```
"""
Database module - SQLite for local, PostgreSQL for Railway
"""

import os
import json
import asyncio
import logging
from datetime import datetime
from typing import Optional, List, Dict

logger = logging.getLogger(__name__)


class Database:
    def __init__(self, db_url: str):
        self.db_url = db_url
        self.is_postgres = db_url.startswith("postgresql") or db_url.startswith("
        self._conn = None

    async def init(self):
        """Initialize database and create tables"""
        if self.is_postgres:
            import asyncpg
            self._conn = await asyncpg.connect(self.db_url)
            await self._create_tables_postgres()
        else:
            import aiosqlite
            db_path = self.db_url.replace("sqlite:///", "") or "bot.db"
            self._conn = await aiosqlite.connect(db_path)
            self._conn.row_factory = aiosqlite.Row
            await self._create_tables_sqlite()
        logger.info("Database initialized")

    async def _create_tables_sqlite(self):
        await self._conn.executescript("""
            CREATE TABLE IF NOT EXISTS users (
                user_id INTEGER PRIMARY KEY,
                username TEXT DEFAULT '',
                full_name TEXT DEFAULT '',
                role TEXT DEFAULT 'user',
                tokens INTEGER DEFAULT 5,
                unlimited_tokens INTEGER DEFAULT 0,
                reports_made INTEGER DEFAULT 0,
```

```
            banned INTEGER DEFAULT 0,
            created_at TEXT DEFAULT CURRENT_TIMESTAMP
        );

        CREATE TABLE IF NOT EXISTS accounts (
            id INTEGER PRIMARY KEY AUTOINCREMENT,
            user_id INTEGER,
            phone TEXT UNIQUE,
            session_string TEXT,
            active INTEGER DEFAULT 1,
            reports_done INTEGER DEFAULT 0,
            added_at TEXT DEFAULT CURRENT_TIMESTAMP,
            FOREIGN KEY (user_id) REFERENCES users(user_id)
        );

        CREATE TABLE IF NOT EXISTS reports (
            id INTEGER PRIMARY KEY AUTOINCREMENT,
            user_id INTEGER,
            target TEXT,
            report_type TEXT,
            custom_text TEXT DEFAULT '',
            count INTEGER DEFAULT 1,
            success INTEGER DEFAULT 0,
            failed INTEGER DEFAULT 0,
            created_at TEXT DEFAULT CURRENT_TIMESTAMP
        );

        CREATE TABLE IF NOT EXISTS token_packages (
            id INTEGER PRIMARY KEY AUTOINCREMENT,
            tokens INTEGER,
            price REAL,
            active INTEGER DEFAULT 1
        );

        CREATE TABLE IF NOT EXISTS settings (
            key TEXT PRIMARY KEY,
            value TEXT
        );
    """)
    # Seed default packages
    await self._conn.execute("""
        INSERT OR IGNORE INTO token_packages (id, tokens, price) VALUES
        (1, 10, 1.0), (2, 50, 4.0), (3, 100, 7.0), (4, 500, 30.0)
    """)
    await self._conn.commit()

async def _create_tables_postgres(self):
```

```
await self._conn.execute("""
    CREATE TABLE IF NOT EXISTS users (
        user_id BIGINT PRIMARY KEY,
        username TEXT DEFAULT '',
        full_name TEXT DEFAULT '',
        role TEXT DEFAULT 'user',
        tokens INTEGER DEFAULT 5,
        unlimited_tokens BOOLEAN DEFAULT FALSE,
        reports_made INTEGER DEFAULT 0,
        banned BOOLEAN DEFAULT FALSE,
        created_at TIMESTAMP DEFAULT NOW()
    );
    CREATE TABLE IF NOT EXISTS accounts (
        id SERIAL PRIMARY KEY,
        user_id BIGINT,
        phone TEXT UNIQUE,
        session_string TEXT,
        active BOOLEAN DEFAULT TRUE,
        reports_done INTEGER DEFAULT 0,
        added_at TIMESTAMP DEFAULT NOW()
    );
    CREATE TABLE IF NOT EXISTS reports (
        id SERIAL PRIMARY KEY,
        user_id BIGINT,
        target TEXT,
        report_type TEXT,
        custom_text TEXT DEFAULT '',
        count INTEGER DEFAULT 1,
        success INTEGER DEFAULT 0,
        failed INTEGER DEFAULT 0,
        created_at TIMESTAMP DEFAULT NOW()
    );
    CREATE TABLE IF NOT EXISTS token_packages (
        id SERIAL PRIMARY KEY,
        tokens INTEGER,
        price REAL,
        active BOOLEAN DEFAULT TRUE
    );
    CREATE TABLE IF NOT EXISTS settings (
        key TEXT PRIMARY KEY,
        value TEXT
    );
""")
await self._conn.execute("""
    INSERT INTO token_packages (tokens, price) VALUES
    (10, 1.0), (50, 4.0), (100, 7.0), (500, 30.0)
    ON CONFLICT DO NOTHING
```

```
    """)

# ─── Sync wrappers (for use from sync code in aiogram handlers) ───

def _run(self, coro):
    """Run a coroutine from sync context"""
    loop = asyncio.get_event_loop()
    if loop.is_running():
        import concurrent.futures
        with concurrent.futures.ThreadPoolExecutor() as pool:
            future = asyncio.run_coroutine_threadsafe(coro, loop)
            return future.result(timeout=30)
    return loop.run_until_complete(coro)

def register_user(self, user_id: int, username: str, full_name: str):
    return self._run(self._register_user(user_id, username, full_name))

async def _register_user(self, user_id: int, username: str, full_name: str):
    from config import Config
    config = Config()
    role = 'owner' if user_id == config.OWNER_ID else 'user'
    tokens = 999999 if user_id == config.OWNER_ID else 5

    if self.is_postgres:
        await self._conn.execute("""
            INSERT INTO users (user_id, username, full_name, role, tokens)
            VALUES ($1, $2, $3, $4, $5)
            ON CONFLICT (user_id) DO UPDATE SET username=$2, full_name=$3
        """, user_id, username, full_name, role, tokens)
    else:
        await self._conn.execute("""
            INSERT OR IGNORE INTO users (user_id, username, full_name, role,
            VALUES (?, ?, ?, ?, ?)
        """, (user_id, username, full_name, role, tokens))
        await self._conn.execute("""
            UPDATE users SET username=?, full_name=? WHERE user_id=?
        """, (username, full_name, user_id))
        await self._conn.commit()

def get_user(self, user_id: int) -> Optional[Dict]:
    return self._run(self._get_user(user_id))

async def _get_user(self, user_id: int) -> Optional[Dict]:
    if self.is_postgres:
        row = await self._conn.fetchrow("SELECT * FROM users WHERE user_id=$1
        return dict(row) if row else None
    else:
```

```
        async with self._conn.execute("SELECT * FROM users WHERE user_id=?",
            row = await cur.fetchone()
            return dict(row) if row else None

def get_all_users(self, limit=50) -> List[Dict]:
    return self._run(self._get_all_users(limit))

async def _get_all_users(self, limit):
    if self.is_postgres:
        rows = await self._conn.fetch("SELECT * FROM users ORDER BY created_a
        return [dict(r) for r in rows]
    else:
        async with self._conn.execute("SELECT * FROM users ORDER BY rowid DES
            rows = await cur.fetchall()
            return [dict(r) for r in rows]

def add_tokens(self, user_id: int, amount: int):
    return self._run(self._add_tokens(user_id, amount))

async def _add_tokens(self, user_id: int, amount: int):
    if self.is_postgres:
        await self._conn.execute("UPDATE users SET tokens=tokens+$1 WHERE use
    else:
        await self._conn.execute("UPDATE users SET tokens=tokens+? WHERE user
        await self._conn.commit()

def deduct_tokens(self, user_id: int, amount: int):
    return self._run(self._deduct_tokens(user_id, amount))

async def _deduct_tokens(self, user_id: int, amount: int):
    if self.is_postgres:
        await self._conn.execute("UPDATE users SET tokens=GREATEST(0,tokens-$
    else:
        await self._conn.execute("UPDATE users SET tokens=MAX(0,tokens-?) WHE
        await self._conn.commit()

def set_user_role(self, user_id: int, role: str):
    return self._run(self._set_user_role(user_id, role))

async def _set_user_role(self, user_id: int, role: str):
    if self.is_postgres:
        await self._conn.execute("UPDATE users SET role=$1 WHERE user_id=$2",
    else:
        await self._conn.execute("UPDATE users SET role=? WHERE user_id=?", (
        await self._conn.commit()

def ban_user(self, user_id: int, banned: bool):
```

```
    return self._run(self._ban_user(user_id, banned))

async def _ban_user(self, user_id: int, banned: bool):
    val = 1 if banned else 0
    if self.is_postgres:
        await self._conn.execute("UPDATE users SET banned=$1 WHERE user_id=$2
    else:
        await self._conn.execute("UPDATE users SET banned=? WHERE user_id=?",
        await self._conn.commit()

# ─── Accounts ───

def get_user_accounts(self, user_id: int) -> List[Dict]:
    return self._run(self._get_user_accounts(user_id))

async def _get_user_accounts(self, user_id: int):
    if self.is_postgres:
        rows = await self._conn.fetch("SELECT * FROM accounts WHERE user_id=$
        return [dict(r) for r in rows]
    else:
        async with self._conn.execute("SELECT * FROM accounts WHERE user_id=?
            return [dict(r) for r in await cur.fetchall()]

def get_all_active_accounts(self) -> List[Dict]:
    return self._run(self._get_all_active_accounts())

async def _get_all_active_accounts(self):
    if self.is_postgres:
        rows = await self._conn.fetch("SELECT * FROM accounts WHERE active=TR
        return [dict(r) for r in rows]
    else:
        async with self._conn.execute("SELECT * FROM accounts WHERE active=1"
            return [dict(r) for r in await cur.fetchall()]

def save_account(self, user_id: int, phone: str, session_string: str):
    return self._run(self._save_account(user_id, phone, session_string))

async def _save_account(self, user_id: int, phone: str, session_string: str):
    if self.is_postgres:
        await self._conn.execute("""
            INSERT INTO accounts (user_id, phone, session_string)
            VALUES ($1, $2, $3)
            ON CONFLICT (phone) DO UPDATE SET session_string=$3, active=TRUE
        """, user_id, phone, session_string)
    else:
        await self._conn.execute("""
            INSERT OR REPLACE INTO accounts (user_id, phone, session_string)
```

```
            VALUES (?, ?, ?)
        """, (user_id, phone, session_string))
        await self._conn.commit()

def remove_account(self, acc_id: int):
    return self._run(self._remove_account(acc_id))

async def _remove_account(self, acc_id: int):
    if self.is_postgres:
        await self._conn.execute("DELETE FROM accounts WHERE id=$1", acc_id)
    else:
        await self._conn.execute("DELETE FROM accounts WHERE id=?", (acc_id,)
        await self._conn.commit()

def get_account_by_id(self, acc_id: int) -> Optional[Dict]:
    return self._run(self._get_account_by_id(acc_id))

async def _get_account_by_id(self, acc_id: int):
    if self.is_postgres:
        row = await self._conn.fetchrow("SELECT * FROM accounts WHERE id=$1",
        return dict(row) if row else None
    else:
        async with self._conn.execute("SELECT * FROM accounts WHERE id=?", (a
            row = await cur.fetchone()
            return dict(row) if row else None

def set_account_active(self, acc_id: int, active: bool):
    return self._run(self._set_account_active(acc_id, active))

async def _set_account_active(self, acc_id: int, active: bool):
    val = 1 if active else 0
    if self.is_postgres:
        await self._conn.execute("UPDATE accounts SET active=$1 WHERE id=$2",
    else:
        await self._conn.execute("UPDATE accounts SET active=? WHERE id=?", (
        await self._conn.commit()

# ── Reports ──

def save_report(self, user_id: int, target: str, rtype: str, custom_text: str
    return self._run(self._save_report(user_id, target, rtype, custom_text, c

async def _save_report(self, *args):
    user_id, target, rtype, custom_text, count, success, failed = args
    if self.is_postgres:
        await self._conn.execute("""
            INSERT INTO reports (user_id, target, report_type, custom_text, c
```

```
            VALUES ($1,$2,$3,$4,$5,$6,$7)
        """, user_id, target, rtype, custom_text, count, success, failed)
        await self._conn.execute("UPDATE users SET reports_made=reports_made+
    else:
        await self._conn.execute("""
            INSERT INTO reports (user_id, target, report_type, custom_text, c
            VALUES (?,?,?,?,?,?,?)
        """, (user_id, target, rtype, custom_text, count, success, failed))
        await self._conn.execute("UPDATE users SET reports_made=reports_made+
        await self._conn.commit()

def get_user_reports(self, user_id: int, limit=10) -> List[Dict]:
    return self._run(self._get_user_reports(user_id, limit))

async def _get_user_reports(self, user_id: int, limit: int):
    if self.is_postgres:
        rows = await self._conn.fetch(
            "SELECT * FROM reports WHERE user_id=$1 ORDER BY created_at DESC
        )
        return [dict(r) for r in rows]
    else:
        async with self._conn.execute(
            "SELECT * FROM reports WHERE user_id=? ORDER BY rowid DESC LIMIT
        ) as cur:
            return [dict(r) for r in await cur.fetchall()]

# ── Stats ──

def get_stats(self) -> Dict:
    return self._run(self._get_stats())

async def _get_stats(self):
    stats = {}
    if self.is_postgres:
        stats['users'] = await self._conn.fetchval("SELECT COUNT(*) FROM user
        stats['accounts'] = await self._conn.fetchval("SELECT COUNT(*) FROM a
        stats['reports'] = await self._conn.fetchval("SELECT COALESCE(SUM(suc
    else:
        async with self._conn.execute("SELECT COUNT(*) FROM users") as c:
            stats['users'] = (await c.fetchone())[0]
        async with self._conn.execute("SELECT COUNT(*) FROM accounts") as c:
            stats['accounts'] = (await c.fetchone())[0]
        async with self._conn.execute("SELECT COALESCE(SUM(success),0) FROM r
            stats['reports'] = (await c.fetchone())[0]
    return stats

def get_detailed_stats(self) -> Dict:
```

```
    return self._run(self._get_detailed_stats())

async def _get_detailed_stats(self):
    stats = await self._get_stats()
    if self.is_postgres:
        stats['active_accounts'] = await self._conn.fetchval("SELECT COUNT(*)
        stats['successful_reports'] = await self._conn.fetchval("SELECT COALE
    else:
        async with self._conn.execute("SELECT COUNT(*) FROM accounts WHERE ac
            stats['active_accounts'] = (await c.fetchone())[0]
        async with self._conn.execute("SELECT COALESCE(SUM(success),0) FROM r
            stats['successful_reports'] = (await c.fetchone())[0]
    return stats

def save_account_report(self, acc_id: int, count: int = 1):
    return self._run(self._save_account_report(acc_id, count))

async def _save_account_report(self, acc_id: int, count: int):
    if self.is_postgres:
        await self._conn.execute("UPDATE accounts SET reports_done=reports_do
    else:
        await self._conn.execute("UPDATE accounts SET reports_done=reports_do
        await self._conn.commit()

# ─── Token packages ───

def get_token_packages(self) -> List[Dict]:
    return self._run(self._get_token_packages())

async def _get_token_packages(self):
    if self.is_postgres:
        rows = await self._conn.fetch("SELECT * FROM token_packages WHERE act
        return [dict(r) for r in rows]
    else:
        async with self._conn.execute("SELECT * FROM token_packages WHERE act
            return [dict(r) for r in await cur.fetchall()]

# ─── Settings ───

def get_setting(self, key: str) -> Optional[str]:
    return self._run(self._get_setting(key))

async def _get_setting(self, key: str):
    if self.is_postgres:
        val = await self._conn.fetchval("SELECT value FROM settings WHERE key
        return val
    else:
```

```
        async with self._conn.execute("SELECT value FROM settings WHERE key=?
            row = await cur.fetchone()
            return row[0] if row else None

def set_setting(self, key: str, value: str):
    return self._run(self._set_setting(key, value))

async def _set_setting(self, key: str, value: str):
    if self.is_postgres:
        await self._conn.execute("""
            INSERT INTO settings (key, value) VALUES ($1, $2)
            ON CONFLICT (key) DO UPDATE SET value=$2
        """, key, value)
    else:
        await self._conn.execute("""
            INSERT OR REPLACE INTO settings (key, value) VALUES (?, ?)
        """, (key, value))
        await self._conn.commit()
```