Role: You are an expert Python engineer. Generate production-ready code.

Goal: Build an async Binance API client class named `BinanceAPI` that exactly matches the following architecture and behavior for a crypto bot. Use Python 3.11 and aiohttp.

Project context:
- The bot runs multiple strategies (scalping/mean-reversion/arbitrage) and needs high-frequency price reads with a robust simulation mode. Latency and stability matter more than exotic features.

Code to produce:
- A single file: binance_api.py

Dependencies:
- aiohttp, asyncio, time, json, os, math, hmac, hashlib, urllib.parse.urlencode, structlog
- Import from local project: `from config import Config, ApiKeys` and `from utils import clean_symbol`
- Typing: List, Dict, Optional

Design requirements:
1) Class pattern
  - `class BinanceAPI` implemented as a **singleton**: subsequent instantiations reuse the same instance.
  - Lazy-initialized, shared `aiohttp.ClientSession` via `_get_session()`.
  - Base URL: **https://api.binance.us**
  - Logging with `structlog.get_logger("BinanceAPI")`, include a `module="BinanceAPI"` field.

2) Configuration flags and keys
  - `Config.USE_LIVE_MODE` (bool) toggles live vs sim.
  - `Config.SIMULATION` (bool) may be present; when true and not using live balance, return mock balances.
  - `Config.USE_LIVE_BALANCE_IN_SIM` (bool) optionally fetches real balances while in simulation.
  - `Config.MOCK_PRICES` is a dict of symbol->price for final fallback.
  - `ApiKeys.BINANCE_API_KEY`, `ApiKeys.BINANCE_SECRET_KEY`.

3) Symbol validation and caching
  - Maintain `self.valid_symbols` loaded by `_load_valid_symbols()`:
    - Use on-disk cache file `valid_symbols_cache.txt`.
    - In sim mode, always use the cache if present; in live, refresh if cache is older than 1 hour.
    - If refresh is needed, call Binance `/api/v3/exchangeInfo`, parse tradable spot symbols, and write back to cache (as a list).
  - `is_valid_symbol(symbol: str) -> bool` returns membership in `self.valid_symbols`.

4) HTTP and signing
  - `_make_request(method, endpoint, params=None, auth=False)`:
    - Enforce a minimal 1.0s gap between HTTP requests (`self._min_request_interval`).
    - Use the shared session.
    - If `auth=True`, add `timestamp` ms, sign with HMAC-SHA256 over the query string using `ApiKeys.BINANCE_SECRET_KEY`, append `signature` param, and set `X-MBX-APIKEY`.
    - Handle 429 (log warning, sleep 60s, return None), non-200 (log error with text), timeouts, and general exceptions with clear logs.
  - No global retries; leave that to callers/strategies.

5) Price retrieval (critical path)
  - `async def get_price(symbol: str) -> float`:
    - Normalize `symbol = clean_symbol(symbol)`.
    - Maintain a local per-symbol cache dict (`self._price_cache`) with sub-second TTL (~0.5s). If fresh, return cached.
    - **Priority chain**:
      1) Try WebSocket: `from ws_price_feed import ws_price_feed` then `ws_price_feed.get_cached_price(symbol)`. If found, cache and return it.
      2) REST `/api/v3/ticker/price?symbol=SYMBOL` → parse `price`.
      3) REST `/api/v3/ticker/24hr?symbol=SYMBOL` → parse `lastPrice`.
      4) Final fallback: `Config.MOCK_PRICES.get(symbol, 0.0)`.
    - If the price > 500000, cap to 500000 and warn.
    - If at the end no source yields > 0, return `0.0`.
    - **Important**: Reuse the shared session (do not open ad-hoc sessions here).

6) Market data
  - `async def get_klines(symbol: str, interval: str, limit: int = 100) -> List[List[float]]`:
    - Validate symbol.
    - If live: call `/api/v3/klines`, parse into `[openTime, open, high, low, close, volume]` floats/ints.
    - On error, return `[]`.
  - `async def get_order_book(symbol: str, limit: int = 10)`:
    - In sim: return a mocked order book centered around `get_price(symbol)` with a tiny spread and descending quantities.
    - In live: (optional) implement `/api/v3/depth`; it's acceptable to start with sim-only behavior.

7) Trading and validation
  - `async def get_filters(symbol: str) -> List[Dict]]`:
    - In live: read from `/api/v3/exchangeInfo`.
    - In sim: return realistic LOT_SIZE/MIN_NOTIONAL defaults for top symbols (BTCUSDT, ETHUSDT, BNBUSDT, SOLUSDT, XRPUSDT, DOGEUSDT, SHIBUSDT, etc.). Include minQty, maxQty, stepSize, minNotional.
  - `async def validate_quantity(symbol: str, quantity: float, price: float) -> float`:
    - Enforce LOT_SIZE (minQty, stepSize rounding, maxQty).
    - If invalid, return `0.0`.
  - `async def validate_notional(symbol: str, quantity: float, price: float) -> bool`:
    - Enforce MIN_NOTIONAL (qty*price).
  - `async def place_order(symbol: str, side: str, quantity: float, **kwargs) -> Dict`:
    - Validate symbol + quantity + notional.
    - In sim: log a filled order and return `{"orderId": f"sim_{symbol}_{timestamp}", "status": "FILLED"}` using `get_price(symbol)` as fill price.
    - In live: it's acceptable to stub out or return `{}` for now (no real orders).

8) Account
  - `async def get_account() -> Dict`:
    - If API keys are absent, return `{"balances": []}` and warn.
    - If sim AND NOT `USE_LIVE_BALANCE_IN_SIM`: return `{"balances": [{"asset":"USDT","free": "<sim-balance>","locked":"0"}]}` based on `Config.SIM_STARTING_BALANCE` (default 10000.0).
    - Otherwise, call signed `/api/v3/account`, return only non-zero balances with numeric floats.

9) Logging

- Log notable branches: cache hits/misses, WS hits, REST fallbacks, invalid symbol, validation failures, large price caps, rate limit sleeps, and account balancing.

10) Quality
- Add type hints and docstrings.
- No blocking calls; all I/O is async.
- Ensure deterministic returns: prices are floats (0.0 when invalid), lists are `[]` on failure, dicts for `get_account()`.

Deliverable:
- A single `binance_api.py` implementing everything above, production-ready, matching the described behavior so it can drop-in replace an existing handler with identical semantics.