Signal Lynx Key Commander: Product Guide & Instruction Manual

This guide is evergreen and applies to all modern releases of the Key Commander application. It will be updated periodically to reflect major feature additions and changes in functionality.

Last Updated: December 18, 2025

Overview

Welcome to the official Product Guide for Signal Lynx Key Commander, the private, self-hosted command center for your digital empire. Whether you're running a Software-as-a-Service (SaaS) or Licensing-as-a-Service (LaaS) business, Key Commander is the complete operational engine designed to handle the critical backend infrastructure—payments, license generation, automated emails, in-app communications, and secure backups.

This comprehensive manual is designed to help you install, configure, and effectively utilize Key Commander to automate your business operations with unparalleled security and privacy. It is written for users of all technical levels, explaining not just the "what" but the "why" at each stage.

Our Philosophy: Your Stack, Your Rules

Our entire ecosystem is built on a foundational principle: **your customer data should live on your server, not ours, and you shouldn't be taxed a percentage of your own success.**

Key Commander is a self-hosted product. It runs on your hardware, under your control. We provide the arsenal; you command your empire.

Architectural Overview

Key Commander consists of two distinct but connected applications:

1. **The Headless Server (The Engine):** This is the brains of the operation. A powerful FastAPI application that handles all core logic: processing payments from Stripe, interacting with the database, validating license keys, and serving the API. It is designed to be deployed as a

lightweight Docker container on a Linux VPS (recommended) or as a standalone executable on a Windows machine.

2. **The Admin Portal (The Command Deck):** This is your command center. A modern, intuitive desktop GUI for Windows built with Flet/Flutter. It connects securely to your Key Commander Server—whether it's running on a remote VPS or on your local machine—and gives you full control over every aspect of your licensing operation. All communication happens exclusively through the server's secure API.

Table of Contents

- 5.0 Before You Begin
- 5.1 API Keys (Provider Setup)
- 5.2 Products & Prices
- 5.3 Webhooks (Stripe → Key Commander)
- 5.4 Test Mode (End-to-End)
- 5.5 Troubleshooting (Stripe ↔ Key Commander)
- 5.6 What Each Event Does (Practical Effects)
- 5.7 Final Checklist (Go/No-Go)

6. First-Time System Setup (via Admin Portal)

- 6.0 Before You Begin (Checklist)
- 6.1 Connect the Admin Portal to Your Server
- 6.2 Unlock the Server (EULA → Master Password)
- 6.3 Initial Configuration (Config Tab)
    - 6.3.1 License Key
    - 6.3.2 Website API Key (Optional, Recommended)
    - 6.3.3 Database Connection (DSN)
    - 6.3.4 Integration Endpoints (Helper)
    - 6.3.5 Offline License Validation (Optional)
    - 6.3.6 Email Provider (SMTP, Brevo, or SES)
    - 6.3.7 Database Backups & Restoration
    - 6.3.8 Stripe Secrets (Provider Card)
    - 6.3.9 Aliases & In-App Messaging (Optional)
- 6.4 Verify Everything Works (Go/No-Go Checklist)

7. Feature Deep Dive: The Admin Portal

- 7.1 Licenses Tab
- 7.2 Metrics Tab
- 7.3 Aliases & Messaging Tab
- 7.4 Audit Log Tab
- 7.5 Error Log Tab
- 7.6 Reports Tab
- 7.7 License Types (Overview)
- 7.8 Machine ID (Anti-Piracy & Activation Management)
- 7.9 Anti-Trial Hopping
- 7.10 Export Database as CSV

---

1. Key Features at a Glance

Key Commander is a dense suite of hardened, operational tools that handle the mission-critical parts of your business so you can get back to building.

Revenue Operations: The Robotic Accountant

A battle-tested Stripe engine that handles subscriptions, trials, and one-time sales. It auto-issues license keys, delivers welcome emails via SMTP, Brevo, or SES, and relentlessly syncs data to ensure your records are always accurate.

Application Intel: The Comms Relay

Push targeted, in-app announcements directly to your users' software. See every administrative and system action in a detailed, searchable audit log. Your application is no longer a black box.

System Armor: The Self-Healing Backend

Automated database backups, proactive data integrity health checks, and anti-piracy tools run on autopilot. Cryptographically signed offline tokens keep your application working for your users, even when their internet doesn't.

Error Reporting: Your Stack's Flight Recorder

Key Commander comes pre-wired with a self-hosted error reporting pipeline. Your website and your client applications can send error logs directly to your server, allowing you to find and fix bugs before your users even complain.

Website Integration: The SaaS Launchkit

Plug Key Commander into any website with a clean, secure API. Use our free, production-ready SvelteKit template to get a complete storefront with user authentication, a customer dashboard, gated downloads, and self-service license management from day one.

Deployment: Linux VPS (Recommended)

This is the definitive guide for deploying the **Key Commander** server application in a secure, robust, and production-ready environment. It walks from a brand-new Linux server to a containerized app secured behind Cloudflare.

Designed for minimal Linux experience—every command is provided. By the end, you'll have a multi-layer, professional-grade backend.

Architectural Overview

We build two access planes:

- **Public Plane** – For customers, your website, and external services (e.g., Stripe). Traffic flows through Cloudflare, is filtered by security rules, and reaches your app via a secure *Cloudflare Tunnel*. Your server firewall stays closed to the public internet.

- **Admin Plane** – For you, the administrator. Use **Tailscale** (private, encrypted VPN) as a "secret passage" for SSH and the Admin Portal.

2.1 Prerequisites

- **Domain name** (e.g., `yourdomain.com`) managed in a **free Cloudflare** account

- **Cloudflare** account (free)

- **Tailscale** account (free)

- **Windows PC** for administration (PowerShell available)
- **Application artifacts** (built on Windows):
    - Docker image `.tar` file (e.g., `kc-server-lm-8m107.tar`)
    - Matching `.tar.sha256` checksum file
    - `docker-compose.prod.yml` (You must configure the provided docker-compose.prod-MOD_ME.yml to yield this file)

User-Specific Configuration (required)
Before deploying to your VPS, review and edit **docker-compose.prod-MOD_ME.yml** with your own database details.

Required change: Update Admin Plane port and Rename File

- Update your admin plan port to match your Tailscale (or other) working IP
    - `"<YOUR_TAILSCALE_IP_MOD_ME>:9000:8000"`
- Rename the file from **docker-compose.prod-MOD_ME.yml** to **docker-compose.prod.yml**

Recommended change: Database DSN (DB_DSN)
The DSN controls which PostgreSQL database the server uses (including username and password). A default is provided, but you should change the **database name**, **username**, and **password** to your own unique values.

Example:

```
DB_DSN: postgresql+asyncpg://kcadmin:strongpass!@postgres:5432/kc
```

- **Host:** `postgres` (this is the Compose service name for the PostgreSQL container)
- **User:** `kcadmin`
- **Password:** `strongpass!`
- **Database:** `kc`
- **Port:** `5432`

Match the Postgres service environment in your docker-compose.prod.yml to your DSN
Update the `postgres` service block so `POSTGRES_DB`, `POSTGRES_USER`, and `POSTGRES_PASSWORD` align with the DSN above:

```
services:
  postgres:
    image: postgres:16-alpine
    environment:
      POSTGRES_DB: kc            # Database name
      POSTGRES_USER: kcadmin     # Username
      POSTGRES_PASSWORD: strongpass!  # Password
```

Notes

- Use **lowercase** for database and user names (PostgreSQL folds unquoted identifiers to lowercase).
- If your password contains special characters, **URL-encode** them in the DSN.
- Keep these three values (**DB name, user, password**) **consistent** in both places: the `DB_DSN` and the `postgres` service environment.

2.2 Phase 1: Local Preparation (Windows PC)

2.2.1 Generate SSH Keys (PowerShell)

ssh-keygen -t ed25519 -C "KeyCommander-Production-VPS"

Press **Enter** three times to accept defaults (location, no passphrase).

This creates in `C:\Users\<YourName>\.ssh\`:

- `id_ed25519` (private – **never share**)
- `id_ed25519.pub` (public)

Critical: Back up your entire `.ssh` folder to a secure USB drive.

2.2.2 Install Local Tools

- Install **Tailscale** on Windows and sign in.

2.3 Phase 2: Server Provisioning & Initial Hardening

2.3.1 Choose a VPS Provider

Example: **OVHcloud** (VPS-2). Recommended baseline:

- **OS:** Ubuntu 24.04 LTS
- **CPU/RAM:** 2+ vCPUs, 4+ GB RAM
- **Storage:** 80+ GB NVMe SSD
- **Region:** Closest to your customers

2.3.2 First Login & User Setup

Your provider emails an IP and initial credentials (often user `ubuntu`).

Connect from Windows PowerShell:

ssh ubuntu@<YOUR_VPS_PUBLIC_IP>

Accept host prompt; change any initial password if asked.

Create your permanent admin user (replace with your name):

sudo adduser <ADMIN_USERNAME>
sudo usermod -aG sudo <ADMIN_USERNAME>

2.3.3 Install Your SSH Key

Switch to your new user:

sudo su - <ADMIN_USERNAME>

Create the SSH directory and set permissions:

mkdir -p ~/.ssh
chmod 700 ~/.ssh

Add your public key:

nano ~/.ssh/authorized_keys

Paste the contents of your local `id_ed25519.pub`. Save and exit (**Ctrl+X**, **Y**, **Enter**), then:

chmod 600 ~/.ssh/authorized_keys
exit

2.3.4 Harden the SSH Server

Edit `sshd_config`:

sudo nano /etc/ssh/sshd_config

- Set: `PermitRootLogin no`
- Leave `PasswordAuthentication yes` (emergency fallback)

Restart SSH:

sudo systemctl restart ssh

2.3.5 Test SSH Key Login

Open a **new** PowerShell window and test:

ssh <ADMIN_USERNAME>@<YOUR_VPS_PUBLIC_IP>

If it works, close the original `ubuntu` session.

2.3.6 System Updates

sudo apt update
sudo apt upgrade -y

If a kernel updates, reboot:

sudo reboot

2.4 Phase 3: Security & Networking

2.4.1 Install Security Tools

sudo apt install fail2ban ufw -y

`fail2ban` starts protecting SSH immediately.

2.4.2 Configure the Firewall (ufw)

Default deny inbound:

sudo ufw default deny incoming

Allow SSH:

sudo ufw allow ssh

Allow Admin Portal via Tailscale (port **9000**):

sudo ufw allow in on tailscale0 to any port 9000 proto tcp

Enable and check:

sudo ufw enable
sudo ufw status numbered

2.4.3 Install & Link Tailscale

curl -fsSL https://tailscale.com/install.sh | sh
sudo tailscale up

Note the server's Tailscale IP (e.g., `100.x.x.x`). Test SSH using it:

ssh <ADMIN_USERNAME>@<YOUR_TAILSCALE_IP>

2.5 Phase 4: Application Environment

2.5.1 Install Docker & Docker Compose v1

sudo apt install docker.io docker-compose -y

Add your user to the `docker` group:

sudo usermod -aG docker <ADMIN_USERNAME>

Log out and back in for group changes to apply.

2.5.2 Set Up the Cloudflare Tunnel

Install `cloudflared`:

curl -L --output cloudflared.deb \
  https://github.com/cloudflare/cloudflared/releases/latest/download/cloudflared-linux-amd64.deb
sudo dpkg -i cloudflared.deb

Authenticate `cloudflared` with your Cloudflare account:

cloudflared tunnel login

Create a tunnel and note its UUID:

cloudflared tunnel create key-commander-tunnel

Create and edit the config:

sudo nano /etc/cloudflared/config.yml

Paste and adjust placeholders for the below file (provided in the zip package):

#
==========
======
TEMPLATE FOR CLOUDFLARE TUNNEL CONFIGURATION - RENAME TO config.yml

```
#
================================================================================
======
# This is a template file for configuring the Cloudflare Tunnel (`cloudflared`) to
# securely expose your Key Commander server to the internet.
#
# BEFORE YOU USE THIS, YOU MUST:
# 1. Place this file on your server. A common location for a system-wide service
#    is `/etc/cloudflared/config.yml`.
# 2. Modify ALL values marked with `<...>`.
# 3. Ensure your `cloudflared` service is configured to use this file.
#
# USAGE:
# a. After creating your tunnel (`cloudflared tunnel create <tunnel_name>`),
#    you will receive a Tunnel ID and a credentials file. Use them below.
# b. After editing, restart the service: `sudo systemctl restart cloudflared`
#
================================================================================
======

# --- Tunnel Identification ---
# Replace <YOUR_TUNNEL_ID> with the UUID of your tunnel from the
# `cloudflared tunnel create ...` command output.
tunnel: <YOUR_TUNNEL_ID>

# The path to your tunnel's credentials file. This path MUST be correct.
# - If you ran `cloudflared` as a regular user (e.g., 'myuser'), this is typically:
#   /home/<YOUR_LINUX_USERNAME>/.cloudflared/<YOUR_TUNNEL_ID>.json
# - If you configured it system-wide as root, it might be:
#   /root/.cloudflared/<YOUR_TUNNEL_ID>.json
# Check the output from the `cloudflared tunnel create ...` command for the exact path.
credentials-file: /home/<YOUR_LINUX_USERNAME>/.cloudflared/<YOUR_TUNNEL_ID>.json

# Stabilizers (reduce tunnel latency spikes / 520s)
protocol: http2
edge-ip-version: "4"
ha-connections: 8

# --- Ingress Rules: How traffic reaches your server ---
# The rules below define which public requests are allowed to reach your application.
ingress:
  # --- Public API Routing ---
  # All rules below MUST use your public domain name.
  # Replace `<YOUR_PUBLIC_API_DOMAIN>` everywhere with your domain
```

```
# (e.g., api.myproduct.com).

# === Health Check Endpoints ===
- hostname: <YOUR_PUBLIC_API_DOMAIN>
  path: /ping
  service: http://localhost:8000
- hostname: <YOUR_PUBLIC_API_DOMAIN>
  path: /status
  service: http://localhost:8000
- hostname: <YOUR_PUBLIC_API_DOMAIN>
  path: /api/v1/system/healthz
  service: http://localhost:8000

# === Public Client Endpoints ===
- hostname: <YOUR_PUBLIC_API_DOMAIN>
  path: /api/v1/system/status
  service: http://localhost:8000
- hostname: <YOUR_PUBLIC_API_DOMAIN>
  path: /api/v1/system/setup
  service: http://localhost:8000
- hostname: <YOUR_PUBLIC_API_DOMAIN>
  path: /api/v1/system/unlock
  service: http://localhost:8000
- hostname: <YOUR_PUBLIC_API_DOMAIN>
  path: /api/v1/system/configure-database
  service: http://localhost:8000
- hostname: <YOUR_PUBLIC_API_DOMAIN>
  path: /api/v1/validate/*
  service: http://localhost:8000
- hostname: <YOUR_PUBLIC_API_DOMAIN>
  path: /api/v1/license/status
  service: http://localhost:8000
- hostname: <YOUR_PUBLIC_API_DOMAIN>
  path: /api/v1/license/activate
  service: http://localhost:8000
- hostname: <YOUR_PUBLIC_API_DOMAIN>
  path: /api/v1/report-error
  service: http://localhost:8000

# === Internal Website Endpoints ===
- hostname: <YOUR_PUBLIC_API_DOMAIN>
  path: /api/v1/internal/user-entitlements/*
  service: http://localhost:8000
- hostname: <YOUR_PUBLIC_API_DOMAIN>
```

```
    path: /api/v1/internal/licenses/licenses-by-email/*
    service: http://localhost:8000
  - hostname: <YOUR_PUBLIC_API_DOMAIN>
    path: /api/v1/internal/licenses/claim-by-email
    service: http://localhost:8000
  - hostname: <YOUR_PUBLIC_API_DOMAIN>
    path: /api/v1/internal/licenses/reset-activations-by-user-id
    service: http://localhost:8000
  - hostname: <YOUR_PUBLIC_API_DOMAIN>
    path: /api/v1/internal/users/public/reset-activations
    service: http://localhost:8000
  - hostname: <YOUR_PUBLIC_API_DOMAIN>
    path: /api/v1/report-website-error
    service: http://localhost:8000

  # === Webhook Endpoints ===
  - hostname: <YOUR_PUBLIC_API_DOMAIN>
    path: /api/v1/webhook/*
    service: http://localhost:8000

  # === Default Deny Rule (DO NOT CHANGE) ===
  # This is a critical security feature. Any request that does not match one of
  # the specific `hostname` and `path` rules above will be blocked by Cloudflare
  # with a 404 error before it ever reaches your server. This prevents scanners
  # and bots from probing for vulnerabilities on unexposed paths like /.env.
  - service: http_status:404
```

Create DNS route for your public hostname (e.g., `api.yourdomain.com`):

cloudflared tunnel route dns key-commander-tunnel <API_HOSTNAME>

Install and start the service:

sudo cloudflared service install
sudo systemctl start cloudflared

2.6 Cloudflare Edge Security

In Cloudflare dashboard → your domain → **Security ▸ WAF**:

- **Bot Fight Mode:** Enable.
- **Rate Limiting Rule (example):**
    - **Rule name:** Global API Rate Limiting
    - **Expression:**
        - `(http.request.uri.path contains "/api/v1/validate") or (http.request.uri.path contains "/api/v1/report-error")`
    - **Rate:** 10 requests per 10 seconds
    - **Action:** Block for 10 seconds

Adjust paths to match your critical endpoints.

2.7 Application Deployment & Management

2.7.1 Note on Docker Compose v1

Ubuntu's `apt` provides **v1**, so commands use a hyphen: `docker-compose`.

2.7.2 Initial Deployment

Create directories and set ownership:

```
sudo mkdir -p /opt/kc/server/{data,storage}
sudo chown -R <ADMIN_USERNAME>:<ADMIN_USERNAME> /opt/kc
```

From **Windows PowerShell**, transfer artifacts to your server's **Tailscale** IP:

```
scp ".\dist\Key_Commander_Server.tar" `
   ".\dist\Key_Commander_Server.tar.sha256" `
   ".\docker-compose.prod.yml" `
   <ADMIN_USERNAME>@<YOUR_TAILSCALE_IP>:/opt/kc/
```

On the server, deploy:

```
cd /opt/kc
sha256sum -c Key_Commander_Server.tar.sha256
docker load -i Key_Commander_Server.tar
docker-compose -f docker-compose.prod.yml up -d
```

Connect the Admin GUI: `http://<YOUR_TAILSCALE_IP>:9000`

2.7.3 Hot-Fix Update Procedure

```
cd /opt/kc
docker-compose -f docker-compose.prod.yml down
# Optionally remove the old image tag (edit tag accordingly):
# docker rmi key-commander-server:<old_version_tag>

# Re-transfer new artifacts from Windows as in 2.7.2, then:
sha256sum -c Key_Commander_Server.tar.sha256
docker load -i Key_Commander_Server.tar
docker-compose -f docker-compose.prod.yml up -d
```

2.7.4 Day-to-Day Operations

```
# Status of services
docker-compose -f docker-compose.prod.yml ps

# Follow logs for main server
docker-compose -f docker-compose.prod.yml logs -f server

# Stop / Start / Restart
docker-compose -f docker-compose.prod.yml stop
docker-compose -f docker-compose.prod.yml start
docker-compose -f docker-compose.prod.yml restart
```

2.8 Viewing Application Logs

All commands from `/opt/kc` on the server.

Base form:

```
docker-compose -f docker-compose.prod.yml logs [OPTIONS] [SERVICE_NAME]
```

- Omit `SERVICE_NAME` to see logs for all services (e.g., `server`, `postgres`, `redis`).

Common cases:

```
# Last 200 lines (main app)
docker-compose -f docker-compose.prod.yml logs --tail=200 server

# Live stream (Ctrl+C to stop)
docker-compose -f docker-compose.prod.yml logs -f server

# Postgres / Redis
docker-compose -f docker-compose.prod.yml logs -f postgres
docker-compose -f docker-compose.prod.yml logs -f redis

# Quick checks
docker-compose -f docker-compose.prod.yml logs --tail=500 server
docker-compose -f docker-compose.prod.yml ps
```

---

3. Deployment: The Windows Server

This guide covers installing and running the **Key Commander** server on Windows 10/11 or Windows Server. It is an alternative to the Linux VPS deployment for teams that prefer a Windows-based backend.

The server is the backend for license management, validation, and data storage. Plan to run it persistently on a machine you control.

3.1 System Requirements

- **OS:** Windows 10 / Windows 11 / Windows Server 2016+ (64-bit)
- **CPU/RAM:** 2+ vCores, 4+ GB RAM
- **Storage:** 80+ GB NVMe SSD
- **Internet:** Required for activation and communication with providers

---

3.2 Phase 1: Prerequisites Installation

We recommend **Chocolatey** to simplify installs.

3.2.1 Install Chocolatey (once)

Open **PowerShell as Administrator** and run:

Set-ExecutionPolicy Bypass -Scope Process -Force;
[System.Net.ServicePointManager]::SecurityProtocol =
[System.Net.ServicePointManager]::SecurityProtocol -bor 3072; iex ((New-Object
System.Net.WebClient).DownloadString('https://community.chocolatey.org/install.ps1'))

Close and reopen PowerShell **as Administrator** so `choco` is in `PATH`.

3.2.2 Install Microsoft Visual C++ Redistributable

Using Chocolatey (recommended):

choco install vcredist140 -y

Or download the **x64** installer from Microsoft and run it.

3.2.3 Install PostgreSQL 16

Using Chocolatey (recommended):

choco install postgresql16 -y

Or download **PostgreSQL 16 (x64)** and install manually. During setup, set a password for the default `postgres` user and keep it safe.

Create a dedicated user and database:

1. Open **SQL Shell (psql)** from Start Menu.
2. Press **Enter** through Server/Database/Port/Username prompts until asked for the `postgres` password. Enter it.
3. Run:

-- Create a dedicated user for the application

CREATE USER kc WITH PASSWORD 'kcpass';

-- Create the database and assign ownership to the new user

CREATE DATABASE kc OWNER kc;

Type `\q` and press Enter to exit.

3.2.4 Install Redis (Memurai for Windows)

Using Chocolatey (recommended):

choco install memurai -y

Or download **Memurai** and run the installer. It will configure a Windows Service to start automatically.

3.3 Phase 2: Server Application Installation

Run the provided installer **`KeyCommanderServer-Setup.exe`**.

- The default path is: `C:\SignalLynx\KeyCommanderServer`
- Shortcuts are added to Start Menu and Desktop.

3.4 Phase 3: First-Time Configuration (Limited Mode)

On first run, the server needs database settings via the Admin Portal.

1. Launch **KeyCommanderServer.exe** (from install directory or Start Menu). A console window appears.
2. Launch **KeyCommanderGUI.exe**.
3. The Admin Portal detects **Limited Mode** and prompts for the database DSN.
4. In **PostgreSQL Connection URL (DSN)** enter exactly:

postgresql+asyncpg://kc:kcpass@localhost:5432/kc

5. Click **Save DSN** (the GUI will confirm).
6. **Restart the server**: close the `KeyCommanderServer.exe` console window and run it again. It should now connect to the DB and enter **Normal Mode**.

3.5 Phase 4: Run the Server Persistently

Option A — Manual (simple)

Run `KeyCommanderServer.exe` and leave the window open. Closing it or rebooting stops the server. Suitable for testing only.

Option B — Windows Service (recommended)

Use **NSSM** to run the server in the background and auto-start on boot.

Install NSSM:

choco install nssm -y

Register and start the service (PowerShell as Administrator):

Install the service with a friendly name

nssm install KeyCommanderServer

Point to the executable

nssm set KeyCommanderServer Application "C:\SignalLynx\KeyCommanderServer\KeyCommanderServer.exe"

Set the working directory

nssm set KeyCommanderServer AppDirectory "C:\SignalLynx\KeyCommanderServer"

Start the service

nssm start KeyCommanderServer

Manage with Windows Services, `sc` commands, or NSSM:

sc stop KeyCommanderServer

nssm stop KeyCommanderServer

3.6 Phase 5: Networking & Firewall

By default the server listens **only on** `localhost:8000` (not accessible externally). To expose it publicly, use a **tunnel** or a **reverse proxy**.

Scenario 1 — Tunnel (e.g., Cloudflare Tunnel)

A tunnel client (like `cloudflared.exe`) makes an **outbound** connection. Your domain points to that tunnel.

- **Firewall:** No inbound rule is required (simplest & most secure).

Scenario 2 — Reverse Proxy (e.g., Nginx, Caddy)

A reverse proxy listens on a public port (e.g., 443 for HTTPS) and forwards to `localhost:8000`.

- **Firewall:** Create an **inbound** rule for the proxy's public port (not for `KeyCommanderServer.exe`).

Open port **443** for HTTPS (PowerShell as Administrator):

New-NetFirewallRule -DisplayName "Web Server (HTTPS)" -Direction Inbound -Protocol TCP -LocalPort 443 -Action Allow

---

4. Installing the Admin Portal

The **Admin Portal** is a Windows-only desktop application used to manage your Key Commander server (whether it's on a remote VPS or on your local machine).

4.0 System Requirements

What it is

Minimum environment to install and run the Admin Portal smoothly.

What to do

- Windows 10/11 (64-bit) or Windows Server 2016+.
- 2+ cores, 4+ GB RAM, ~300 MB free disk space (more for logs/exports).
- Outbound HTTPS access to your server's base URL.
- (If you plan to use the admin plane via Tailscale) Tailscale installed and signed in on your Windows PC.

4.1 Get the Installer

What it is

The Admin Portal installer package.

What to do

- Locate the Windows installer (e.g., **KeyCommanderGUI-Setup.exe**) from your download link or internal share.
- (Optional) Verify the file's SHA-256 checksum if one is provided alongside the download.

4.2 Run the Installer

What it is

Standard Windows installation flow.

What to do

1. **Double-click** `KeyCommanderGUI-Setup.exe`.

    - If Windows SmartScreen prompts, click **More info** → **Run anyway** (if you trust the source).
2. Follow the wizard prompts:
    - Choose the **install folder** (default is fine for most users).
    - Choose whether to **create Desktop/Start Menu shortcuts**.
3. Click **Install**. When complete, click **Finish**.

Notes

- You may be asked for **Administrator approval** during install (standard behavior).
- No admin rights are required to **use** the app after installation.

4.3 Launch the Admin Portal

What it is
Starting the application the first time.

What to do

- From the **Desktop** or **Start Menu**, launch **Key Commander Admin Portal**.
  —or—
- Open your chosen install folder and double-click the main executable (e.g., `KeyCommanderGUI.exe`).

What you should see on first launch

- The app opens to a welcome/login screen or directly into the **Config** tab.

- If the backend isn't configured yet, you may see a "backend unreachable" banner—this is normal until you point the GUI at your server (see 4.5).

4.4 Post-Install File Locations (for reference)

What it is

Where things typically live on disk.

What to know

- **Application files:** your chosen install folder (e.g., `C:\Program Files\KeyCommander\AdminPortal\` or a custom path).
- **User settings / preferences / logs:** stored under your Windows user profile (roaming/local app data) or the app's `data` subfolder, depending on build options.
- **Exports (CSV, etc.):** wherever you choose when saving.

(You don't need these paths for normal usage; they are helpful for backups or troubleshooting.)

4.5 First-Launch Checklist (Quick)

What it is

A short path to get from "fresh install" to ready for use.

What to do

1. **Boot the Server**
2. **Boot the Admin Portal**
    - Accept the **EULA** (first start only):
3. **Point the GUI at your server**:
    - Enter your **server base URL**:
        - **Linux VPS via Tailscale (recommended deployment):**
          `http://1XX.1XX.X.XX:9000`
          `1. This is your tailscale IP`
        - **Windows Hosted Server (.exe):**
          `http://127.0.0.1:8000`
        - **Docker on Windows (Compose network):**
          `http://server:8000`
    - Save/Connect.
4. **Unlock the server**:
    - Create/enter the **Master Password** (store it safely; it cannot be recovered).

5. Proceed with your **Initial Configuration** (see Section 6).

4.6 Optional: Connect Over Tailscale (Admin Plane)

What it is

A private path to the server that avoids exposing admin traffic to the public internet.

What to do

- Install and sign in to **Tailscale** on your Windows PC.
- Use the server's **100.x.y.z** Tailscale IP and mapped **admin port** (commonly `9000`) as your backend URL in the GUI, e.g.:

  `http://1XX.1XX.X.XX:9000`
  - `This is your tailscale IP`

Why

- Keeps admin traffic private and typically more reliable than public DNS during early setup.

4.7 Keeping the Admin Portal Up-to-Date

What it is

How to upgrade without losing settings.

What to do

- Close the Admin Portal.
- Run the **new** installer over the existing install (in-place upgrade).
- Your saved preferences (backend URL, window layout, etc.) are preserved.

Notes

- If you use automated tooling (RMM/IT), a **silent install** can be supported depending on your installer build options. Coordinate with your packaging team.

4.8 Uninstall / Repair

What it is

Removing or repairing the installation.

What to do

- **Uninstall:** Windows **Settings** → **Apps** → **Installed apps**, find **Key Commander Admin Portal**, click **Uninstall**.
  —or— Use the **Uninstall** shortcut in the Start Menu (if present).
- **Repair/Reinstall:** Run the same installer and choose **Repair** (if offered) or install again to overwrite.

4.9 Quick Troubleshooting

What it is

Common first-run issues and fast fixes (see Section 9 for full details).

What to do

- **Backend unreachable:** Recheck the **server URL** (Config → Backend/Server Connection).
  - Common Server
    - **Linux VPS via Tailscale (recommended deployment):**
      `http://1XX.1XX.X.XX:9000`
      - `This is your tailscale IP`
    - **Windows Hosted Server (.exe):**
      `http://127.0.0.1:8000`
    - **Docker on Windows (Compose network):**
      `http://server:8000`
- **Stuck in Limited Mode / DB not configured:** Provide a valid **PostgreSQL DSN** and **restart** the server; then reconnect.

- **“Key Loading Error” / Incorrect Password:** Retry the **Master Password**. If forgotten, you must wipe server data and re-initialize (then restore from backup + reconcile).
- **SmartScreen blocked the installer:** Click **More info** → **Run anyway** if you trust the source.

---

5. Stripe Configuration

Use this section to take Stripe from blank to fully wired into Key Commander. Follow the steps in order; each subsection includes **what it is**, **what to do**, and any **gotchas**.

5.0 Before You Begin

What it is
Prerequisites you’ll want in place before configuring Stripe.

What to do

- Ensure your **Key Commander server** is reachable over HTTPS on your public plane.
- Have Admin Portal access (so you can add the Stripe provider and paste secrets).
- Decide whether you’re starting in **Test mode** or **Live mode** in Stripe.

5.1 API Keys (Provider Setup)

What it is
Your Stripe **Secret key** (sk_…) allows Key Commander to reconcile purchases and subscriptions. The **Publishable key** (pk_…) is used by your own checkout UI, not by Key Commander.

What to do

1. In **Stripe Dashboard** → **Developers** → **API keys**:
    - Locate **Secret key** (sk_…).
    - (Optional) Note your **Publishable key** (pk_…) for your own website checkout.
2. In **Admin Portal** → **Payments / Providers**:

    - Click **Add Provider** → choose **Stripe**.
    - Paste your **Secret key (sk_…)**.
    - Click **Save** (you'll paste the webhook **Signing secret (whsec_…)** later in 5.3).

Notes

- Key Commander does **not** need your publishable key.
- Keep the **sk_…** private; rotate if it leaks.

5.2 Products & Prices

What it is
Stripe **Products** represent what you sell; **Prices** represent billing plans (recurring or one-time). Key Commander links license tiers to **price_…** IDs.

What to do

1. In **Stripe Dashboard** → **Products**:
    - Create one **Product** per app/package/tier.
2. Within each Product:
    - Create one or more **Prices**:
        - **Subscription** (monthly, annual, etc.).
        - **One-time** (perpetual license).
3. Record IDs:
    - **Product ID** (prod_…)
    - **Price ID** (price_…) → this is what ties Stripe purchases to Key Commander tiers.

Tips

- In Admin Portal → **Aliases & Messaging**, add **friendly names** for prod_/price_ IDs (e.g., "Pro Monthly") so operators don't have to read raw IDs.

5.3 Webhooks (Stripe → Key Commander)

What it is
Webhooks tell Key Commander about checkouts, renewals, cancellations, and payment failures. Without webhooks, your database will drift.

What to do

A) Create the endpoint in Stripe

1. **Stripe Dashboard** → **Developers** → **Webhooks** → **Add endpoint**.
2. **Endpoint URL** (public plane of your server):

```
https://YOUR_DOMAIN/api/v1/webhook/stripe
```

   - Use lowercase `/stripe`.
   - If using Cloudflare Tunnel, ensure this exact path is **allow-listed**.
3. **Select events** (minimum recommended set):
   - **Checkout**
     - `checkout.session.completed`
   - **Customer**
     - `customer.subscription.created`
     - `customer.subscription.updated`
     - `customer.subscription.deleted`
   - **Invoice**
     - `invoice.payment_succeeded`
     - `invoice.payment_failed`
     - `invoice.paid`
4. Click **Add endpoint** to save.
5. On the endpoint details page, **copy the Signing secret** (`whsec_…`).

B) Paste the signing secret in Admin Portal

1. **Admin Portal** → **Payments / Providers** → Stripe card.
2. Paste **Webhook Signing Secret (whsec_…)** → **Save**.

Cloudflare / WAF notes

- Allow the path `/api/v1/webhook/stripe`.
- Do **not** challenge this path with bot/captcha rules (Stripe calls are server→server).
- Keep reasonable rate limits and logging on other routes.

5.4 Test Mode (End-to-End)

What it is
A safe way to validate checkout and webhook flows before going live.

What to do

- In Stripe, switch to **Test mode**.
- Use Stripe's common test card:
    - **Number:** 4242 4242 4242 4242
    - **Expiry:** any future date (e.g., 12/30)
    - **CVC:** any 3 digits (e.g., 123)

Quick subscription test

1. Create a **test Product** + **recurring Price**.
2. Complete a **test Checkout** for that Price.
3. (Optional) In Stripe → **Send test event**: `checkout.session.completed`.
4. In Admin Portal:
    - **Metrics** → **Recent Creations**: confirm a new license.
    - If not visible yet, click **Fast Reconciliation** (last 24h) or **Full Manual Reconciliation**.

Quick one-time test

- Use a **one-time Price** in Checkout. When payment succeeds, a **perpetual** active license should appear.

5.5 Troubleshooting (Stripe ↔ Key Commander)

"Invalid Stripe signature"

- The **whsec_…** signing secret is missing or wrong in the Admin Portal. Re-paste and **Save**.

No events received / 404 on webhook

- Confirm the endpoint is exactly:

```
https://YOUR_DOMAIN/api/v1/webhook/stripe
```

- Ensure your tunnel/WAF **allows** this path (no browser challenges).

Webhook accepted, but nothing changes

- Use **Fast Reconciliation** (last 24h).
- For larger gaps or after a database restore, run **Full Manual Reconciliation**.

403 from webhook endpoint

- Make sure your **Key Commander license** key is applied in Admin → Config; webhooks require a valid KC license.

5.6 What Each Event Does (Practical Effects)

`checkout.session.completed`

- `mode = subscription` + `payment_status = paid` → license starts as **trial** or **active** (based on subscription status).
- `mode = payment` + `payment_status = paid` → **one-time** purchase → **perpetual** active license.

`customer.subscription.created / updated / deleted`

- Updates license state and renewal/trial dates to match the subscription's status and period ends.

`invoice.payment_succeeded / invoice.payment_failed` (and/or `invoice.paid`)

- Advances renewals on success; flags payment issues on failure.

5.7 Final Checklist (Go/No-Go)

- **Secret key (sk_…)** saved in Admin Portal → Stripe provider.
- Webhook endpoint created at:

  `https://YOUR_DOMAIN/api/v1/webhook/stripe`

- **Signing secret (whsec_…)** pasted and **Saved** in Admin Portal.
- **Products & Prices** exist; you have the correct **price_…** IDs.
- **Test checkout** produces a license (or appears after **Fast/Full Reconciliation**).
- Cloudflare/WAF **allows** `/api/v1/webhook/stripe` without browser challenges.

Stripe is now configured for Key Commander.

---

6. First-Time System Setup (via Admin Portal)

Use the **Windows Admin Portal** to connect to your server, unlock it, and load the essentials. Follow the steps in order.

6.0 Before You Begin (Checklist)

- You can open **KeyCommanderGUI.exe** on Windows.
- Your server is running and reachable (Docker or native).
- If you plan to use Stripe, have:
  - **Stripe Secret Key** (starts with `sk_…`)
  - **Webhook Signing Secret** (starts with `whsec_…`)
- If you plan to send emails, have credentials for **SMTP**, **Brevo**, or **Amazon SES**.
- If you're on Linux VPS + Tailscale, you know your **Tailscale IP** (e.g., `100.x.y.z`) and the mapped **admin port** (typically `9000`).

6.1 Connect the Admin Portal to Your Server

1. **Launch the Admin Portal** on Windows.
2. Go to **Config → Backend/Server Connection** and choose **Remote Backend**.
3. Enter the **base URL** for your server. Common examples:
   - **Linux VPS via Tailscale (recommended deployment):**
     `http://1XX.1XX.X.XX:9000`
     i. This is your tailscale IP
   - **Windows Hosted Server (.exe):**
     `http://127.0.0.1:8000`
   - **Docker on Windows (Compose network):**
     `http://server:8000`
4. Click **Save** (or **Connect**).
   - On the very first run, seeing "backend not found" before you set this URL is normal.
   - When connected, the **Licenses** tab will appear (and will be empty on a brand-new system).

Why this matters: The Admin Portal is a client. It must know the server's base URL before anything else will work.

6.2 Unlock the Server (EULA → Master Password)

1. **EULA Approval**
    - Read and accept the End User License Agreement when prompted.
2. **Create the Master Password**
    - Enter a strong password and confirm.
    - This master password **encrypts your server's secrets** (API keys, signing material, etc.) and **cannot be recovered** by anyone.
    - Store it securely (password manager). If lost, you must wipe server data and re-initialize.
3. After successful unlock, the Admin Portal will proceed to the main interface.

Why this matters: The unlock step establishes your admin control and enables the server's encrypted configuration.

6.3 Initial Configuration (Config Tab)

Enter the essentials on the **Config** tab. Save after each subsection.

6.3.1 License Key

- **What to do:** Paste the **Key Commander license key** you received after purchase.
- **Where to find it:** Your Signal Lynx account downloads page.
- **Why:** Enables full functionality and entitlements in production.

6.3.2 Website API Key (Optional, Recommended)

- **What to do:** Generate or paste a **Website API Key** for server-to-server calls (e.g., your website using Key Commander's internal endpoints).
- **Why:** Needed for entitlement lookups from your site or other trusted backends.
  Not required for the basic client-license validation flow.

6.3.3 Database Connection (DSN)

- **Docker (prod) installs:** Usually set via `docker-compose.prod.yml`; no action needed here unless you're customizing.

Windows or custom installs: Provide the full **PostgreSQL DSN**:

```
postgresql+asyncpg://<user>:<password>@<host>:<port>/<database>
```

Examples

```
# Localhost
postgresql+asyncpg://kc_user:StrongPass!@localhost:5432/key_commander

# Docker Compose (service "db")
postgresql+asyncpg://kc_user:StrongPass!@db:5432/key_commander
```

- **Notes**
  - PostgreSQL database names and users are typically **lowercase**.
  - If you set the DSN in a "Database Not Configured" screen, you will need to **restart the server** to move from **Limited Mode** → **Normal Mode**.

6.3.4 Integration Endpoints (Helper)

- **What to do:** Enter your base domain or IP to preview common public/internal endpoint URLs.
- **Why:** This is **informational only** to help you understand routing. It does **not** change server behavior.

6.3.5 Offline License Validation (Optional)

Enable signed offline tokens for users who work air-gapped or with intermittent connectivity.

1. **Set the expiration window** (in hours), e.g., `72`.
2. In **Enter New Signing Passphrase**, either:
   - Paste your own strong passphrase, **or**
   - Click **Generate** to create a random one.
3. Click **Save Signing String** (this derives the ED25519 keys and stores them securely).
4. Click **Save & Update Offline Token**.

Important: Changing the signing passphrase **invalidates all previously issued offline tokens** (this is intentional for security).
Why: Offline tokens let clients operate without constant internet, while still expiring and proving authenticity.

6.3.6 Email Provider (SMTP, Brevo, or SES)

Configure email so the system can send license keys and notifications automatically.

1. Choose a provider:
    - **SMTP:** host, port, username, password, TLS/SSL, “From” email.
    - **Brevo:** API key, “From” email.
    - **Amazon SES:** Access Key, Secret, Region, “From” email.
2. **Enable Email Notifications** (checkbox).
3. Click **Save Provider Settings**.
4. Click **Send Test Email** to confirm delivery.
    - If test fails, re-check credentials, TLS/ports, and “From” domain setup.

Why: Automated delivery reduces manual effort and speeds up onboarding for your customers.

6.3.7 Database Backups & Restoration

- **Automated backups:** Runs at **4:00 AM (server time)**.
    1. Configure retention in the GUI; older backups are pruned automatically.
- **Manual backups:** Can be run at any time; also subject to retention.
- **Restore from backup:**
    1. Choose a backup and click **Restore**.
    2. After restore, immediately run a **full manual reconciliation** (see Stripe section) to pull in any purchases/changes that happened after the backup timestamp.

Why: Backups protect against data loss. Reconciling after restore resynchronizes purchases/licenses that occurred later.

6.3.8 Stripe Secrets (Provider Card)

1. Go to **Payments / Providers** → **Add Provider** → choose **Stripe**.
2. Paste:
    - **Stripe Secret Key** (`sk_…`)
    - **Webhook Signing Secret** (`whsec_…`)
3. Click **Save**.

Also ensure you set a **Stripe webhook** in your Stripe Dashboard that points to your server’s Stripe webhook endpoint, for example:

`https://YOUR_DOMAIN/api/v1/webhook/stripe`

(If you front with Cloudflare, make sure this path is allowed through your tunnel/WAF policy.)

Why: The secret key allows secure API calls; the webhook signing secret ensures only genuine Stripe events are processed.

6.3.9 Aliases & In-App Messaging (Optional)

- **Aliases:** Replace raw product IDs (e.g., `prod_...`) with friendly names (e.g., “Pro Monthly”) in the GUI.
    - Go to **Alias & Messaging** → **Add Alias** → choose the product → enter the friendly name → **Save**.
- **In-App Messages:** Send announcements directly into your client app.
    - **Add Message** → fill in content and targeting → toggle **Enabled** → **Save**.

Why: Aliases make the UI easier to work with; messaging keeps customers informed without email.

6.4 Verify Everything Works (Go/No-Go Checklist)

- **Admin Portal is connected** (no “backend unreachable” warnings).
- **Server is unlocked** and shows main tabs without “Limited Mode” prompts.
- **License Key** saved and valid.
- **Website API Key** saved (if using entitlements from your website/other backend).
- **DB DSN** correct; server is in **Normal Mode**.
- **Email provider** saved and **test email** delivered.
- **Backups** show next run at **4:00 AM (server time)**; retention set.
- **Stripe provider** saved with valid `sk_…` and `whsec_…`.
- **Stripe webhook** configured to your public endpoint; test event reaches the server.
- **Offline license** (if enabled): passphrase saved, expiration set, public key visible.

---

7. Feature Deep Dive: The Admin Portal

The **Admin Portal** is your mission control for operating Key Commander. Use it to manage licenses, monitor health, reconcile payments, communicate with users, and diagnose issues.

7.1 Licenses Tab

This is your primary workspace for managing customer licenses.

What you can do

- **Search & Filter:**
  Search by **email**, **license key**, **machine ID**, **Stripe product/price/subscription ID**, or **status** (Active, Trial, Inactive, Blocked).
  Filters can be combined to narrow results quickly.

- **Create / Edit / Delete:**
  - **Create** new licenses (for manual sales, promotions, dev/test keys).
  - **Edit** existing licenses (update customer email, status, notes).
  - **Delete** removes a license record from your database. Use with care.
- **Clear Activations (Reset Machine IDs):**
  If a customer changes hardware or gets locked out, open the license → **Edit** → **Clear All Activations**. This wipes machine bindings so the customer can activate again on a new device.

- **Bulk Actions (where applicable):**
  Use multi-select to export, block/unblock, or change status for multiple records at once.
- **Import/Export (CSV):**

  - **Export** all licenses or the current view with one click.

  - **Import** (see section **7.11**, BETA) if migrating from a prior system.

Best practices

- Keep **notes** on manual licenses so operators know the context later.
- Use **Aliases** (see 7.3) so product IDs show with friendly names, reducing mistakes.

Why it matters
This tab is where you'll spend most of your time: issuing, modifying, or troubleshooting licenses and activations.

7.2 Metrics Tab

Get a quick, data-driven view of system health and recent activity.

Key panels

- **At-a-Glance Cards:**
  Totals for **active vs trial**, recent **conversions**, **cancellations**, and overall license counts.
- **Database Health Report:**
  Automated checks for anomalies (e.g., duplicate keys, malformed IDs, stale subscription states).
  Findings include **action buttons** (e.g., run reconciliation) to resolve issues.
- **Recent Creations:**
  A live-updating list of the most recent licenses created (useful for real-time ops checks).

Reconciliation tools

- **Fast Reconciliation (last 24 hours):**
  Click **Fast Reconciliation** to sweep for any Stripe orders and changes that occurred in the last day and apply them to your database.
  Use when you suspect minor drift—e.g., a brief webhook outage yesterday.

- **Full Manual Reconciliation:**
  Runs a comprehensive pass to pull current truth from Stripe (and/or your website store, if integrated) and reconcile it into Key Commander.
  Summary & Notes:

    - Queries **provider-backed** purchases (Stripe/website).
    - **Does not** recreate or alter **manual** licenses you created in the GUI.
    - If you use **Restore from Backup** (see 7.7), a **Full Manual Reconciliation** is recommended right after restore to re-import missed purchases/events.

Why it matters
Reconciliation ensures your database matches payment reality. Use **Fast** for quick catch-ups; use **Full Manual** after restores, major outages, or suspected long-window drift.

7.3 Aliases & Messaging Tab

Identifier Aliases
Create human-friendly names for Stripe Product/Price IDs (e.g., map `prod_ABC123` → **“Pro Monthly”**).

- Click **Add Alias** → choose product/price → enter friendly name → **Save**.
- The new name appears throughout the GUI (and in exports), making operations clearer.

Application Messaging
Send announcements directly to your client software.

- Click **Add Message** → write your content (plain text or simple HTML), add links/images, and define audience/visibility.
- Toggle **Enabled** to push live or **Disabled** to hold for later.
- Great for release notes, downtime notices, promotions, or onboarding tips.

Why it matters
Aliases reduce operator mistakes; in-app messaging lets you talk to users where they actually work—inside your app.

7.4 Audit Log Tab

Your “black box recorder” for administrative and system actions.

What you’ll see

- **Admin Changes:** License creation, edits, deletions, activation clears.
- **Security & System Events:** Unlocks, sign-ins, webhook processing events.
- **Filtering & Search:** Find events by type, actor (which admin), target (license/user), or time window.

Why it matters
Audit logs support accountability, compliance, and fast incident investigation.

7.5 Error Log Tab

Raw, detailed server error context for advanced troubleshooting.

Capabilities

- **Stack traces & context** for errors encountered by the server or incoming clients.
- **Search & filters** to locate errors by endpoint, license, or time.
- **Operator reporting:** If you encounter a severe condition you want to flag, you can report it here so it’s captured and centralized.

When to use

- Payment/webhook issues, email delivery failures, unexpected validation errors, or anything that needs deep dive beyond the Metrics tab.

7.6 Reports Tab

Centralizes operational signals so you immediately know if your software or website is encountering problems.

What's included

- **Website Error Summaries:** Aggregations of error events reported from your site (if wired to Key Commander's intake).
- **Client App Error Summaries:** High-level counts and recent samples from deployed client apps.
- **Quick Links to Detail:** Jump into **Error Log** for full context or **Audit Log** when an error triggers an administrative action.

How to use

- Keep this tab open during launches or promotions to spot spikes.
- Treat rising error counts as a cue to investigate endpoints, Stripe webhooks, or email settings.

Why it matters
You get **one place** to see if something is breaking—across app and website—without digging through separate tools first.

7.7 License Types (Overview)

Understanding status values helps you act quickly and consistently.

- **Active:** Fully paid and entitled; normal operation.
- **Trial:** Time-limited access for evaluation; may have feature gating.
- **Inactive:** Disabled (e.g., canceled, expired). Cannot activate until reinstated.
- **Blocked:** Manually or automatically blocked due to policy violations (e.g., trial abuse, chargebacks).

Tip
Document your team's criteria for moving between these states so all operators handle edge cases the same way.

7.8 Machine ID (Anti-Piracy & Activation Management)

Key Commander includes machine binding to deter sharing of license keys.

How it works (operationally)

- Each license activation ties to a **Machine ID** provided by your client software.
- If a legitimate user changes hardware, use **Clear All Activations** to release the binding so they can activate again.

Self-service reset (optional)

- You may enable a **public endpoint** that allows a customer to **reset their own Machine ID** on a cadence (e.g., once every 30 days).
- This reduces support tickets while still protecting against abuse.

Operator resets

- Unlimited resets can be performed by an admin: open the license → **Edit** → **Clear All Activations** → confirm.
- You will see Machine IDs removed for that license; the next activation will bind to the customer's current hardware.

Why it matters
Machine binding curbs casual piracy while still giving you tools to quickly help legitimate users when they switch devices.

7.9 Anti-Trial Hopping

To mitigate repeated misuse of trials:

- The system monitors **Machine IDs** used for **Trial** licenses.
- If the same machine is repeatedly requesting trials for the same product, that behavior is flagged and the license can be **Blocked** automatically or by an operator.

Your part
Ensure your client software provides a **stable Machine ID** signal; anti-abuse policies rely on it.

7.10 Export Database as CSV

- From the **Licenses** tab, click **Export CSV**.
- Choose **All licenses** or **Current view** (if filters/search are active).
- Save the file to your chosen location for analysis, migration, or reporting.

Tip
Use exports to reconcile with external BI tools or to snapshot state before large changes.

7.11 Import Database as CSV (BETA)

(For migrations from a prior system.)

- Prepare a CSV that matches the portal's **import template** (column names and data types must align).
- Click **Import CSV**, select your file, and review the preview.
- Confirm to import. The system validates fields and will report any rows it can't accept.

Important notes

- This is a **BETA** feature—validate on a staging system first.
- Import **does not** automatically create Stripe subscriptions; it only seeds Key Commander's license records.
- After importing historic licenses, you may still need to perform a **Full Manual Reconciliation** to align with live payment state.

Quick Reference: When to Use What

- **Licenses Tab:** Day-to-day issuing, editing, clearing activations.
- **Metrics → Health Report:** Routine sanity checks and drift detection.
- **Metrics → Fast Reconciliation:** Yesterday's missed events or small drift.
- **Metrics → Full Manual Reconciliation:** After restore, long drift, or major outages.
- **Aliases & Messaging:** Clean operator UI + direct-to-user announcements.
- **Audit Log:** Who changed what and when.
- **Error Log:** Deep technical diagnosis.
- **Reports:** High-level "are things breaking right now?" dashboard.

8. Website Integration (Your SaaS Backend)

This section is your comprehensive guide to taking the provided SvelteKit website template and forging it into your own branded, production-ready storefront. We will walk through every critical step, from initial rebranding to final deployment, ensuring your website integrates seamlessly and securely with your self-hosted Key Commander backend.

8.1 Architectural Overview & Philosophy

What it is

The website in this repository is a complete, standalone SvelteKit application. It is not merely a theme; it is a fully functional frontend engineered to serve as the public face of your SaaS or LaaS business. Its core responsibilities are:

- **Marketing & Sales**: Presenting your products to the world through dedicated marketing pages.
- **User Authentication**: Handling the entire user lifecycle, including sign-up, sign-in, and password recovery, via Supabase.
- **Checkout & Billing**: Initiating secure payment sessions directly with Stripe.
- **Account Management**: Providing customers with a private dashboard to manage their profile and billing details.
- **Admin Dashboard**: Functional admin dashboard, with MMR data, customer billing portal, customer messaging, beta portal authorization, and a convenient machine ID reset point (for early resets)

Why it matters

This architecture deliberately separates your public-facing website from your secure backend operations. Your Key Commander instance is the hardened vault; this website is the public-facing storefront.

- **Security**: Customer data, license keys, and payment logic are handled by Key Commander on your private server. The website never holds this sensitive information; it only makes authenticated requests to Key Commander for entitlement data.

- **Performance**: By offloading heavy logic to Key Commander, the website remains lightweight and fast. It can be deployed on a global edge network (like Vercel's) for near-instant load times for your customers worldwide.
- **Control**: You command both ends of the operation—the public message and the private backend—without being locked into a monolithic platform. This is the essence of "Your Stack, Your Rules."

8.2 Rebranding & Customization Checklist

This checklist details the exact files and variables you must change to transform the Signal Lynx template into your own branded application.

1. Core Branding & Configuration

- **What it is**: A central configuration file that holds all primary branding and contact information.
- **What to do**: Open the file `src/config.ts`. This is your primary control panel. Modify the following constants:
    - `WebsiteName`: Your product or company name.
    - `WebsiteBaseUrl`: Your final production domain (e.g., `https://www.yourproduct.com`).
    - `WebsiteDescription`: The default meta description for search engines.
    - Inside the `SITE_CONFIG` object:
        - `logoPath`: Change to the path of your logo file within the `/static/images/` directory.
        - `companyLegalName`, `companyPhone`, `companyAddress`: Update with your official business details for legal pages.
        - Update all email addresses (`contactEmail`, `supportEmail`, etc.) to your own.
        - `socials`: Update the links to your social media profiles.
        - `footerNav`: Adjust the links that appear in the website footer.

- **Why it matters**: This single file populates your brand name, logo, and contact details across the entire site, including SEO tags, legal documents, and automated emails, ensuring consistency.

2. Color Theme & Styling

- **What it is**: A centralized theme file that controls the website's entire color palette.
- **What to do**: Open the file `src/lib/theme.ts`. This file exports a `saasstartertheme` object that defines colors for DaisyUI.
    - Modify the hex codes for `primary`, `secondary`, `accent`, and the `base-100` through `base-300` colors to match your brand's aesthetic.

- **Why it matters**: The entire website is built to be theme-aware. By editing only this file, every button, link, background, and component on the site will automatically adopt your new color scheme. You do not need to edit colors in any other CSS file.

3. Product Definitions

- **What it is**: The static, fallback source of truth for your product offerings.
- **What to do**: Open the file `src/lib/data/products.ts`.
    1. **Delete the Examples**: Remove the existing product objects for "Signal Shield," "Lynx-Relay," and "Key Commander" from the `allProducts` array.
    2. **Add Your Products**: Create new objects in the `allProducts` array for each of your products or subscription tiers. Pay close attention to these fields:
        - `id`: A unique, lowercase string for your product (e.g., `basic-plan`).
        - `name`, `title`, `tagline`, `features`: The marketing copy for the product cards.
        - `price`: The display price (e.g., "$10 / month").
        - `stripe_price_id`: This is critical. Paste the **Price ID** (e.g., `price_...`) from your Stripe Dashboard for this product. This is how the website tells Stripe which product the customer wants to buy.
    3.
    4. **Update Exported Views**: At the bottom of the file, modify the `automationProducts` and `licenseHubProduct` exports to filter for your new products as needed for your marketing pages.
    5. **Set the Default Plan**: Change the `defaultPlanId` to the `id` of one of your products.
-
- **Why it matters**: This file populates the pricing tables and product cards across the site. Getting the `stripe_price_id` correct is essential for the checkout process to function.

4. Content & Marketing Pages

- **What it is**: The Svelte files that contain the actual marketing copy and text for your site.
- **What to do**:
    - **Homepage**: Edit `src/routes/(marketing)/+page.svelte` to change the main hero section, trust bar, and other content.
    - **Product Pages**: The template includes example pages at `src/routes/(marketing)/key-commander/` and `src/routes/(marketing)/trading-automation/`. You should use these as a blueprint to build pages for your own products.
    - **FAQ**: Edit the arrays in `src/lib/data/faqData.ts` to update the questions and answers.
    - **Legal Pages**: Edit the Svelte files inside `src/routes/(marketing)/legal/` to update your Terms of Service, Privacy Policy, etc.

- **Why it matters**: This is where you inject your brand's voice and value proposition.

5. Static Assets & Icons

- **What it is**: The directory for globally accessible files like images, logos, and configuration files.
- **What to do**: Manually replace the files inside the `/static` directory with your own:
    - `/static/images/`: Replace the logos and marketing images.
    - `/static/favicon.png`: Your site's browser icon.
    - `/static/robots.txt`: Update this file if you have different SEO crawling requirements.

- **Why it matters**: These files define the visual identity of your site and its instructions for web crawlers and security researchers.

8.3 Integrating with Your Key Commander Backend

What it is

The process of securely linking your new website to your self-hosted Key Commander instance. This connection is vital for checking a logged-in user's entitlements (i.e., what they have paid for).

What to do

This integration is handled through environment variables, which are secret keys that are not stored in your code.

1. **Create Your Environment File**: If you haven't already, copy the example file to create your local configuration:
2. codeBash
3. download
4. content_copy
5. expand_less
6. cp .env.example .env.local
7. **Configure the Variables**: Open the `.env.local` file and find the following lines. You will also need to set these same variables in your production hosting environment (e.g., Vercel).
    - `PRIVATE_LICENSE_MANAGER_URL`: Set this to the public HTTPS URL of your Key Commander server. This is the domain you configured with the Cloudflare Tunnel (e.g., `https://api.yourdomain.com`).
    - `PRIVATE_LICENSE_MANAGER_API_KEY`: Set this to the **Website API Key** you generated in the Key Commander Admin Portal. You can find or create this key in the **Config** tab of the Admin Portal.

Why it matters

- **Security**: These API calls are made from your website's *server* to your Key Commander *server*. The API key is never exposed to the user's browser. This server-to-server pattern is secure and prevents unauthorized access to your licensing backend.
- **Functionality**: Without this connection, your website cannot verify what products a user has purchased. This will prevent logged-in users from accessing their downloads or seeing their active subscriptions in their account dashboard.

8.4 Deployment

Our Recommendation: Vercel for the Frontend

We strongly recommend deploying your SvelteKit website to **Vercel**. Their platform is optimized for modern web frameworks, offers a global edge network for incredible speed, and provides a generous free "Hobby" plan that is more than sufficient for most SaaS businesses to get started.

Critical Warning: Cost-Saving Strategy for File Hosting

What it is

A crucial distinction between hosting your *website* and hosting your *downloadable product files* (e.g., `.exe`, `.zip`, `.dmg`).

Why it matters

Do not use Vercel to host your software downloads. Vercel is designed to serve websites, not large binary files. Their free and even Pro plans have fair-use policies and hard limits on bandwidth and storage that are not suited for software distribution. Attempting to serve your product files from Vercel will likely result in slow downloads for your customers and can lead to your account being flagged or incurring significant costs.

The Recommended Solution: A Hybrid Approach

1. **Host Your Website on Vercel**: For a fast, reliable, and globally-distributed storefront.
2. **Host Your Files Elsewhere**:
    - **(Preferred) Cloud Object Storage**: Use a service like **AWS R2**. R2 is an ideal choice because it has a substantial free tier and, most importantly, **zero egress (bandwidth) fees**. This means you are not charged when your customers download your files, making it incredibly cost-effective.
    - **(Alternative) Self-Hosted**: You can also serve your files from the same Linux VPS that runs your Key Commander instance, using a simple web server like Nginx. This gives you full control but requires more configuration.

Deployment Walkthrough

1. **Deploy the Website to Vercel**:
    - Create a GitHub repository for your customized website code.
    - Sign up for a Vercel account and create a "New Project."
    - Connect Vercel to your GitHub account and import the repository. Vercel will automatically detect it as a SvelteKit project.

    - In the Vercel project's "Settings" -> "Environment Variables" section, add all the variables from your `.env.local` file (your Supabase keys, Stripe keys, and `PRIVATE_LICENSE_MANAGER_URL` / `PRIVATE_LICENSE_MANAGER_API_KEY`).
    - Click "Deploy." Vercel will build and launch your site.
2. **Host Your Product Files**:
    - If using R2, create a bucket, upload your product files, and make them publicly accessible. Note the public URL for your bucket.
3. **Final Configuration**:
    - Open `src/config.ts` one last time.
    - Find the `DOWNLOADS_BASE` variable.
    - Change its value to the URL of the folder where your product files are hosted (e.g., `https://your-r2-bucket-url.r2.dev/downloads`).
    - Commit and push this change to GitHub. Vercel will automatically redeploy your site with the updated download links.

Your customers will now browse your fast website on Vercel, and when they click to download a purchased product, they will be seamlessly directed to your efficient, low-cost file storage. This architecture provides the best of both worlds: world-class performance for your storefront and complete control over your costs and backend.

---

9. Troubleshooting

Use this section when something doesn't behave as expected. Each item includes **what it means** and **what to do**—from quickest checks to deeper fixes.

9.1 Backend Unreachable

What it means

The Admin Portal can't reach your server at the URL you entered.

Quick checks

- Verify the **server base URL** in **Config → Backend/Server Connection**.
- Confirm the **server is running** (container or service).
- If using a **Linux VPS**, ensure your **Cloudflare Tunnel** is **active** (public plane) and your **Tailscale** admin route is reachable (admin plane).
- Try the **admin plane URL** (e.g., `http://100.x.y.z:9000`) if the public URL is failing.

Deeper fixes

- **Port mapping:** The server normally listens on **8000**; ensure your tunnel/forwarder maps correctly (e.g., Cloudflared → 127.0.0.1:8000; Tailscale: 100.x.y.z:9000 → 8000).
- **Firewall:** Allow inbound on your admin port (Tailscale interface) and ensure the tunnel process can reach localhost:8000.
- **DNS:** If you're using a domain, confirm the record exists and points to your tunnel.
- **Restart the GUI** after changing the URL; stale tokens can confuse the first connection.

9.2 "Key Loading Error" / Incorrect Password

What it means

The **master password** entered is wrong, or the encrypted data on disk can't be opened.

What to do

1. Re-enter the **master password** carefully (check Caps Lock, keyboard layout).
2. If you've **forgotten** the password, there is **no recovery**:
    - Back up any exported data you still have.
    - **Wipe the server's data directory** ("scorched earth"), re-run initialization, and **restore from a backup** if available.
    - After restore, run a **Full Manual Reconciliation** to resync purchases and subscriptions.

Tip

Store the master password in a secure manager. Treat it like the root key to your environment.

9.3 Stuck in “Limited Mode” / Database Not Configured

Symptoms

- The Admin Portal shows a DSN prompt or “Limited Mode.”
- Server reports it can’t reach the database.

What to do

- Provide a valid **PostgreSQL DSN** in the GUI:
  `postgresql+asyncpg://<user>:<password>@<host>:<port>/<database>`

  Note: DSN must be all lowercase

1. **Restart the server** after saving the DSN to enter **Normal Mode**.
2. Check database basics:
   - DB is running and **reachable** on the host/port you entered.
   - Username/password are correct (URL-encode special characters).
   - Database exists and is accessible by your user.
   - For Docker compose, use the **service name** (e.g., `db`) as the host.

Notes

- PostgreSQL prefers **lowercase** database/user names.
- If you moved hosts, update the DSN accordingly.

9.4 Stripe Webhooks Not Working

Symptoms

- New purchases aren’t appearing.

- Health report shows **stale** or **out-of-sync** items.

What to do

1. **In Stripe Dashboard:**
- Ensure an active **Webhook Endpoint** points to your server, e.g.:
  `https://YOUR_DOMAIN/api/v1/webhook/stripe`
    - Confirm **Signing Secret** starts with `whsec_…`
    - Use **Send Test Event** to verify delivery.
2. **In Admin Portal (Providers):**
    - Check **Stripe Secret Key** (`sk_…`) and **Webhook Signing Secret** (`whsec_…`) are saved correctly.
3. **At the edge (Cloudflare/Tunnel):**
    - Allow the **webhook path** through your tunnel/WAF.
    - Disable aggressive bot/JS challenges for this path.
4. Run **Fast Reconciliation** (last 24h). If still missing, run **Full Manual Reconciliation**.

Tip
Rotated the signing secret in Stripe? Update it in the Admin Portal immediately.

9.5 Email Sending Fails (SMTP/Brevo/SES)

Symptoms

- Test email doesn't arrive.
- Automated emails aren't going out.

What to do

1. In **Email Settings**:
    - Confirm provider choice (SMTP, Brevo, or SES).
    - Enter correct host/port (SMTP), API key (Brevo), or access keys + region (SES).
    - Check **"Enable Email Notifications"** is turned on.
2. Send a **Test Email** again.
3. If still failing:
    - Verify **From** domain is verified (SES/Brevo).

- Check SPF/DKIM/DMARC at your DNS provider.
- Ensure outbound ports (e.g., 587 for SMTP) aren't blocked by your host.
- Check spam/junk folders.

9.6 Health Report Flags "Active License Past Renewal Date"

What it means

Your local data is out-of-sync with your payment source—often due to a missed webhook.

What to do

1. Run **Fast Reconciliation** (last 24h).
2. If not fixed, run **Full Manual Reconciliation**.
3. Double-check **Stripe secrets** and **webhook** configuration so future events arrive.

9.7 Offline License Tokens Not Working

Symptoms

- Client reports "offline validation failed."
- Tokens appear expired or unrecognized.

What to do

1. Ensure **Offline Validation** is enabled and **expiration hours** are set (e.g., 72).
2. Confirm a **Signing Passphrase** is saved; if you changed it recently, **previous tokens become invalid**.
3. Verify the client machine **time is correct** (NTP). Significant clock drift breaks token validity.
4. Regenerate the **offline token** and redeploy it to the client.

9.8 Machine ID / Activation Problems

Symptoms

- User changed hardware and can’t re-activate.
- Activation limit reached.

What to do

1. In **Licenses**, open the record → **Edit** → **Clear All Activations**.
   This wipes previous machine bindings so the user can activate again.
2. If you allow **self-service resets**, remind users the public reset endpoint typically allows **one reset every 30 days**.
3. Ensure your client app uses a **stable Machine ID** method—VMs or hardware changes can affect stability.

9.9 Backups & Restore Issues

Symptoms

- Backups not appearing at expected times.
- Restore fails or data looks old after restore.

What to do

1. **Automated Backups** run at **4:00 AM (server time)**—check the server’s timezone and next run.
2. Confirm the **backup path** exists and has **write** permissions (and disk space).
3. After **Restore**, immediately run a **Full Manual Reconciliation** to import post-backup purchases/events.

9.10 Windows Service Won’t Start (NSSM)

Symptoms

- Service starts and stops immediately.
- Server runs fine manually but not as a service.

What to do

1. In **NSSM**, confirm:
    - **Application Path** points to the server executable.
    - **Working Directory** is the server's folder (where configs live).
2. Set **Recovery** to auto-restart on failure.
3. Check **Windows Event Viewer** → Application logs for startup errors.
4. If the server depends on local Postgres/Redis, ensure those services start **before** Key Commander.

9.11 Performance or Timeouts

Symptoms

- Slow UI, slow API responses, or intermittent timeouts.

What to do

1. Check **CPU/RAM** on the host; consider more vCPU or memory.
2. Review **logs** for repeated errors or rate limits.
3. For high load, ensure the server is allowed to run **multiple workers** and that your DB pool sizing fits the host.
4. If email/webhook providers are throttling, reduce bursty operations or schedule them during off-peak.

9.12 Cloudflare / Tailscale Gotchas

Cloudflare (public plane)

- Tunnel must be **running** and the route **exists**.
- The API paths you use must be **allow-listed** in your tunnel/WAF rules.
- Avoid strict bot challenges on webhook paths or Admin Portal API calls.

Tailscale (admin plane)

- Confirm **device is connected** and you can ping its **100.x.y.z** address.
- Check **ACLs** permit your user to reach the server's admin port.
- If you changed the admin port mapping, update the GUI's backend URL.

9.13 Common HTTP Status Codes (What to Do)

- **401 Unauthorized:** Your GUI session token is missing/expired. Reconnect or unlock again.
- **403 Forbidden:** You're calling an admin/internal route without the right key or from an untrusted origin. Use the Admin Portal for admin routes and ensure origin trust is set when prompted.
- **404 Not Found:** URL path or base URL is wrong. Re-check the server URL and API path.
- **409 Conflict:** Action already in progress or duplicate. Wait and retry, or refresh and ensure you didn't double-click.
- **422 Unprocessable Entity:** A field is missing/invalid. Re-check inputs (emails, IDs, DSN, etc.).
- **429 Too Many Requests:** You hit a rate limit. Slow down or retry later.
- **500 Server Error:** Check Error Log for details; gather logs and retry once transient issues clear.

9.14 CSV Export / Import (BETA) Issues

Export fails

- Verify write permissions at your chosen path.
- Try exporting a **smaller filtered view**.

Import fails

- Use the **exact column names and types** from the import template.
- Fix any rows reported as invalid and re-import.
- Remember imports **do not** create Stripe subscriptions; run a **Full Manual Reconciliation** after migrating to align with live payments.

9.15 Website Entitlement Issues

Symptoms

- Your website can't fetch entitlements or license state.

What to do

1. Ensure you generated and configured the **Website API Key** in the Admin Portal and used it in your website backend.
2. Confirm the website calls the **internal** endpoints over the correct **base URL**.
3. If using Cloudflare, allow internal API paths through your tunnel/WAF.
4. Check timeouts and CORS (if applicable to your integration approach).

9.16 Time Synchronization Problems

Symptoms

- Tokens or webhook signatures fail intermittently.
- "Expired" errors even for newly issued data.

What to do

- Ensure **server time is accurate** (NTP).

- Ensure **client time** on customer machines is correct (especially for offline tokens).

9.17 Disk Space / Log Growth

Symptoms

- Backups stop, services crash, or database writes fail.

What to do

- Prune old backups (retention should do this, but verify).
- Rotate or prune **logs**.
- On Docker hosts, use `docker system prune` with care (avoid removing needed images/volumes).

9.18 Collecting Info for Support

Before asking for help, gather:

- **Admin Portal version** and **server build** details (from your deployment notes).
- **Exact error text** and **timestamp**.
- Relevant **Audit Log** and **Error Log** excerpts.
- Whether this is **public plane** or **admin plane**, and the URL used.
- Recent changes (DNS, tunnel, DSN, keys, email provider, upgrades).

Still stuck?
Work top-down: **URL → server health → secrets/keys → logs → network/tunnel → reconciliation**. Most issues resolve within those layers without code changes.

---

10. Integrating with the Signal Lynx License Manager API

This guide provides the essential information for developers to integrate their applications and websites with a self-hosted Signal Lynx License Manager (LM) instance. The LM server is treated as a service, and this document details the API endpoints your application will interact with.

All code examples are provided in Python using the `requests` library.

Integration Table Of Contents

10.1 Client Application Integration

This section covers endpoints designed to be called directly from your desktop or client-side application.

10.1.1. License Validation (Online & Offline)

This is the core feature of the License Manager. It's a two-step process designed for resilience:

1. **Online Check:** Your application makes an API call to your LM instance to validate the user's license key.
2. **Offline Token:** Upon successful validation, the LM issues a cryptographically signed token. Your application should cache this token. For a configurable period (e.g., 72 hours), your application can verify this token locally without needing an internet connection, allowing it to run offline.

Step 1: Performing an Online Validation Check

Your application should make a `POST` request to the `/api/v1/validate` endpoint.

- **Method:** `POST`
- **Endpoint:** `/api/v1/validate`
- **Request Body (JSON):**
    - `license_key` (string, required): The user's license key.
    - `machine_id` (string, required): A stable, unique identifier for the user's machine. See the "Client Machine ID Guidance" section for details.
    - `client_product_id` (string, required): The unique product identifier you've configured in your LM instance for this application.
    - `client_tier_level` (integer, required): The numeric tier level this application requires (e.g., `0` for basic, `1` for pro). The LM will check if the license's tier is greater than or equal to this value.
    - `client_version` (string, optional): The version of your client application (e.g., "1.2.3"). Providing this helps the License Manager target in-app messages to specific versions of your application.
- **Success Response (200 OK):**
  A JSON object with a structure based on `ValidationResponse`. The most important fields for your client are:
    - `overall_status` (string): The result of the validation (`"active"`, `"trial"`, `"developer"`, or `"invalid"`).
    - `token_payload_bundle_plaintext` (object): A JSON object containing the claims for the offline token. **Your application must cache this.**
    - `token_payload_bundle_signature` (string): The Base64-encoded signature for the plaintext bundle. **Your application must cache this.**
- **Error Responses:**
    - `400 Bad Request`: The license is invalid for a specific reason (e.g., expired, wrong product, activation limit reached). The `detail` field will contain a machine-readable error code.
    - `5xx Server Error`: A server-side error occurred.

Code Example: Online Validation

```
import requests
import json

def validate_license_online(lm_base_url: str, license_key: str, machine_id: str, product_id: str, tier_level: int, client_version: str = "1.0.0"):
```

```
    """
    Performs an online validation check against the License Manager.

    Returns:
        A tuple of (response_data, error_message).
        On success, response_data is the parsed JSON from the server.
        On failure, error_message contains a description of the issue.
    """
    endpoint = f"{lm_base_url.rstrip('/')}/api/v1/validate"
    payload = {
        "license_key": license_key,
        "machine_id": machine_id,
        "client_product_id": product_id,
        "client_tier_level": tier_level,
        "client_version": client_version,
    }

    try:
        response = requests.post(endpoint, json=payload, timeout=15)
        response.raise_for_status()
        return response.json(), None

    except requests.exceptions.HTTPError as e:
        error_detail = "Unknown validation error."
        try:
            error_detail = e.response.json().get("detail", e.response.text)
        except json.JSONDecodeError:
            error_detail = e.response.text
        return None, f"API Error {e.response.status_code}: {error_detail}"

    except requests.exceptions.RequestException as e:
        return None, f"Network Error: {e}"

# --- Usage ---
# lm_url = "http://your-license-manager.com"
# data, error = validate_license_online(lm_url, "LM-...", "CLIENT-MACHINE-ID-...", "prod_MyCoolApp", 0)
# if data:
#     print("Validation successful!")
#     cached_token = data.get("token_payload_bundle_plaintext")
#     cached_signature = data.get("token_payload_bundle_signature")
# else:
#     print(f"Validation failed: {error}")
```

Step 2: Caching and Verifying the Offline Token

After a successful online validation, your application must cache the `token_payload_bundle_plaintext` and `token_payload_bundle_signature`. When your application starts without an internet connection, it should validate this cached token locally.

IMPORTANT: To verify the token, your application needs the **Ed25519 Public Key** from your License Manager instance. You can find this key in your LM's GUI under the **Config -> Offline License Validation** section. Copy this key and embed it as a constant in your client application's source code.

Code Example: Offline Token Verification

```
import base64
import json
from datetime import datetime, timezone
from nacl.signing import VerifyKey
from nacl.exceptions import BadSignatureError

# IMPORTANT: PASTE YOUR PUBLIC KEY HERE
# Get this from your License Manager GUI -> Config -> Offline License Validation
LM_PUBLIC_SIGNING_KEY_B64 = "fO7dLD22gIS5vp2vLHy7mCFN_yz08yLr4XhxxlcUcbM=" # <-- REPLACE WITH YOUR KEY

def verify_license_offline(cached_token_payload: dict, cached_signature_b64: str, machine_id: str, product_id: str) -> Tuple[bool, str]:
    """
    Verifies a cached offline token. Returns a tuple of (is_valid, message).
    """
    if not cached_token_payload or not cached_signature_b64:
        return False, "No cached token available."

    try:
        # 1. Verify the cryptographic signature
        public_key = VerifyKey(base64.urlsafe_b64decode(LM_PUBLIC_SIGNING_KEY_B64))
        message_bytes = json.dumps(
            cached_token_payload, sort_keys=True, separators=(",", ":")
        ).encode("utf-8")
        signature_bytes = base64.urlsafe_b64decode(cached_signature_b64)
        public_key.verify(message_bytes, signature_bytes)

    except BadSignatureError:
        return False, "Invalid signature. The cached token is corrupt or has been tampered with."
    except Exception as e:
        return False, f"Token verification failed due to an error: {e}"

    # 2. Verify the token's claims
    now_ts = int(datetime.now(timezone.utc).timestamp())

    if now_ts >= cached_token_payload.get("exp", 0):
        return False, "Token has expired."
    if cached_token_payload.get("machine_id") != machine_id:
        return False, "Token is for a different machine."
```

```
    if cached_token_payload.get("product_id") != product_id:
        return False, "Token is for a different product."
    if cached_token_payload.get("status") not in ["active", "trial", "developer"]:
        return False, f"Token has an invalid status: {cached_token_payload.get('status')}"

    return True, f"Offline validation successful. Status: {cached_token_payload.get('status')}"

# --- Usage ---
# is_valid, message = verify_license_offline(cached_token, cached_signature, "CLIENT-MACHINE-ID-...",
"prod_MyCoolApp")
```

10.1.2. Client Machine ID Guidance

The `machine_id` is a crucial piece of data your client sends for validation. It is how the License Manager ties an activation to a specific computer.

- **Uniqueness:** The ID should be unique to the physical or virtual machine.
- **Stability:** The ID must remain the same across application restarts and OS updates. It should only change if there is a major hardware replacement (e.g., new motherboard).
- **Anonymity:** The ID should **not** contain any personally identifiable information (PII) like usernames, MAC addresses, or IP addresses.

Format: The License Manager expects the `machine_id` as an **uppercase alphanumeric string**, which can include hyphens (`-`).

> **Note on Server-Side Normalization:** The License Manager server will automatically normalize the machine ID you send (e.g., by converting it to uppercase). Providing a consistent, stable ID as recommended is crucial for reliable activation tracking.

Implementation Suggestions:
A robust strategy is to gather multiple stable hardware identifiers and create a cryptographic hash of them.

- **Windows:** `wmic csproduct get uuid` or the `MachineGuid` from the registry.
- **macOS:** Hardware UUID from `system_profiler SPHardwareDataType`.
- **Linux:** Machine ID from `/var/lib/dbus/machine-id` or `/etc/machine-id`.

10.1.3. In-App Messaging

The License Manager allows you to send messages directly to your client applications. These messages are delivered as part of the successful `ValidationResponse`.

The `app_messages` field in the response is an array of message objects. Each object contains:

- `message_title` (string): The title of the announcement.
- `message_text_html` (string): The body of the message (the server sanitizes it).
- `banner_image_url` (string, optional): A URL to an image to display.
- `cta_url` (string, optional): A "Call to Action" URL.

Code Example: Processing In-App Messages

```
# (Continuing from the online validation example)
# data, error = validate_license_online(...)
if data:
    app_messages = data.get("app_messages", [])
    if app_messages:
        print(f"--- You have {len(app_messages)} new message(s) ---")
        for message in app_messages:
            print(f"\nTitle: {message.get('message_title')}")
```

10.1.4. Error Reporting Pipeline

Your client application can report unhandled exceptions or critical errors back to your LM instance for analysis.

- **Method:** `POST`
- **Endpoint:** `/api/v1/report-error`
- **Request Body (JSON):**
    - `product_id` (string, required): Your application's product ID.
    - `client_version` (string, optional): The version of your client application.
    - `machine_id` (string, optional): The client's machine ID.
    - `license_key` (string, optional): The license key in use, if any.
    - `error_summary` (string, required): A single-line summary of the error.
    - `error_details` (string, optional): The full stack trace or other diagnostic information.
- **Success Response:** `202 Accepted`. This indicates the report was received for processing.

Code Example: Sending an Error Report

```
import requests
```

```
import traceback

def report_error_to_server(lm_base_url: str, product_id: str, version: str, license_key: str, machine_id: str, exception: Exception):
    endpoint = f"{lm_base_url.rstrip('/')}/api/v1/report-error"
    payload = {
        "product_id": product_id, "client_version": version,
        "license_key": license_key, "machine_id": machine_id,
        "error_summary": f"{type(exception).__name__}: {str(exception)}",
        "error_details": traceback.format_exc(),
    }
    try:
        requests.post(endpoint, json=payload, timeout=10)
        print("Error report sent successfully.")
    except requests.exceptions.RequestException as e:
        print(f"Failed to send error report: {e}")
```

10.2 Integrating Your Website with the License Manager

This guide details the server-to-server API endpoints your website's backend will use to interact with your self-hosted Signal Lynx License Manager (LM) instance.

10.2.1.Security: Server-to-Server Communication

All communication must be performed from your website's backend (e.g., a Node.js server, Python/Django, etc.). **Never expose the Internal API Key to the user's browser.**

Authentication is handled via a secret API key sent in the `X-Internal-API-Key` header with every request. You can configure this key in your LM instance's GUI under **Config -> Website Connection Key**.

Core Integration Workflow

The primary integration points for a website are:

1. **Claim Licenses on User Login:** Associate a user's account on your website with any licenses they may have purchased.
2. **Fetch User Entitlements:** Get a list of a user's owned products to display in their account dashboard (e.g., for showing download links).
3. **User Self-Service Machine ID Reset:** Allow authenticated users on your website to reset their own machine activations.

10.2.2. Claim Licenses on User Login

Purpose: This crucial step links a user from your website's authentication system (e.g., Supabase) to any licenses in the LM that match their email. This ensures that if a user purchases a product before creating an account on your site, their purchase is automatically linked when they sign up or log in.

This should be called from your backend every time a user successfully authenticates.

Method: `POST`
Endpoint: `/api/v1/internal/licenses/claim-by-email`

Request Body (JSON)

```
{
  "supabase_user_id": "a-uuid-string-from-your-user-system",
  "email": "user@example.com"
}
```

- `supabase_user_id` (string, required): The unique, stable identifier for the user from your website's authentication system.
- `email` (string, required): The user's verified email address.

Success Response (200 OK)

```
{
  "status": "success",
  "claimed_count": 1
}
```

- `claimed_count` (integer): The number of license records that were newly associated with the user's ID.
    - **Note:** A `claimed_count` of `0` is a successful response and simply means there were no un-claimed licenses matching the user's email at this time.

10.2.3. Fetch User Entitlements

Purpose: To get a complete list of all active products and licenses a user owns. Your website backend will call this endpoint to populate the user's "Account", "Dashboard", or "Downloads" page.

Method: `GET`
Endpoint: `/api/v1/internal/user-entitlements/{supabase_user_id}`

URL Parameters

- `supabase_user_id` (string, required): The user's unique ID from your website's auth system.
    - **Note:** This must be a valid UUID string. Sending a non-UUID value will result in a `422 Unprocessable Entity` error from the API.

Success Response (200 OK)

An array of `UserEntitlement` objects.`


```
[
  {
    "license_key": "LM-ABCD-EFGH-IJKL-MNOP-QRST",
    "status": "active",
    "product_identifier": "prod_MyCoolAppV1",
    "tier_identifier": "price_ProPlan",
    "product_display_name": "My Cool App",
    "tier_display_name": "Pro Plan",
    "renews_at": "2025-12-01T00:00:00Z",
    "trial_ends_at": null,
    "quantity": 1
  }
]
```


Each object in the array represents a single entitled product and contains all the necessary information to display to the user.

10.2.4. User Self-Service Machine ID Reset

Purpose: To allow a logged-in user on your website to reset their own machine activations, subject to the server's 30-day cooldown period.

Method: `POST`

Endpoint: `/api/v1/internal/users/public/reset-activations`

Request Body (JSON)


```
{
  "license_key": "LM-ABCD-EFGH-IJKL-MNOP-QRST",
  "email": "user@example.com"
}
```


- `license_key` (string, required): The specific license key the user wishes to reset.
- `email` (string, required): The user's email, for verification against the license record.

Success Response (200 OK)

```
{
  "status": "success",
  "message": "All machine activations for your license have been cleared.",
  "cleared_at_utc": "2024-10-27T10:00:00Z"
}
```

Error Responses

The `status` field will indicate the outcome. Common error statuses include:

- `rate_limited`: The user has already performed a reset within the last 30 days.
- `email_mismatch`: The provided email does not match the license record.
- `not_found`: The license key does not exist.

10.2.5. Full Code Example (Python Website Backend)

This example demonstrates a simple class that a Python-based web server (like Flask or Django) could use to perform all the necessary actions.

For a Svelte example, please see our free website template, available through the Signal Lynx Website (www.signallynx.com)

```python
import requests
import os
import logging
from typing import Dict, Any, Optional, List, Tuple

# Configure basic logging
logging.basicConfig(level=logging.INFO, format='%(asctime)s - %(levelname)s - %(message)s')

class WebsiteBackendIntegration:
    """
    A client for integrating a website backend with the Signal Lynx License Manager.
    """
    def __init__(self, lm_base_url: str, internal_api_key: str):
        if not lm_base_url or not internal_api_key:
            raise ValueError("License Manager URL and Internal API Key are required.")
        self.base_url = lm_base_url.rstrip('/')
        self.api_key = internal_api_key
        self.headers = {
            "X-Internal-API-Key": self.api_key,
            "Content-Type": "application/json",
            "Accept": "application/json"
        }
```

```python
    def _make_request(self, method: str, endpoint: str, **kwargs) -> Tuple[Optional[Dict | List],
Optional[str]]:
        """A robust wrapper for making API calls."""
        url = f"{self.base_url}{endpoint}"
        try:
            response = requests.request(method, url, headers=self.headers, timeout=15, **kwargs)
            response.raise_for_status()

            # Handle 204 No Content for DELETE
            if response.status_code == 204:
                return {"status": "success"}, None

            return response.json(), None
        except requests.exceptions.HTTPError as e:
            error_detail = "An unknown API error occurred."
            try:
                error_detail = e.response.json().get("detail", e.response.text)
            except ValueError:
                error_detail = e.response.text
            logging.error(f"API Error {e.response.status_code} from {url}: {error_detail}")
            return None, str(error_detail)
        except requests.exceptions.RequestException as e:
            logging.error(f"Network error connecting to {url}: {e}")
            return None, "Network error: Could not connect to the License Manager."

    def claim_licenses_on_login(self, user_id: str, user_email: str) -> Optional[int]:
        """
        Call this after a user successfully authenticates on your website.
        Returns the number of licenses claimed, or None on failure.
        """
        endpoint = "/api/v1/internal/licenses/claim-by-email"
        payload = {"supabase_user_id": user_id, "email": user_email}

        logging.info(f"Attempting to claim licenses for user_id: {user_id}")
        data, error = self._make_request("POST", endpoint, json=payload)

        if error:
            logging.error(f"Failed to claim licenses for user {user_id}: {error}")
            return None

        claimed_count = data.get("claimed_count", 0) if isinstance(data, dict) else 0
        if claimed_count > 0:
            logging.info(f"Successfully claimed {claimed_count} new license(s) for user {user_id}.")
        return claimed_count

    def get_user_entitlements(self, user_id: str) -> Optional[List[Dict[str, Any]]]:
        """
        Fetches all product entitlements for a given user ID.
```

```
        Returns a list of entitlement dictionaries, or None on failure.
        """
        endpoint = f"/api/v1/internal/user-entitlements/{user_id}"

        logging.info(f"Fetching entitlements for user_id: {user_id}")
        data, error = self._make_request("GET", endpoint)

        if error:
            logging.error(f"Failed to fetch entitlements for user {user_id}: {error}")
            return None

        if isinstance(data, list):
            return data
        return []

    def reset_machine_id(self, license_key: str, user_email: str) -> Optional[Dict[str, Any]]:
        """
        Initiates a machine ID reset for a user.
        Returns the full response dictionary from the server, or None on failure.
        """
        endpoint = "/api/v1/internal/users/public/reset-activations"
        payload = {"license_key": license_key, "email": user_email}

        logging.info(f"Attempting machine ID reset for license key: {license_key[:8]}...")
        data, error = self._make_request("POST", endpoint, json=payload)

        if error:
            logging.error(f"Failed to reset machine ID for key {license_key[:8]}: {error}")
            # Return a consistent error format
            return {"status": "error", "message": error}

        return data

# --- EXAMPLE USAGE ---

# In your website's backend, initialize this once
LM_URL = os.getenv("LICENSE_MANAGER_URL", "http://your-license-manager-url.com")
LM_API_KEY = os.getenv("LM_INTERNAL_API_KEY", "your-secret-api-key")

lm_client = WebsiteBackendIntegration(LM_URL, LM_API_KEY)

# --- Scenario 1: A user logs into your website ---
def handle_user_login(user_id: str, user_email: str):
    # This runs in the background and doesn't need to block the user.
    lm_client.claim_licenses_on_login(user_id, user_email)

    # Now, to render their account page, you fetch what they own.
    entitlements = lm_client.get_user_entitlements(user_id)
```

```
    if entitlements is not None:
        print(f"\n--- User {user_email} owns {len(entitlements)} product(s) ---")
        for ent in entitlements:
            print(f"- Product: {ent.get('product_display_name', 'N/A')}")
            print(f"  Key: {ent.get('license_key', 'N/A')}")
            print(f"  Status: {ent.get('status', 'N/A')}")
            if ent.get('renews_at'):
                print(f"  Renews At: {ent.get('renews_at')}")

# --- Scenario 2: User clicks "Reset Machine ID" for one of their licenses ---
def handle_machine_reset_request(license_key: str, user_email: str):
    result = lm_client.reset_machine_id(license_key, user_email)

    if result and result.get("status") == "success":
        print(f"\nSUCCESS: {result.get('message')}")
    elif result:
        print(f"\nFAILED: {result.get('message')}")
    else:
        print("\nFAILED: An unknown error occurred during the reset request.")


if __name__ == '__main__':
    # This is a simulation of how your web server would use the client.
    # Replace with your actual user data.
    test_user_id = "00000000-0000-0000-0000-000000000000" # A valid UUID from your user system
    test_user_email = "test@example.com"

    print("--- Simulating user login flow ---")
    handle_user_login(test_user_id, test_user_email)

    print("\n--- Simulating machine ID reset flow ---")
    # You would get this key from the entitlements fetched above.
    test_license_key = "LM-KEY-FROM-ENTITLEMENTS"
    handle_machine_reset_request(test_license_key, test_user_email)
```

---

11. Contact & Support

If you have worked through this guide and the Error Log but are still facing persistent issues, we are here to help. When reaching out, providing detailed information will allow us to resolve your issue much faster.

- **Contact Channels:**
    1. **Primary:** www.signallynx.com/contact_us

2. **Community:** Reach out on our official **Telegram** or **X/Twitter** pages.

- **Information to Provide:**
  1. Your **Signal Lynx License Key**.
  2. A **detailed description** of the problem, including when it started and what you were doing at the time.
  3. Any steps you have already taken to try and resolve it.
  4. **Screenshots** or copy of the Error Log
  5. Relevant copy-pasted excerpts from the **Error Log Tab** that correspond to the time the issue occurred. If you used the **"Send Error Report"** button, please let us know

---

12. Legal, Risks, and Disclaimers

This section provides a summary of key risks, disclaimers, and terms associated with the use of Key Commander. It is not a replacement for our full legal agreements.

By installing, accessing, or using the Signal Lynx software, you acknowledge that you have read, understood, and agree to be bound by our full, legally binding terms.

12.1. Governing Documents

Our complete legal framework can be found online at www.signallynx.com/legal. Your use of the software is governed by these documents in their entirety:

- **Terms of Service (TOS/EULA)**
- **Privacy Policy**
- **Billing & Refunds Policy**
- **DMCA Policy**

12.2. Key Disclaimers and Acknowledgment of Risk

The Key Commander software and all related services are provided "AS IS" and "AS AVAILABLE," without warranty of any kind, express or implied. We expressly disclaim all warranties, including but not limited to merchantability and fitness for a particular purpose. There is no guarantee of service, uptime, or uninterrupted availability.

Automated trading is a high-risk activity. While Key Commander is a robust tool, it cannot eliminate the inherent risks of trading cryptocurrencies. You acknowledge and agree that:

- Unforeseen technical issues can occur at any time. This includes, but is not limited to: internet outages, exchange API failures or downtime, power interruptions, operating system errors, software bugs, and **user configuration errors.**
- **Signal Lynx is not responsible for any financial outcomes.** We are not liable for performance of the Key Commander Software or Website template, or any damages arising from your use of or inability to use the software.

Signal Lynx is a technology company. We provide software tools for execution assistance only. **No content, documentation, or communication from Signal Lynx constitutes financial, legal, or tax advice.** You are solely responsible for the operation of your license manager and client software Key Commander is a self-hosted product, which means you are responsible for the security of the environment where it runs. This includes OS hardening, firewall configuration, anti-malware, access controls, and proper credential hygiene for your computer and exchange accounts.Signal Lynx security suggestions are not an implication of security, the risk belongs to the user.

Key Commander integrates with third-party services (e.g., Docker, Linux, TailScale, VPS providers, Windows, Stripe, OVHcloud, etc). We do not control and are not responsible for the availability, terms, policies, or performance of these third parties.