fin_intelli — Secure Auth (Cookie Refresh + CSRF) — Copy-Paste Pack

This pack upgrades your auth to **HttpOnly cookie refresh tokens** with **rotation + revocation**, **double-submit CSRF protection**, **bcrypt passwords** (with legacy SHA-256 fallback), security headers, and ready hooks for AI/risk analytics. It fits your current structure and PostgreSQL schema.

---

0) .env (new/updated keys)

```
# JWT
JWT_SECRET=change_this_super_secret_string
JWT_ALG=HS256
ACCESS_TOKEN_EXPIRE_MINUTES=15
REFRESH_TOKEN_EXPIRE_DAYS=14

# Cookies
COOKIE_SECURE=false            # true in production (HTTPS)
COOKIE_DOMAIN=127.0.0.1        # prod: yourdomain.tld (omit for localhost)
COOKIE_SAMESITE=Lax            # Lax or Strict (Strict blocks some flows)

# DB (you already have these)
DB_USER=postgres
DB_PASSWORD=postgres
DB_HOST=127.0.0.1
DB_PORT=5432
DB_NAME=fin_intelli
DB_DRIVER=postgresql+psycopg2   # recommended driver string
```

> On production: `COOKIE_SECURE=true`, `COOKIE_SAMESITE=Strict` (if your flows allow), and `COOKIE_DOMAIN` set to your domain.

---

1) `src/backend/utils/security.py` — bcrypt + legacy SHA-256 fallback

```python
# src/backend/utils/security.py
from __future__ import annotations

import hashlib
from passlib.context import CryptContext

pwd_context = CryptContext(
    schemes=["bcrypt"],
    deprecated="auto",
```

```
)

def _looks_sha256(hexstr: str) -> bool:
    return (
        isinstance(hexstr, str)
        and len(hexstr) == 64
        and all(c in "0123456789abcdef" for c in hexstr.lower())
    )

def hash_password(password: str) -> str:
    return pwd_context.hash(password)


def verify_password(plain: str, hashed: str) -> bool:
    if not hashed:
        return False
    try:
        if hashed.startswith("$2a$") or hashed.startswith("$2b$") or
hashed.startswith("$2y$"):
            return pwd_context.verify(plain, hashed)
    except Exception:
        pass
    if _looks_sha256(hashed):
        return hashlib.sha256(plain.encode()).hexdigest() == hashed
    try:
        return pwd_context.verify(plain, hashed)
    except Exception:
        return False


def needs_rehash(hashed: str) -> bool:
    if _looks_sha256(hashed):
        return True
    try:
        return pwd_context.needs_update(hashed)
    except Exception:
        return True
```

---

2) `src/backend/models/user.py` — delegate to security helpers

```
# src/backend/models/user.py
from sqlalchemy import Column, Integer, String, Boolean, Sequence
from src.backend.utils.database import Base
from src.backend.utils.security import hash_password as _hash,
verify_password as _verify

class User(Base):
```

```
    __tablename__ = "users"

    id = Column(Integer, Sequence("users_id_seq"), primary_key=True)
    username = Column(String(255), unique=True, nullable=False)
    email = Column(String(255), unique=True, nullable=False)
    hashed_password = Column(String(255), nullable=False)
    is_active = Column(Boolean, default=True)
    is_admin = Column(Boolean, default=False)

    @staticmethod
    def hash_password(password: str) -> str:
        return _hash(password)

    @staticmethod
    def verify_password(password: str, hashed: str) -> bool:
        return _verify(password, hashed)
```

(Optional) Add a relationship if you want ORM navigation:

```
# inside class User
# from sqlalchemy.orm import relationship
# refresh_tokens = relationship(
#     "RefreshToken", backref="user", cascade="all, delete-orphan",
passive_deletes=True
# )
```

---

3) `src/backend/utils/auth.py` — cookie refresh + rotation + CSRF

```
# src/backend/utils/auth.py
from __future__ import annotations

import os, hmac, hashlib, secrets
from datetime import datetime, timedelta
from typing import Optional, Tuple

import requests
from jose import jwt, JWTError
from fastapi import Request, Depends, HTTPException
from fastapi.responses import Response
from fastapi.security import OAuth2PasswordBearer
from sqlalchemy.orm import Session
from dotenv import load_dotenv

from src.backend.models.user import User
from src.backend.models.refresh_token import RefreshToken
```

```
from src.backend.models.user_activity_log import UserActivityLog
from src.backend.utils.database import get_db
from src.backend.utils.security import verify_password, hash_password as
hash_pwd, needs_rehash

load_dotenv()

# === JWT & cookie settings ===
JWT_SECRET = os.getenv("JWT_SECRET", "dev_change_me")
JWT_ALG = os.getenv("JWT_ALG", "HS256")
ACCESS_TOKEN_EXPIRE_MINUTES = int(os.getenv("ACCESS_TOKEN_EXPIRE_MINUTES",
15))
REFRESH_TOKEN_EXPIRE_DAYS = int(os.getenv("REFRESH_TOKEN_EXPIRE_DAYS", 14))

COOKIE_SECURE = os.getenv("COOKIE_SECURE", "false").lower() in
("1","true","yes")
COOKIE_DOMAIN = os.getenv("COOKIE_DOMAIN") or None
COOKIE_SAMESITE = os.getenv("COOKIE_SAMESITE", "Lax")  # "Lax" | "Strict" |
"None"

REFRESH_COOKIE_NAME = "refresh_token"
CSRF_COOKIE_NAME = "XSRF-TOKEN"

# OAuth2 docs helper (tokenUrl used by /docs only)
oauth2_scheme = OAuth2PasswordBearer(tokenUrl="/auth/login")

# === JWT access token ===

def create_access_token(sub: str, minutes: Optional[int] = None) -> str:
    exp = datetime.utcnow() + timedelta(minutes=minutes or
ACCESS_TOKEN_EXPIRE_MINUTES)
    payload = {"sub": sub, "exp": exp}
    return jwt.encode(payload, JWT_SECRET, algorithm=JWT_ALG)

# === CSRF helpers ===

def ensure_csrf_cookie(response: Response, request: Request):
    if request.cookies.get(CSRF_COOKIE_NAME):
        return
    token = secrets.token_urlsafe(32)
    response.set_cookie(
        CSRF_COOKIE_NAME,
        token,
        httponly=False,                # JS must read it to send header
        secure=COOKIE_SECURE,
        samesite=COOKIE_SAMESITE,
        domain=COOKIE_DOMAIN,
        path="/",
        max_age=7*24*3600,
    )
```

```
async def csrf_protect(request: Request):
    if request.method in ("POST", "PUT", "PATCH", "DELETE"):
        header = request.headers.get("X-CSRF-Token")
        cookie = request.cookies.get(CSRF_COOKIE_NAME)
        if not header or not cookie or header != cookie:
            raise HTTPException(status_code=403, detail="CSRF token missing
or invalid")

# === Refresh token storage (HMAC for DB) ===

def _hmac_hash(raw: str) -> str:
    return hmac.new(JWT_SECRET.encode(), raw.encode(),
hashlib.sha256).hexdigest()


def _set_refresh_cookie(response: Response, raw_token: str):
    response.set_cookie(
        REFRESH_COOKIE_NAME,
        raw_token,
        httponly=True,
        secure=COOKIE_SECURE,
        samesite=COOKIE_SAMESITE,
        domain=COOKIE_DOMAIN,
        path="/",
        max_age=REFRESH_TOKEN_EXPIRE_DAYS * 24 * 3600,
    )


def _clear_refresh_cookie(response: Response):
    response.delete_cookie(
        REFRESH_COOKIE_NAME,
        domain=COOKIE_DOMAIN,
        path="/",
    )


# === Geo + activity logging (best effort) ===

def _geolocate(ip: str) -> Tuple[str, str]:
    try:
        res = requests.get(f"https://ipapi.co/{ip}/json/", timeout=1.5)
        if res.status_code == 200:
            j = res.json()
            return (j.get("city") or "Unknown", j.get("country_name") or
"Unknown")
    except Exception:
        pass
    return ("Unknown", "Unknown")


def _log_activity(db: Session, user: User, request: Request, event: str,
```

```
success: bool = True, risk: float = 0.0, extra: dict | None = None):
    ip = request.client.host if request and request.client else "0.0.0.0"
    ua = request.headers.get("User-Agent", "Unknown")
    city, country = _geolocate(ip)
    log = UserActivityLog(
        user_id=user.id,
        event_type=event,
        ip_address=ip,
        device_info=ua[:255],
        user_agent=ua,
        geolocation_city=city,
        geolocation_country=country,
        login_success=success,
        risk_score=risk,
        extra_info=extra or {},
    )
    db.add(log)
    db.commit()

# === Core auth helpers ===

def authenticate_user(db: Session, username: str, password: str) ->
Optional[User]:
    username = (username or "").strip()
    if not username or not password:
        return None
    user = db.query(User).filter(User.username == username).first()
    if not user:
        return None
    if not verify_password(password, user.hashed_password):
        return None
    # Optional: upgrade hash on successful login
    if needs_rehash(user.hashed_password):
        user.hashed_password = hash_pwd(password)
        db.add(user)
        db.commit()
    return user


def _issue_refresh(db: Session, user: User, request: Request) -> str:
    raw = secrets.token_urlsafe(64)
    token_db = RefreshToken(
        user_id=user.id,
        token_hash=_hmac_hash(raw),
        device_info=(request.headers.get("User-Agent", "Unknown") if request
else "Unknown")[:255],
        ip_address=(request.client.host if (request and request.client) else
"0.0.0.0"),
        expires_at=datetime.utcnow() +
timedelta(days=REFRESH_TOKEN_EXPIRE_DAYS),
        is_revoked=False,
```

```
    )
    db.add(token_db)
    db.commit()
    return raw


def issue_tokens_and_set_cookie(db: Session, user: User, response: Response,
request: Request) -> dict:
    access = create_access_token(sub=str(user.id))
    raw_refresh = _issue_refresh(db, user, request)
    _set_refresh_cookie(response, raw_refresh)
    _log_activity(db, user, request, event="login", success=True)
    return {
        "access_token": access,
        "token_type": "bearer",
        "user": {
            "id": user.id,
            "username": user.username,
            "email": user.email,
            "is_active": user.is_active,
            "is_admin": user.is_admin,
        },
    }


def rotate_refresh_and_access(db: Session, response: Response, request:
Request) -> dict:
    raw_cookie = request.cookies.get(REFRESH_COOKIE_NAME)
    if not raw_cookie:
        raise HTTPException(status_code=401, detail="Missing refresh token")

    token = db.query(RefreshToken).filter(
        RefreshToken.token_hash == _hmac_hash(raw_cookie),
        RefreshToken.is_revoked.is_(False),
    ).first()
    if not token or token.expires_at < datetime.utcnow():
        raise HTTPException(status_code=401, detail="Invalid or expired
refresh token")

    user = db.query(User).filter(User.id == token.user_id).first()
    if not user:
        raise HTTPException(status_code=401, detail="User not found")

    # Revoke old, issue new
    token.is_revoked = True
    db.commit()

    new_raw = _issue_refresh(db, user, request)
    _set_refresh_cookie(response, new_raw)

    # New access token
```

```
    access = create_access_token(sub=str(user.id))
    return {"access_token": access, "token_type": "bearer"}


def revoke_refresh_cookie(db: Session, response: Response, request: Request)
-> dict:
    raw_cookie = request.cookies.get(REFRESH_COOKIE_NAME)
    if raw_cookie:
        token = db.query(RefreshToken).filter(RefreshToken.token_hash ==
_hmac_hash(raw_cookie)).first()
        if token:
            token.is_revoked = True
            db.commit()
    _clear_refresh_cookie(response)
    return {"detail": "Logged out"}


# === Token to user (for protected APIs) ===

def get_current_user(
    db: Session = Depends(get_db),
    token: str = Depends(oauth2_scheme)
) -> User:
    try:
        payload = jwt.decode(token, JWT_SECRET, algorithms=[JWT_ALG])
        user_id: str = payload.get("sub")
        if not user_id:
            raise HTTPException(status_code=401, detail="Invalid token")
    except JWTError:
        raise HTTPException(status_code=401, detail="Invalid token")

    user = db.query(User).filter(User.id == int(user_id)).first()
    if not user:
        raise HTTPException(status_code=401, detail="User not found")
    return user
```

---

4) `src/backend/routes/auth_api.py` — login/register/refresh/logout with CSRF + cookies

```
# src/backend/routes/auth_api.py
from fastapi import APIRouter, Form, Request, Depends, HTTPException
from fastapi.responses import JSONResponse
from sqlalchemy.orm import Session

from src.backend.utils.database import get_db
from src.backend.utils.auth import (
    authenticate_user,
    issue_tokens_and_set_cookie,
```

```
    rotate_refresh_and_access,
    revoke_refresh_cookie,
    csrf_protect,
)
from src.backend.models.user import User
from src.backend.utils.crud import create_user, get_user_by_username

auth_api = APIRouter()

# ------------------------------
# Register
# ------------------------------
@auth_api.post("/register", dependencies=[Depends(csrf_protect)])
async def register(
    request: Request,
    username: str = Form(...),
    email: str = Form(...),
    password: str = Form(...),
    db: Session = Depends(get_db),
):
    if get_user_by_username(db, username):
        raise HTTPException(status_code=400, detail="Username already
exists")

    hashed_pw = User.hash_password(password)
    user = create_user(db, username=username.strip(), email=email.strip(),
hashed_password=hashed_pw)

    # Return tokens + set refresh cookie
    resp = JSONResponse(issue_tokens_and_set_cookie(db, user,
JSONResponse({}), request))
    return resp

# ------------------------------
# Login (username/password)
# ------------------------------
@auth_api.post("/login", dependencies=[Depends(csrf_protect)])
async def login(
    request: Request,
    username: str = Form(...),
    password: str = Form(...),
    db: Session = Depends(get_db),
):
    user = authenticate_user(db, username, password)
    if not user:
        raise HTTPException(status_code=401, detail="Invalid credentials")

    data = issue_tokens_and_set_cookie(db, user, JSONResponse({}), request)
    return JSONResponse(data)

# ------------------------------
```

```
# Refresh (reads HttpOnly cookie, rotates)
# ------------------------------
@auth_api.post("/refresh", dependencies=[Depends(csrf_protect)])
async def refresh_token(request: Request, db: Session = Depends(get_db)):
    data = rotate_refresh_and_access(db, JSONResponse({}), request)
    return JSONResponse(data)

# ------------------------------
# Logout (revoke cookie + clear)
# ------------------------------
@auth_api.post("/logout", dependencies=[Depends(csrf_protect)])
async def logout(request: Request, db: Session = Depends(get_db)):
    data = revoke_refresh_cookie(db, JSONResponse({}), request)
    return JSONResponse(data)
```

> Note: We return JSON bodies **and** set/clear cookies. The refresh token **never** leaves HttpOnly storage.

---

5) `src/backend/routes/pages_router.py` — set CSRF cookie for GET pages

```
# src/backend/routes/pages_router.py
import os
from fastapi import APIRouter, Request
from fastapi.templating import Jinja2Templates
from fastapi.responses import Response
from src.backend.utils.auth import ensure_csrf_cookie

router = APIRouter()

current_dir = os.path.dirname(os.path.abspath(__file__))
project_root = os.path.abspath(os.path.join(current_dir, "../../.."))
templates_path = os.path.join(project_root, "frontend", "templates")

templates = Jinja2Templates(directory=templates_path)

@router.get("/")
async def landing_page(request: Request):
    resp = templates.TemplateResponse("home.html", {"request": request,
"title": "Home"})
    ensure_csrf_cookie(resp, request)
    return resp

@router.get("/login")
async def login_page(request: Request, error: str = None):
    resp = templates.TemplateResponse("login.html", {"request": request,
"title": "Login", "error": error})
    ensure_csrf_cookie(resp, request)
```

```
    return resp

@router.get("/register")
async def register_page(request: Request):
    resp = templates.TemplateResponse("register.html", {"request": request,
"title": "Register"})
    ensure_csrf_cookie(resp, request)
    return resp

@router.get("/dashboard")
async def dashboard_page(request: Request):
    resp = templates.TemplateResponse("dashboard.html", {"request": request,
"title": "Dashboard"})
    ensure_csrf_cookie(resp, request)
    return resp
```

---

6) Frontend JS — attach CSRF header; store only access token

`frontend/static/js/auth/token.js`

```
// frontend/static/js/auth/token.js
(function () {
  function setAccessToken(token) {
    if (token) sessionStorage.setItem("access_token", token);
  }
  function getAccessToken() {
    return sessionStorage.getItem("access_token");
  }
  function clearAccessToken() {
    sessionStorage.removeItem("access_token");
  }
  function getCookie(name) {
    const m = document.cookie.match(new RegExp("(?:^|; )" + name.replace(/([.
$?*|{}()\[\]\\/+^])/g, "\\$1") + "=([^;]*)"));
    return m ? decodeURIComponent(m[1]) : null;
  }
  async function authFetch(url, options = {}) {
    const headers = new Headers(options.headers || {});
    const at = getAccessToken();
    if (at) headers.set("Authorization", `Bearer ${at}`);
    if ((options.method || "GET").toUpperCase() !== "GET") {
      const csrf = getCookie("XSRF-TOKEN");
      if (csrf) headers.set("X-CSRF-Token", csrf);
    }
    const res = await fetch(url, { ...options, headers, credentials:
"include" });
    if (res.status === 401) {
      try {
```

```
        // Try cookie-based refresh
        const csrf = getCookie("XSRF-TOKEN");
        const res2 = await fetch("/auth/refresh", {
          method: "POST",
          headers: csrf ? { "X-CSRF-Token": csrf } : {},
          credentials: "include",
        });
        if (res2.ok) {
          const data = await res2.json();
          if (data.access_token) setAccessToken(data.access_token);
          const headers2 = new Headers(options.headers || {});
          const at2 = getAccessToken();
          if (at2) headers2.set("Authorization", `Bearer ${at2}`);
          if ((options.method || "GET").toUpperCase() !== "GET") {
            const csrf2 = getCookie("XSRF-TOKEN");
            if (csrf2) headers2.set("X-CSRF-Token", csrf2);
          }
          return fetch(url, { ...options, headers: headers2, credentials:
"include" });
        }
      } catch (_) {}
    }
    return res;
  }
  window.Auth = { setAccessToken, getAccessToken, clearAccessToken,
authFetch, getCookie };
})();
```

frontend/static/js/auth/login.js

```
// frontend/static/js/auth/login.js
document.getElementById("loginForm")?.addEventListener("submit", async (e)
=> {
  e.preventDefault();
  const data = new FormData(e.target); // username, password
  const csrf = Auth.getCookie("XSRF-TOKEN");

  try {
    const res = await fetch("/auth/login", {
      method: "POST",
      body: data,
      headers: csrf ? { "X-CSRF-Token": csrf } : {},
      credentials: "include",
    });
    const out = await res.json().catch(() => ({}));
    if (res.ok) {
      if (out.access_token) Auth.setAccessToken(out.access_token);
      window.location.href = "/dashboard";
    } else {
      document.getElementById("errorMsg").textContent = out.detail || "Login
```

```
failed";
    }
  } catch {
    document.getElementById("errorMsg").textContent = "Network error, please
try again";
  }
});
```

frontend/static/js/auth/register.js

```
// frontend/static/js/auth/register.js
document.getElementById("registerForm")?.addEventListener("submit", async
(e) => {
  e.preventDefault();
  const form = e.target;
  const data = new FormData(form); // username, email, password (+
confirmPassword optional)
  const csrf = Auth.getCookie("XSRF-TOKEN");
  const err = document.getElementById("errorMsg");
  err.textContent = "";

  const pwd = data.get("password");
  const confirm =
form.querySelector('[name="confirmPassword"]')?.value?.trim();
  if (confirm && pwd !== confirm) {
    err.textContent = "Passwords do not match"; return;
  }

  try {
    const res = await fetch("/auth/register", {
      method: "POST",
      body: data,
      headers: csrf ? { "X-CSRF-Token": csrf } : {},
      credentials: "include",
    });
    const out = await res.json().catch(() => ({}));
    if (res.ok) {
      if (out.access_token) Auth.setAccessToken(out.access_token);
      window.location.href = "/dashboard";
    } else {
      err.textContent = out.detail || "Registration failed";
    }
  } catch {
    err.textContent = "Network error, please try again";
  }
});
```

`frontend/static/js/auth/logout.js`

```
// frontend/static/js/auth/logout.js
(document.getElementById("logoutBtn") || {}).addEventListener?.("click",
async () => {
  try {
    const csrf = Auth.getCookie("XSRF-TOKEN");
    await fetch("/auth/logout", {
      method: "POST",
      headers: csrf ? { "X-CSRF-Token": csrf } : {},
      credentials: "include",
    });
  } catch (_) {}
  Auth.clearAccessToken();
  window.location.href = "/";
});
```

Include `token.js` **before** `login.js` / `register.js` in your templates.

---

7) Security headers middleware (app-wide)

```
# src/backend/middleware/security_headers.py
from fastapi import Request

async def security_headers_middleware(request: Request, call_next):
    resp = await call_next(request)
    resp.headers["Content-Security-Policy"] = (
        "default-src 'self'; script-src 'self'; style-src 'self' 'unsafe-
inline'; "
        "img-src 'self' data:'self'; font-src 'self'; object-src 'none';
base-uri 'none'; "
        "frame-ancestors 'none'; upgrade-insecure-requests"
    )
    resp.headers["X-Content-Type-Options"] = "nosniff"
    resp.headers["X-Frame-Options"] = "DENY"
    resp.headers["Referrer-Policy"] = "strict-origin-when-cross-origin"
    resp.headers["Permissions-Policy"] = "geolocation=(), microphone=(),
camera=()"
    return resp
```

Register it in `app.py`:

```
# src/backend/app.py (snippet)
from src.backend.middleware.security_headers import
security_headers_middleware
app.middleware("http")(security_headers_middleware)
```

Remove your previous middleware that tried to call `get_current_user` per request; use explicit dependency on protected APIs instead.

---

8) Optional: rate-limit login (Redis)

Sketch (choose one library like `slowapi` or `starlette-limiter`). Example with `starlette-limiter`:

```
# pip install starlette-limiter aioredis
# app.py
from starlette_limiter import Limiter, RateLimitMiddleware
from starlette_limiter.depends import RateLimiter

limiter = Limiter()
app.state.limiter = limiter
app.add_middleware(RateLimitMiddleware, authenticate=None)

# auth_api.py
@auth_api.post("/login", dependencies=[Depends(csrf_protect),
Depends(RateLimiter(times=5, seconds=60))])
async def login(...):
    ...
```

---

9) DB notes for your schema (PostgreSQL)

- Add helpful indexes:

```
CREATE INDEX IF NOT EXISTS idx_refresh_tokens_user_id ON
refresh_tokens(user_id);
CREATE INDEX IF NOT EXISTS idx_refresh_tokens_hash ON
refresh_tokens(token_hash);
CREATE INDEX IF NOT EXISTS idx_user_activity_logs_user_id ON
user_activity_logs(user_id);
```

- `token_hash` uses HMAC(JWT_SECRET, raw) so DB leaks can't directly replay.
- Foreign keys already match your schema; `ON DELETE CASCADE` is set for refresh tokens.

---

10) Threat model checklist (what this pack mitigates)

- **XSS impact reduced**: access token is short-lived (sessionStorage). Use CSP above; never `innerHTML` untrusted.
- **CSRF**: double-submit cookie + header required on state-changing routes.
- **Token replay/theft**: Refresh is in **HttpOnly** cookie; DB stores **HMAC hash**; rotation & revocation enforced.

- **Brute force**: add rate limiting (section 8) + log failures in `_log_activity` (extend as needed).
- **Clickjacking**: `X-Frame-Options: DENY`, `frame-ancestors 'none'`.
- **Session fixation**: rotate refresh on login/refresh; clear on logout.
- **Prying cookies**: `Secure` in prod; `SameSite=Lax/Strict`.

---

11) AI hooks / future scope

- `_log_activity` already records IP/UA/geo with `event` + `success`. Add features:
- Compute per-user anomaly scores (impossible travel, device drift).
- Feed events to a small detector (e.g., z-score) and notify via email/Telegram.
- Store embeddings of device strings & geo; flag outliers.
- Add a background job to periodically expire old refresh tokens.

---

12) End-to-end test flow

1. Restart app; open `/login` (DevTools → Network).
2. Verify a `XSRF-TOKEN` cookie exists.
3. Submit login; watch `POST /auth/login` → `Set-Cookie: refresh_token=...; HttpOnly` and JSON body with `access_token`.
4. Call a protected API with `Authorization: Bearer <access_token>`.
5. Force 401 (wait expiry or tamper), ensure client auto-calls `POST /auth/refresh` and retries.
6. Click **logout**; cookie cleared; further refresh returns 401.

Paste these files in place, set your `.env`, and you'll have a hardened, cookie-based refresh system with CSRF and rotation that fits your current project and schema. Ready to grow with AI-driven monitoring.