3. Save the signing secret securely when it is shown.
4. Verify a simulator event before processing production traffic.

Verify the raw body

The X-PermitCore-Signature header contains sha256=<hex digest>. Compute HMAC-SHA256 over the exact received body bytes with your endpoint secret. Do not parse and reserialize JSON before verification. Reject missing or malformed signatures without throwing.

Node.js

```
const { createHmac, timingSafeEqual } = require('node:crypto');

function verifyWebhook(rawBody, signatureHeader, secret) {
  if (typeof signatureHeader !== 'string') return false;
  const match = /^sha256=([a-f0-9]{64})$/i.exec(signatureHeader);
  if (!match || !Buffer.isBuffer(rawBody)) return false;
  const expected = createHmac('sha256', secret).update(rawBody).digest();
  const received = Buffer.from(match[1], 'hex');
  return received.length === expected.length &&
    timingSafeEqual(received, expected);
}
```

Configure your HTTP framework to preserve the raw request body. Only parse the payload after verification. The signature authenticates the body, but does not prevent replay: store processed event IDs durably, enforce a unique constraint, and apply your business update in the same transaction as that record. A duplicate event should receive a successful response without repeating the work.

Event envelope

JSON

```
{
  "id": "evt_1234567890abcdef1234567890abcdef",
  "event": "license.activated",
  "timestamp": "2026-09-06T10:00:00Z",
  "data": {}
}
```

data depends on the event. Use the simulator to inspect the payload for each subscribed event. Do not assume the payload contains a plaintext license key. Treat unknown fields as additive and unknown event names as unsupported rather than crashing the endpoint.

Events

Event	Purpose
license.created / license.updated / license.revoked	Keep downstream license records in sync.
license.activated	React to an installation being registered.
license.expiring	Handle upcoming expiry notifications.
license.abuse_suspected	Investigate unusual request volume or IP fan-out.
tenant.device_limit_reached	A new activation was refused at the account cap. Includes tenantId, licenseId, productId, errorCode, used and limit; at most one notification per tenant per five minutes. Counts are a post-rejection snapshot.
product.created / product.updated / product.deleted	Synchronize the product catalog.
webhook.test	Test delivery and signature verification.

Delivery and retries

A 2xx response marks delivery successful. Non-2xx responses and network failures are retried. The current receiver-failure path makes up to five total attempts: the initial attempt, then delays of 5 minutes, 30 minutes, 2 hours, and 24 hours after the previous failure. Check **Next Retry At** for a delivery's scheduled retry rather than assuming exact wall-clock arrival.

Status	Meaning
pending	Queued, not yet attempted.
retrying	An attempt is being processed or a retry is scheduled.
delivered	The receiver returned 2xx.
exhausted	No further automatic attempts remain.
blocked	The destination failed the public-address security check.

Accept events quickly into a durable queue and return 2xx only after they are safely recorded. Do slow downstream work separately. Monitor failed deliveries; do not assume delivery is exactly once or that events always arrive in creation order.

Transport exceptions in the simulator and new delivery error records show Webhook request failed. rather than private transport details. HTTP status and retry state remain available. The simulator still previews the receiver response body; historical error records are unchanged.

Test before launch

1. Use the endpoint's simulator to preview and send an event.
2. Confirm a correct signature is accepted and a modified body is rejected.
3. Send the same event ID through your receiver twice and verify the business action happens once.
4. Review HTTP status and delivery history in PermitCore.

Analytics

Analytics charts, drilldowns, expiring-license reports and CSV exports exclude test licenses and their activations. Test records remain in Licenses and Store Orders.

Admin Panel
The Analytics page gives you a detailed view of your license activity and activation trends over time.

Analytics dashboard showing activation trends, license distribution, and product breakdowns.

What analytics shows

- **Activation trends** — daily/weekly/monthly activation volume
- **License distribution** — breakdown by product, status, and expiry
- **Top products** — which products have the most activations
- **Activation activity over time** — registration events over time, including reactivations; standing device occupancy is shown separately in Dashboard and Billing

Time range

Switch between **Today**, **Last 7 days**, and **Last 30 days** to see different views of your data.

Activity Log

Admin Panel

The Activity Log records every significant action in your account. It is append-only — entries cannot be edited or deleted. This provides a full audit trail for compliance and security purposes.

The Activity Log showing all account actions with timestamps, IP addresses, and user attribution.

What gets logged

- License created, updated, revoked
- User logins, logouts, failed login attempts
- MFA events (enabled, disabled, backup codes used)
- User created, updated, deactivated
- Product created, updated, deleted
- Company settings updated
- Password changes
- Account locked (after too many failed login attempts)

Reading a log entry

Each entry shows:

- **Action** — what happened (e.g. CreateLicense, Login, RevokeLicense)
- **Entity** — which record was affected (product name, user email, etc.)
- **User** — which team member performed the action
- **IP Address** — where the request came from
- **Timestamp** — when it happened (UTC)
- **Details** — a JSON diff showing before/after values for updates

Use cases

- **Compliance** — demonstrate to auditors that key operations are tracked
- **Security investigation** — if a key was unexpectedly revoked, find out who did it and from where
- **Team accountability** — see which team member made which change and when

Dashboard

Admin Panel
The Dashboard is your home screen after logging in. It gives you a real-time overview of your license management activity.

Find PermitCore project information and directory links, including ToolFame, in the public website footer.

Your Admin session

Your Admin session ends after **30 minutes of inactivity**. A warning shows the remaining time during the last five minutes. Click **Continue session** to stay signed in. Clicks, typing, scrolling, navigation and manually reloading the page before expiry also renew the session. Keyboard shortcuts such as Escape closing help also renew activity. Input bursts are grouped for up to 15 seconds. Background requests and automatic JWT refresh do not extend the countdown. The server enforces the session deadline. Save your work before leaving the page.

If the timer reaches zero without renewal, Admin returns to Sign In and explains: **You have been signed out because your session expired.** Sign in again to continue. Reloading after expiry does not restore the expired session. If the connection fails while continuing, retry before the timer ends; a temporary network failure does not immediately sign you out.

If simultaneous sign-ins show **Your session changed during sign-in. Please try again.**, retry your sign-in. PermitCore preserves the successful session and does not issue credentials for the losing attempt.

The Dashboard showing stats, Getting Started checklist, and activation charts.

License counts, activation charts and the weekly digest exclude test licenses. Test records remain in Licenses and Store Orders. Account capacity includes occupied seats on test, revoked, expired and imported keys.

Overview cards

The four cards at the top give you instant metrics:

Card	What it shows
Total Licenses	Live license keys (including revoked ones) across all products
Active Licenses	Keys that are currently valid, not revoked, and not expired
Products	Number of products in your account
Team Members	Users who have access to your admin panel

Topbar status bar

At the very top of every page you can see:

- **Encrypted keys badge** — a reminder of the encryption workflow; validation, Store delivery and optional escrow have separate access boundaries
- **Active device capacity counter** — shows occupied device seats / total capacity. Changes colour: green (safe), yellow (over 80% used), red (limit reached)
- **Plan badge** — your current plan (Free / Starter / Professional / Enterprise) with days left on subscription

Getting Started checklist

The "Getting Started" widget links to creating a product, generating a license key and testing a real license. The test step reopens the welcome guide, with language examples and links to SDK documentation. It completes after a real successful validation, not simply copying a sample or creating a key. Publishing a Store is an additional step on plans with Store access; it is not required on Free. The widget shows progress, disappears when the required steps are complete and can be minimized or dismissed. On mobile it moves into the page flow instead of covering the content.

License Activations chart

Shows how many times licenses were activated over time. Switch between **Today**, **Last 7 days**, and **Last 30 days**. A flat line is normal for new accounts — activations only appear after customers start using their keys.

License Status chart

A donut chart showing the proportion of **Active**, **Expired**, and **Revoked** licenses at a glance.

Recent Activity

A log of the last actions taken in your account — license creation, user logins, revocations, etc. Click **"View all"** to go to the full Activity Log.

Products

Admin Panel

A **product** is a container for license keys. Think of it as the software title you're licensing — for example "Acme Video Editor", "Acme Plugin Pack", or "Acme Enterprise Suite". You can have as many products as you need.

The Products page showing all products with license counts and actions. Each row also shows the product's GUID with a copy button — see the callout below for why you'll need it.

> Every license key belongs to exactly one product. When a customer validates or activates their key, the response includes the product name and productId — but by default, any active key belonging to your tenant validates successfully regardless of which of your products it was issued for. If you sell more than one product, either check the response's productId yourself, or pass the optional expectedProductId parameter (copy it from here or from a license's own detail page) and let the server reject a mismatch for you. See the API reference for details.

Creating a product

Go to **Products → New Product** and fill in:

Field	Description	Example
Name	The product display name — returned in validation API responses	Acme Video Editor Pro
Description	Internal notes — customers don't see this	Windows + macOS bundle, v3
Version	Optional version identifier for your own tracking	3.2.1

Editing a product

Click the **Edit** (pencil) icon on any product. You can change the name, description, version, and toggle the product active/inactive.

> **Deactivating a product** instantly blocks all license validation for every key under that product — your customers' licenses will return isValid: false until you reactivate. Use this only if you intend to stop the product entirely.

Deleting a product

A product can only be deleted if it has **no active licenses**. Revoke all active licenses first, then delete. This prevents accidentally deleting a product that paying customers are still using.

Licenses

Admin Panel

License keys are the core of PermitCore. Each key is a unique string in the format PERMIT-XXXX-XXXX-XXXX-XXXX that your customer enters in your software to activate it.

The Licenses page showing all keys grouped by product, with activation counters, status, and actions.

Understanding the licenses list

Column	What it means
Serial	Shows "Encrypted" until you decrypt with your passphrase. Click **View** to reveal the plaintext key.
Activations	currentActivations / maxActivations — e.g. 2 / 3 means 2 of 3 allowed activations used
Expires	"Never" for perpetual licenses, or the exact expiry date/time
Status	Active (green), Revoked (red), or Expired (grey)
Created	When the key was generated

The license key format

Every key starts with PERMIT- followed by four groups of 4 characters. The charset avoids look-alike characters (no 0/O, no 1/I/l) to prevent customer confusion.

PERMIT-A3K7-MN2P-QR9X-TZ5W

Filtering and searching

Use the **Active / Revoked / All** filter dropdown and the **Search by product** box to narrow down large lists. Licenses are grouped by product for easier navigation.

Creating & Managing Licenses

Admin Panel
The Create License form — fill in product, count, max activations, optional expiry, and your passphrase.

Creating licenses

Go to **Licenses → Create License** (or click **+ New License** in the topbar). Fill in the form:

Field	Description
Product	Which product this license is for
Number of Licenses	How many keys to generate in one batch (1–100). Useful for bulk sales or resellers.
Max Activations per Key	How many devices/installs one key can be used on. E.g. 1 = single machine, 5 = family licence.
Expires At	Optional expiry date. Leave blank for perpetual (never expires) licenses.
Passphrase	Your encryption passphrase (minimum 16 characters). Must be the same passphrase you always use for this product — all keys under a product should use the same passphrase.

Viewing and copying keys

License keys are stored encrypted. To reveal a key, find it in the Licenses list and click **View**. You'll be prompted for your passphrase. After entering it, the plaintext key is decrypted *in your browser* and displayed so you can copy it and send it to your customer.

> **Decrypt All** — On the Licenses page, each product group has a "Decrypt All" button that reveals all keys in that group at once with a single passphrase entry. Useful when you need to export a batch of keys to a CSV.

Editing a license

You can update a license's **Max Activations** and **Expiry Date** after creation. This is useful if a customer buys additional seats or needs a time extension. You cannot lower Max Activations below the current activation count (that would strand existing users).

Revoking a license

To revoke a license (e.g. customer requested a refund, or the key was shared/leaked), click **Revoke**. Revoked licenses return `isValid: false` immediately via the API. **Revocation is permanent** — you cannot un-revoke a key. Issue a new key if needed.

Exporting licenses to CSV

Click **Export CSV** on the Licenses page to download all license data. The exported file contains *encrypted* keys. To export plaintext keys, use "Decrypt All" first, then export — the CSV will include the decrypted serial numbers.

Migrating from Other Systems

Admin Features

Read the licensing migration guide. Separate key preservation from client rollout, device records, offline proofs and payment migration.

Already running licensing on Keygen, Cryptolens, LicenseSpring, or something you built yourself? With the original key values, you can preserve customer keys during a planned application migration. **Licenses → Import** reads a CSV export from your old system and creates matching licenses here — using the exact same key strings your customers already have.

Why the key string doesn't need to change

PermitCore never validates a license by comparing its format — it validates by hash. When a customer's software calls `POST /api/v1/validate` or `/activate`, the server computes a SHA-256 hash of whatever string was sent and looks it up. That means any string works as a license key, not just PermitCore's own `PERMIT-XXXX-XXXX-XXXX-XXXX` format — importing simply registers your existing key strings under that same lookup, unchanged.

> **What import does not do:** it does not redirect copies of your software that are already installed on customer machines away from your old vendor's API. Those installs have your old vendor's validate/activate URL built into the release they're running — only a new release, built with a PermitCore SDK, actually stops calling the old vendor. Plan this as a phased rollout: import your license data now, ship the release that switches over, and let already-installed copies fall off the old vendor naturally as customers update — not as an instant cutover.

How it works

1. **Export** your licenses from your current system as a CSV (most vendors offer this from their dashboard or API).
2. Go to **Licenses → Import**, pick your source system from the dropdown (or "Generic CSV" for anything else), and upload the file or paste its contents.
3. **Map columns** — PermitCore guesses the mapping based on the source system you picked; review and correct anything before continuing.
4. **Preview** every row before anything is sent — rows with a missing key, an unrecognized product name, or an invalid value are flagged and skipped, without blocking the rest of the file.
5. Optionally set a **passphrase** so you can decrypt/reveal these keys again later from the admin panel.
6. **Review the results** — each row shows whether it was created, already existed (skipped, not an error), or failed, with a downloadable CSV of anything that didn't import cleanly.

The passphrase is optional for imports

Manually-created licenses always need a passphrase because PermitCore generates the key itself and needs a way to show it to you once. Imported licenses are different — you already have the plaintext

key from your old system's export, so encryption here is purely for being able to view the key again later from the admin panel.

If you skip the passphrase, the license still works completely normally for validation and activation (only the key's hash matters for that) — you just won't be able to reveal the plaintext again afterward. Store-issued licenses use a separate server-managed delivery and recovery path; this import choice controls optional recovery storage for imported rows.

Carrying over activations already in use

InitialActivations seeds the used count between zero and MaxActivations. For example, importing 3 of 5 used seats preserves two remaining count slots. It does not recreate old device identities or floating sessions. A previously installed node-locked client may consume another slot when it registers against the new system because its old activation row is absent. Decide and test the count/re-enrollment policy before the import.

Field mapping by source system

Field names below are the ones each vendor's own API/export documentation uses at the time of writing — a vendor changing their export format won't automatically update the wizard's auto-guess, but you can always remap columns manually.

Keygen.sh

Keygen field	PermitCore field	Notes
key	License Key	Used as-is, no reformatting.
policy / product	Product	Matched against your PermitCore product names — a Keygen "policy" is roughly a product's license rules.
maxMachines	Max Activations	Keygen's per-license machine limit.
uses	Activations already used	Keygen's own usage counter.
expiry	Expires At	ISO 8601 timestamp.
(owner relationship — no direct field on the license)	Customer Email	Keygen licenses reference a separate user resource; most CSV exports add their own flattened email column.

Cryptolens

Cryptolens field	PermitCore field	Notes
key	License Key	Used as-is.
productId / product name	Product	Matched against your PermitCore product names.

Cryptolens field	PermitCore field	Notes
maxNoOfMachines	Max Activations	Cryptolens's activation limit.
activatedMachines	Activations already used	A list in Cryptolens's API — use its count/length as a plain number in your CSV.
expires	Expires At	
customer	Customer Email	

LicenseSpring

LicenseSpring field	PermitCore field	Notes
license_key	License Key	Used as-is.
product	Product	Matched against your PermitCore product names.
max_activations (or max_consumptions for metered licenses)	Max Activations	LicenseSpring splits activation-based and consumption-based licenses — map whichever your license type uses.
times_activated	Activations already used	
validity_period	Expires At	
customer.email	Customer Email	Nested under a customer object in LicenseSpring's API — most CSV exports flatten it to its own column.

Generic CSV / home-grown systems

Any CSV with a column of license key strings works — the wizard doesn't care about your column names, only how you map them. At minimum you need a key column; a product, max activations, activations-used, expiry, and customer email column are all optional and can be left unmapped (a default product and max-activations value apply instead).

What's not carried over yet

Tags, IP/country allowlists, and custom fields aren't part of the import wizard today — they're genuinely not sent, not silently dropped. If your old system's licenses used any of these, add them afterward from each license's **Edit** screen (or via PUT /api/licenses/{"{id}"} if you're scripting the import).

Permissions

Importing requires the same **Create / Edit / Revoke** license permission as manually creating a license — if you use custom roles, grant that permission to any role that should be able to import.

API

The wizard is a thin client over POST /api/licenses/import — if you'd rather script a migration directly, hash each existing key with SHA-256 client-side (never send a plaintext key to the server) and POST an array of rows, up to 500 per request:

POST /api/licenses/import Authorization: Bearer <token> Content-Type: application/json
[{ "productId": "...", "encryptedKey": "", "iv": "", "plaintextKeyHash": "<sha256 hex of the existing key>", "maxActivations": 5, "initialActivations": 3, "expiresAt": "2027-01-01T00:00:00Z", "customerEmail": "customer@example.com" }]

Leaving encryptedKey/iv empty (as above) skips the reveal-later capability, matching the wizard's "Skip passphrase" option. Each row reports back one of Created, Duplicate (already imported — not an error), ValidationFailed, or Failed — a bad or duplicate row never blocks the rest of the batch.

Active Device Capacity — Deep Dive

Seats and total active device capacity

A per-license seat limit controls one key. Your account cap covers occupied seats across every key: Free 50, Starter 1,000, Professional 10,000, Enterprise unlimited. Active top-ups add capacity. Creating keys does not use seats.

Action	Occupied seats
Validate a key	No change; normal license and API rate limits apply
Activate a new device	Uses one seat on the key and account
Same-device node-locked reactivation	Reuses its primary seat, even above the account cap
Repeated count-based activation	Uses another seat each time
Test, revoked, expired and imported keys	Their occupied seats count
Renewal or plan change	No reset; all occupied seats remain counted
Confirmed device removal or key reset/transfer	Releases the removed occupied seats

Application flow

Activate a new installation with a stable device ID. On later launches validate its license; do not repeatedly register count-based licenses. A local activation flag is not proof of validity. A changed device ID may use a new seat. The same device on two keys occupies a seat on each key. See the team license guide for customer seat packs and device reassignment.

When a limit is reached

New activation returns HTTP 200 with `isValid: false` and `ActiveDeviceLimitReached` when the account cap is full. `SeatsExhausted` still means the particular key is full. `PoolExhausted` is deprecated and no longer returned. All nine SDKs preserve the new code.

After a downgrade or top-up expiry, existing devices retain normal validation and entitlement rules. Only new activations are blocked until capacity is available. There is no monthly reset. Revoking a key, letting it expire or marking it as test does not free capacity. Remove a device or reset/transfer the key to release its seats. Already issued offline proofs remain valid only until their signed deadline.

The client message is: This software's license server has reached its active device limit. Ask the vendor to free a device or upgrade. Every account-cap rejection records an ActivationFailed audit with errorCode and reason. The owning tenant can see this fixed reason in Activity where its plan and permissions allow access; other detailed audit changes remain restricted. Vendors with a subscribed webhook receive `tenant.device_limit_reached`, throttled to once per tenant per five minutes. Notification failures do not change activation refusal.

Read your usage

Billing and the Dashboard show the same account usage. The license list shows per-key `currentActivations / maxActivations`. A 1,000-seat cap can support roughly 1,000 one-device customers or 500 two-device customers in total, rather than that many new customers every month.

Capacity and subscription validity are separate. Existing vendor subscription lapse, warning, grace and license expiry rules still apply. Review pricing and Billing for your plan and active top-ups.

Users & Roles

Admin Panel
PermitCore supports multiple team members under one company account. Each user has a role that determines what they can do.

The New Role form starts with no permissions selected. Company Admin remains the fixed full-access role.

Roles explained

Role	What they can do
Company Admin	Full access: manage products, licenses, users, billing, webhooks, company settings. The account owner is always a Company Admin.
Company User	Access is defined by the custom role assigned to this user. A role can grant selected menus and tabs; it is not automatically read-only. Missing role permissions mean no access.

Create a custom role first

1. Open **Roles → New Role** and give it a descriptive name, such as Support.
2. Select the menus and tabs that role needs. A new role starts with no access.
3. Save the role and assign it when creating or editing a Company User.
4. Test with that user account. A menu can be unavailable because of role permissions or your billing plan.

Company Admin is the fixed full-access role and cannot be edited through the role editor. Custom-role restrictions are checked by the API as well as the interface.

Adding a team member

Go to **Users → Add User** and fill in:

- **First & Last Name** — for display in the admin panel
- **Email** — they will use this to log in
- **Password** — set a temporary password (they can change it in Settings → Security)
- **Role** — Company Admin for full access, or Company User with an assigned custom role

> Invited users do not yet receive an automated welcome email — share their login credentials manually. They can change their password after first login via **Settings → Security**.

Deactivating a user

You can deactivate a user (e.g. when an employee leaves) without deleting them. Deactivated users cannot log in but their full audit history is preserved. You cannot deactivate yourself — another admin must do it.

> **You cannot delete the last Company Admin.** Promote another user to Admin first, then delete/deactivate the original account.

Multi-Factor Authentication (MFA)

Each user can enable TOTP-based MFA independently. Supported apps: Google Authenticator, Authy, 1Password, Bitwarden, Microsoft Authenticator. Setup is under **Settings → Security → Enable MFA**.

Billing & Plans

Admin Panel
When a plan change takes effect, features are checked against the current plan on the server, including existing API keys and queued work. Without Store access, new browsing and purchases stop, while existing buyers retain direct access to their account, licenses and order history. Saved Portal verification and region restrictions are not silently removed. Occupied devices are never reset by a plan change. License validation after a platform subscription ends still follows its separate warning/ deadline rules.

PermitCore offers four plans. The plans differ in the **total active device capacity**, API rate limit, and which features are available. Lower tiers will see a lock icon and an upgrade prompt for features they don't have access to.

When changing plans, review the selected price and billing interval, then check the required Terms of Service acknowledgement. **Continue** is muted and unavailable until it is checked. Analytics cookie choices remain separate.

After registration checkout, **Payment Successful** means a confirmed paid checkout. An eligible trial shows **Subscription Activated**. If confirmation is still pending, sign in to check your subscription before attempting another purchase.

Current prices, activation limits and eligible paid trials are shown on Pricing and Billing. Public HTML contains billing data captured at build time; the browser refreshes it from the pricing service. If that service is unavailable, the page shows an availability message instead of substituting prices or trials. Checkout confirms the final price and eligibility. The summaries below describe standard plan features.

The Billing page showing current plan, activation usage, subscription history, and invoices.

Free
€0
/mo
50 active device seats
Node-locked / HWID licenses
REST API (validate + activate)
AES-256-GCM encryption
Signing key export — take your keys anytime
Community support

Starter
Current price
1,000 active device seats
License Store storefront
Webhooks + Analytics
Geo-blocking, templates, CRM

Floating licenses (up to 5 seats)
Email support (48h)

Professional
Current price
10,000 active device seats
Metered billing + Stripe
Floating (unlimited) + Bundles
SSO / OAuth + Offline tokens
Priority support (8h)

Enterprise
Current price
Unlimited activations
Reseller portal + commissions
Custom CSS (store + portal)
Unlimited audit logs
Custom contracts / DPA
Dedicated support (1h SLA)

Prices shown are in EUR, plus applicable tax.

What each plan includes

Feature	Free	Starter	Professional	Enterprise
Activations / month	50	1,000	10,000	Unlimited
License keys	Unlimited	Unlimited	Unlimited	Unlimited
API rate limit	60/min	60/min	300/min	1,000/min
Node-locked / HWID	✓	✓	✓	✓
GDPR export	✓	✓	✓	✓
License Store	—	✓	✓	✓
Webhooks	—	✓	✓	✓
Analytics & Activity Log	—	✓	✓	✓
API Keys	—	✓	✓	✓

Feature	Free	Starter	Professional	Enterprise
Geo-blocking	—	✓	✓	✓
Templates	—	✓	✓	✓
Customer CRM	—	✓	✓	✓
Floating licenses	—	Up to 5 seats per license	No plan seat limit	No plan seat limit
Shop Designer and static banner	—	✓	✓	✓
Hero slider	—	—	✓	✓
License expiry emails	✓	✓	✓	✓
Metered billing	—	—	✓	✓
Bundles	—	—	✓	✓
SSO / OAuth	—	—	✓	✓
Offline activation	—	—	✓	✓
Resellers	—	—	—	✓
Signing key export	✓	✓	✓	✓
Audit logs	7 days	30 days	90 days	Unlimited
Custom CSS	—	—	—	✓
Custom contracts / DPA	—	—	—	✓
Support	Community	Email (48h)	Priority (8h)	Dedicated (1h)
Uptime SLA	—	—	99.9%	99.99%

Uptime and response-time figures above are targets, not contractually guaranteed service credits, except where formalized in a signed Enterprise agreement.

Subscription history & invoices

The Billing page shows your full subscription history (upgrades, downgrades, cancellations) and a downloadable invoice history. Each invoice has a **PDF** download link for accounting purposes.

Upgrading your plan

Go to **Billing**, find the plan you want, and click it to start the Stripe checkout. Your plan upgrades instantly after successful payment. The new device capacity takes effect immediately; occupied seats are preserved.

> Platform Stripe billing uses Stripe-hosted card processing. Vendor Store payments use the vendor's selected Stripe, Paddle or Lemon Squeezy connection. PermitCore does not store card details.

Platform billing and vendor Store payments

Admin → Payment Providers manages platform Stripe credentials and Paddle sandbox preparation. Stripe remains the default for platform plans. Saving a Paddle connection does not enable live payments or migrate existing Stripe contracts. Each vendor chooses **Stripe, Paddle or Lemon Squeezy** separately in **License Store → Settings**.

SuperAdmin: prepare platform Paddle in sandbox

SuperAdmin: PermitCore plan billing. This connects the platform's Paddle sandbox account for testing tenants' PermitCore plans. A vendor selling their own products uses **License Store → Settings** and the separate Store guide.

Stripe remains the default for platform plans. Saving a Paddle account or marking it Default does not switch the platform provider, migrate Stripe subscriptions or enable live Paddle payments. Live platform Paddle is held until seller verification, business readiness and release approval.

1. Save an inactive sandbox connection

1. Open **Admin → Payment Providers → New Provider**. Choose **Paddle**, give it a clear internal name and keep **Sandbox / Test Mode** selected.
2. In the Paddle Billing sandbox dashboard, find your numeric **Seller ID** in account settings.
3. Under **Developer tools → Authentication**, create a sandbox API key with the account, catalog, customer, transaction, subscription, adjustment and customer-portal permissions needed for this integration. Copy its `pdl_sdbx_` value into **API key**.
4. Create a **client-side token** in the same sandbox workspace. Its value starts with `test_`. Enter it in **Client-side token**; this public token opens Paddle.js and is different from the secret API key.
5. Leave **Active** off and save. Open the saved connection to obtain its exact sandbox webhook URL. Do not reuse a Store webhook destination.

2. Connect signed payment notifications

In Paddle sandbox **Developer tools → Notifications**, add the displayed URL as a destination. Subscribe to `transaction.completed`, `transaction.payment_failed`, `transaction.past_due`, `subscription.created`, `subscription.trialing`, `subscription.activated`, `subscription.updated`, `subscription.past_due`, `subscription.canceled`, `subscription.paused`, `subscription.resumed`, `adjustment.created` and `adjustment.updated`.

Copy that destination's signing secret into **Webhook signing secret**. Enable Active and save. API keys and signing secrets are stored encrypted; a blank secret input on a later edit keeps the saved value. The public client token and numeric Seller ID are not secrets.

3. Verify the account and prepare prices

Click **Verify sandbox account and sync plan catalog**. PermitCore checks the Seller ID against Paddle's account API and creates or reuses platform-only SaaS products and EUR prices. Existing price mappings are retained for historical subscriptions. A successful sync does not prove a completed payment.

The checkout page is `/billing/paddle-checkout` on the configured Public host. Its domain must be approved in Paddle sandbox Website approval. Checkout creation supplies this URL per transaction; it does not change the seller account's global default payment link.

4. Test before changing the platform default

An operator selects the connection ID with `PlatformBilling:PaddleProviderId` and uses `PlatformBilling:Provider=paddle` only in the isolated test/staging configuration. Existing Stripe contracts keep Stripe. Keep `PlatformBilling:PaddleLiveEnabled=false`. Do not change live seller settings.

Use an owned test account and Paddle sandbox cards. Verify the plan, paid amount/currency/tax and provider document; test eligible three-calendar-month trials, cancellation/resumption, upgrades, refunds and real failed/recovered renewal. A browser success URL or simulated webhook does not prove settlement. Sandbox is clearly labeled and moves no real money.

Manage Subscription opens a fresh, temporary Paddle customer-portal link. Open it from Billing whenever needed instead of bookmarking the signed link.

If setup does not work

- **Cannot activate:** check all four sandbox values: Seller ID, API key, client-side token and webhook signing secret. Never paste the secret API key into the public token field.
- **Sync failed:** check that the API key belongs to the same Seller ID and has the required permissions. Keep Stripe as the default while investigating.
- **Checkout will not open:** check Website approval for the actual Public domain and retry from Billing. Do not create duplicate provider subscriptions manually.
- **Payment pending:** wait for canonical payment confirmation and check notification delivery. Zero-total card changes and credit-only updates do not count as paid renewals.
- **Deleting or changing the seller is refused:** a used connection is retained for its historical checkout/catalog bindings. Create a separate connection for another seller.

The Copy button on the platform screen copies its displayed Stripe webhook URL. If the browser blocks clipboard access, the page reports the problem and you can copy the displayed URL manually. A vendor must use the webhook URL shown in their own Store settings.

Settings

Admin Panel
The Settings page — Profile tab showing account info, avatar upload, and API details.

Profile tab

View your email address, role, and company slug. You can upload a profile avatar by clicking the camera icon on the avatar circle. Supported formats: JPG, PNG. The avatar appears in the admin sidebar.

The **API Base URL** and **Tenant Slug** shown here are what you use to construct API calls from your application.

Security tab

Changing your password

Enter your current password, then your new password. Your current password is required for every role, including SuperAdmin. A successful self-change keeps your current session; an incorrect current password changes nothing. For a new password, use 15-128 characters. Spaces and Unicode are allowed; no uppercase/lowercase/number mix is required. The server rejects common passwords and obvious account or service-name variants. Existing passwords keep working for sign-in; there is no forced rotation. Passwords are not silently trimmed, case-folded or normalized before credential hashing.

Password changes require an interactive sign-in, not an API key. If you signed up through an external identity provider and do not know a local password, use the password-reset email flow. An authorised SuperAdmin resetting a different user must provide the acting SuperAdmin's current password; that operation is audited separately and ends the target user's Admin and refresh sessions.

Multi-Factor Authentication (MFA)

PermitCore supports TOTP-based MFA (works with Google Authenticator, Authy, 1Password, Bitwarden, etc.).

To enable:

1. Go to **Settings → Security** and click **Enable MFA**
2. Scan the QR code with your authenticator app (or enter the secret key manually)
3. Enter the 6-digit code from your app to confirm setup
4. Save the one-time backup codes — each code is single-use for account recovery if you lose your device

> If you lose access to your authenticator app and all backup codes, you will need to contact support to regain access. Store backup codes in a safe location (password manager, printed and stored securely).

API Keys tab

API Keys let your server-side applications authenticate with the PermitCore admin API without a browser login. Use them for automated deployments, CI scripts, or backend integrations.

Go to **Settings → API Keys** and click **New Key**:

- **Name** — a label for your own reference (e.g. "Production server", "CI deploy")
- **Expiry** — optional date. Leave blank for keys that never expire.

After creation, the raw key is shown **once** — copy it immediately. It won't be shown again.

```
Usage
X-Api-Key: permit_live_xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
```

> **Treat API keys like passwords.** Store them in environment variables or a secrets manager — never commit them to source control. Maximum 10 active keys per account. Rotate keys regularly and revoke any that may have been exposed.

You can **Rotate** a key (generate a new one with the same settings, invalidate the old one) or **Revoke** it instantly from the API Keys list.

Session timeout

The admin panel automatically signs you out after **30 minutes of inactivity**, enforced by the server. A countdown appears during the last five minutes. Click **Continue session** to renew before expiry. Clicks, typing, scrolling, navigation and reload count as activity; mouse movement and background requests do not. Access JWTs renew separately with their existing 15-minute default.

Company Settings

Go to **Company** in the sidebar to update your company name and contact email. The **Company Slug** (the URL identifier) is set at registration and cannot be changed.

Security Policy (Company Admins only)

Enable **"Require MFA for all users"** to force every team member to set up two-factor authentication before they can access the dashboard. When this policy is on:

- Users who already have MFA enabled log in normally
- Users without MFA are redirected to the MFA setup screen immediately after entering their password
- They cannot access any other page until setup is complete

Sign-in link privacy

Direct Admin login/reset/callback pages and the standalone Customer Portal use no-referrer. Other Admin, Store and public pages use an origin-only baseline (strict-origin): the browser sends only origin, without the path or query, even for same-origin requests. Public pages also use origin-only referrers, protecting SPA transitions without reloading; actual UTM URLs and current-page event values remain

unchanged. Keep verification, reset and temporary download links private: this header does not remove the initial URL from browser history or every access log. Existing links and sign-in behavior remain compatible.

Website analytics preferences

Google website analytics is separate from your product's Admin analytics charts. Private Admin areas, known test tenants and nonproduction hosts are excluded. Team and QA browsers can open the internal-browser preference link once and reload existing tabs after saving work. This preference is shared across PermitCore subdomains using the Secure, identity-free __Secure-pc_internal cookie on HTTPS. The __Secure-pc_admin_active marker follows the Admin session deadline. Existing exclusions are retained during upgrades; reload saved tabs after updating. HTTP development uses localStorage without insecure fallback cookies. It does not change license processing, account access or billing counters. Analytics reporting failures do not fail otherwise successful license checks. See the Cookie Policy to manage it.

Eligible anonymous public pages use Consent Mode v2. Limited cookieless measurements can reach Google while analytics storage is denied, including after Reject all. Analytics cookies and custom signup/checkout events require Analytics consent; all advertising consent stays denied. Absolute internal/private/test exclusions send no Google measurements. You can reopen cookie preferences at any time.

Data & Privacy — GDPR Export

Click **"Export My Data (ZIP)"** under **Company → Data & Privacy** to download the documented account-data subset. The ZIP contains:

File	Contents
tenant.json	Your company profile and billing info
users.json	All team members (no password hashes)
products.json	All products
licenses.json	All license keys (encrypted ciphertext, never plaintext)
activations.json	Full activation history with IP, country, city, user agent
audit_logs.json	Audit trail (last 90 days)
api_keys.json	API key metadata (no secrets — only prefix and expiry)
README.txt	Explanation of each file and data deletion contact

> Review the export's coverage and exclusions when using it for an access or portability request. For data deletion requests, use **Settings → Danger Zone → Request Data Erasure**; the request is tracked for human review with a 30-day due date.

For a consistent recovery snapshot with available original Store values and stored image bytes, use the

separate acknowledged Recovery package. Its sensitive ZIP is not encrypted. Keep the original passphrase and separate signing-key export, and test an isolated restore before relying on it.

Load from Template

Admin Features

Load License from Template

License templates let you define reusable presets — max activations, expiry duration, trial settings, and version constraints — and apply them instantly when creating new licenses.

How to use

1. Go to **Templates** in the sidebar and create a template
2. When creating a new license (**Licenses → Create License**), a green **"Load from Template"** dropdown appears above the form
3. Select a template — the form auto-fills Max Activations, Expiry, Trial, and Version fields
4. Adjust any fields as needed, enter your passphrase, and generate the key

What gets auto-filled

Template field	Maps to
MaxActivations	Max Activations per Key
ExpiresAfterDays	Expires At (today + N days)
IsTrial + TrialDays	Trial License toggle + days
MinVersion / MaxVersion	Version constraints

> The passphrase and product are never set by a template — they always require explicit input for security and clarity.

Expiry Warning Widget

Admin Features

Expiry Warning Dashboard Widget

The dashboard shows a **Licenses Expiring Soon** warning panel whenever any active license expires within 30 days. It gives you immediate visibility without needing to go to the Licenses page.

What the widget shows

- **Badge: X in 7 days** — count of licenses expiring within 7 days (red)
- **Badge: X in 30 days** — total expiring within 30 days (yellow)
- List of up to 8 licenses with product name, customer email, and "in Xd" countdown
- Each row links directly to the license detail page
- Panel is hidden entirely when nothing is expiring — zero clutter

API

The expiry data is included in the standard dashboard stats response:

GET /api/dashboard/stats Response includes: { "expiringIn7Days": 2, "expiringIn30Days": 5, "expiringLicenses": [{ "id": "...", "productName": "Pro License", "expiresAt": "2026-06-15T00:00:00Z", "customerEmail": "alice@co.com" }, ...] }

Weekly Email Digest

Admin Features

Every **Monday at 09:00 UTC**, PermitCore automatically sends a weekly summary email to active CompanyAdmin users who enabled the digest. Demo/internal test tenants and reserved recipient domains are excluded. It gives a snapshot of your license health without logging in.

Email contents

Section	Data shown
Active Licenses	Current total of active licenses
Activations (7d)	Number of activations in the past 7 days
Expiring in 7 days	Count — highlighted red if greater than 0
Expiring in 30 days	Count — highlighted amber if greater than 5
Expiry table	Product, customer, and expiry date for up to 5 upcoming expiries
CTA button	Direct link to your dashboard

Anti-spam logic

If a tenant has **zero active licenses and zero activations** in the past 7 days, the digest is skipped entirely — new or inactive tenants don't receive empty emails.

Password-reset, verification and OTP messages to real domains remain available for internal accounts; reserved/demo domains remain blocked. Suppression is enforced centrally before connecting to SMTP.

Requirements

- **SMTP configured** — the deployment mail configuration (managed by the operator)
- At least one active CompanyAdmin user per tenant

> **Fully automatic:** No configuration beyond SMTP setup. The digest fires every Monday without any manual scheduling.

Expiry Notifications

Admin Features

License Expiry Notifications

PermitCore automatically sends email alerts before licenses expire — both to your **CompanyAdmin users** and (optionally) directly to **end customers**. No configuration is required beyond a working SMTP provider.

When notifications are sent

The job runs every day at **08:00 UTC**. An email is triggered each time a license hits one of three expiry thresholds:

Threshold	Recipient	Colour
30 days before expiry	CompanyAdmin users	Grey
7 days before expiry	CompanyAdmin users	Amber
1 day before expiry	CompanyAdmin users	Red

Each threshold fires once per license (within a ±1 day window to account for timing drift).

Admin notification email

CompanyAdmin users receive a summary table showing all licenses hitting a threshold that day: product name, partial key hash, expiry date with day countdown, and activation count. Multiple licenses in the same threshold window are grouped into a single email per tenant.

End-customer notifications

If a license has a **Customer Email** set, the same thresholds also trigger a shorter notification sent directly to the customer. The email is branded with your **tenant name** (not PermitCore) so it appears to come from you:

Activation with customer email — enables customer-facing expiry notifications POST /api/v1/activate { "licenseKey": "PERMIT-XXXX-XXXX-XXXX-XXXX", "deviceId": "machine-id", "userEmail": "customer@example.com" } # Or set CustomerEmail directly on the license in the admin panel

Implementation

The job is a **Hangfire recurring job** registered in Program.cs. It requires Hangfire + Valkey (the default Docker setup) and a configured mail provider.

Cron schedule — 08:00 UTC daily "0 8 * * *" # Thresholds (days before expiry) 30, 7, 1 # Implementation LicenseExpiryNotificationJob (PermitCore.Infrastructure.Services)

> **Perpetual licenses are never notified.** Only licenses with ExpiresAt set trigger expiry emails. Licenses without

an expiry date are ignored.

Difference from Weekly Digest: The weekly digest (Monday 09:00 UTC) is a summary report for admins. Expiry notifications are actionable alerts triggered by specific license thresholds — they run daily and can fire multiple times per license.

Notification Preferences

Admin Features

Each admin user can individually opt in or out of the two types of automated emails. This lets users control inbox noise for users who don't need both types of alerts.

Available toggles

Setting	Default	Controls
Weekly digest emails	On	The Monday 09:00 UTC dashboard summary email
Expiry notification emails	On	The daily 30/7/1-day expiry alert emails

Where to manage

Go to **Settings → Notifications** tab. Changes are saved per user account — they don't affect other admins in the same tenant.

API

Get current notification preferences GET /api/users/me/notifications Authorization: Bearer {token} # Response { "digestEmailEnabled": true, "expiryEmailEnabled": true } # Update preferences PATCH /api/users/me/notifications Authorization: Bearer {token} Content-Type: application/json { "digestEmailEnabled": false, "expiryEmailEnabled": true }

> Opting out only affects *that user's* emails. Other CompanyAdmin users in the same tenant continue to receive notifications. To suppress all admin emails for a tenant, all admins must individually opt out.

Extend Expiry (Bulk)

Admin Features

Extend Expiry (Bulk Action)

Select one or more licenses on the **Licenses** page and click **Extend Expiry** in the bulk action toolbar to quickly push expiry dates forward.

How it works

1. Check any licenses using the row checkboxes (or the group header checkbox to select all in a product)
2. Click **Extend Expiry** in the selection toolbar
3. Choose a preset (+30 / +60 / +90 / +180 / +365 days from today) or enter a custom date
4. Confirm — all selected licenses receive the same new ExpiresAt

> Extend Expiry sets an **absolute date** for all selected licenses, not relative to each license's existing expiry. If you need per-license relative extension, use the individual license edit form.

API

PATCH /api/licenses/bulk Authorization: Bearer <token> Content-Type: application/json { "ids": ["uuid1", "uuid2"], "expiresAt": "2027-01-01T00:00:00Z", "clearExpiry": false }

Search by User Email

Admin Features

Search Licenses by Activated User Email

Find all licenses that were activated by a specific end-user using the **"Activated by user email"** field in the Licenses page advanced filters. This uses the userEmail field captured during activation (named-user licensing).

How to use in the admin panel

1. Go to **Licenses**
2. Click the **Filters** button to expand advanced filters
3. Enter an email in the **"Activated by user email"** field
4. Click the magnifying glass — the list reloads showing only matching licenses

API

GET /api/licenses?userEmail=john@customer.com Authorization: Bearer <token> # Returns only licenses that have at least one activation # with a matching UserEmail (case-insensitive partial match)

> This requires your end-users to have activated their licenses using the userEmail field in POST /api/v1/activate. See the Named User Licensing section.

Customer Management

Admin Features

The **Customers** page (Admin → Customers) lets you create a CRM-lite record for each of your end-customers and link any number of license keys to them. This gives you a single view of how many licenses a customer holds and how many are currently active — without searching by email across the licenses table.

Customer record fields

Field	Required	Notes
Name	Yes	Contact name (e.g. Alice Johnson)
Email	Yes	Unique per tenant — used to identify the customer
Company	No	Organisation name shown next to the customer's name
Notes	No	Internal free-text notes

Linking licenses to customers

There are three ways to link a license to a customer:

1. **Manual — Customer dropdown:** When creating or editing a license, pick a customer from the **Customer** dropdown in the form.
2. **Automatic on create/update — email matching:** If a license has a **Customer Email** set and no explicit customer selected, PermitCore automatically finds or creates the matching Customer record and links it — on every POST /api/licenses, POST /api/licenses/bulk, and PUT /api/licenses/{id}. Name is derived from the email local part (e.g. john.doe@example.com → *John Doe*). You can fill in the company and notes later via Edit Customer.
3. **Bulk backfill — Sync from Licenses:** Click the **Sync from Licenses** button on the Customers page to scan all existing licenses that have a CustomerEmail set but no linked customer. Missing Customer records are created and linked in one operation. A toast confirms how many were created vs. how many were just linked (email matched an existing record).

Deleting a customer record does *not* delete its licenses — they are simply unlinked (CustomerId → null).

Viewing a customer's licenses

Click the **Licenses** button on any customer row to jump to the Licenses page pre-filtered to that customer. A blue banner at the top shows *"Showing licenses for [Name]"* with a clear-filter link. All license statuses (active, revoked, expired) are shown — not just active — so you have a complete picture of that customer's history. From there you can edit, revoke, transfer, or create new licenses as normal.

API

List customers (paginated, searchable) GET /api/customers?page=1&pageSize=50&search=alice Authorization: Bearer {token} # Get a single customer GET /api/customers/{id} # Get licenses for a customer GET /api/customers/{id}/licenses # Create POST /api/customers { "name": "Alice Johnson", "email": "alice@acme.com", "company": "ACME Corp" } # Update PUT /api/customers/{id} { "name": "Alice Johnson", "email": "alice@acme.com", "company": "ACME Corp", "isActive": true } # Delete (unlinks licenses, does NOT delete them) DELETE /api/customers/{id} # Backfill — auto-create customers from license emails POST /api/customers/backfill # Returns: { "created": 8, "linked": 3, "total": 11 }

Test / Sandbox Keys

Test API keys and sandbox licenses

These are different objects. An API key authenticates a management integration. A license key is the credential your customer uses to validate or activate software.

Object	How to identify it	What to use it for
Admin API key	permit_live_... or permit_test_...	Trusted server-to-server administration. The test prefix is not an isolated tenant or a read-only security boundary.
Sandbox license	PERMIT-... with isTest: true	Exercise license behavior; occupied test seats also count against account capacity.

Create a sandbox license in **Licenses → Create License → Test / Sandbox**. Do not attach an admin API key to public validation or activation requests to "make them test requests." See sandbox licensing.

Create management credentials under **Settings → API Keys**. Copy the raw value when shown, keep it in a secret manager, and revoke it when no longer needed. Both prefixes authenticate a privileged tenant integration; neither belongs in distributed application code.