

Trading Bot V2 — End-to-End Architecture Walkthrough

Date: 2026-02-06

1. High-level purpose of the application

What problem it solves

This is an automated intraday options trading bot for NSE/BSE indices that executes a rules-based strategy (SuperTrend, optionally MACD-confirmed) and tracks trades, P&L, and analytics.

It solves: “run a simple discretionary-style intraday options system automatically, with guardrails (max loss, max trades), plus a live dashboard.”

Who it is for

A single trader/operator (or small team) who wants to run and monitor an index options bot (NIFTY/BANKNIFTY/SENSEX/FINNIFTY) and later review performance.

What kind of system it is

- Backend: Python FastAPI service exposing REST endpoints + WebSocket stream and running an in-process trading engine loop.
- Frontend: React dashboard (SPA) consuming REST + WebSocket.
- Database: SQLite for trades/config/candle snapshots.

2. Architecture overview

Components/modules

Backend (Python):

- backend/server.py — FastAPI routes + WebSocket manager
- backend/bot_service.py — interface layer between API and engine
- backend/trading_bot.py — trading engine loop (entries/exits)
- backend/dhan_api.py — Dhan broker SDK wrapper
- backend/indicators.py — SuperTrend + MACD implementations
- backend/utils.py — market-hours/time helpers
- backend/indices.py — index metadata (lot sizes, strike rounding)
- backend/database.py — SQLite persistence + analytics
- backend/config.py — global runtime state + config dicts
- backend/models.py — request models (Pydantic)

Frontend (React):

- frontend/src/App.js — routing, global state, REST + WebSocket wiring
- frontend/src/pages/Dashboard.jsx — dashboard layout
- frontend/src/pages/Settings.jsx — credentials + risk params
- frontend/src/pages/TradesAnalysis.jsx — analytics UI
- frontend/src/components/* — panels/widgets

Infra:

- docker-compose.yml — runs backend (8001) + frontend (80)
- frontend/nginx.conf — proxies /api and /ws to backend in Docker

How they interact

- REST calls: React UI calls `/api/*` endpoints → FastAPI server delegates to `bot_service` and database.
- WebSocket: UI connects to `/ws` → trading loop broadcasts “state_update” snapshots via `server.ConnectionManager`.
- Persistence: bot writes trades, candle telemetry, and config into SQLite (`backend/data/trading.db`).

Data flow (from input to output)

- 1) Operator updates settings in UI → `POST /api/config/update` → `bot_service` updates config dict → `database.save_config` persists.
- 2) Operator clicks Start → `POST /api/bot/start` → `TradingBot.start` spawns `TradingBot.run_loop()` asyncio task.
- 3) Bot loop fetches spot/option prices via `DhanAPI` (live) or simulation (paper + bypass).
- 4) Bot aggregates 1-second ticks into candles; on candle close updates `SuperTrend` + `MACD`.
- 5) Bot decides exit/entry; updates global `bot_state` and persists trade rows.
- 6) Bot broadcasts updated state via WebSocket; UI updates panels in real-time.
- 7) Analytics page fetches `/api/analytics`; backend computes stats from DB.

Design patterns used

- Thin controller: `backend/server.py` delegates business logic.
- Service façade: `backend/bot_service.py` wraps engine + persistence.
- Singleton-ish state: config + bot_state dicts in `backend/config.py`.
- Push + pull: WebSocket streaming + periodic REST polling fallback.

3. Folder & file-level walkthrough

Root

- `README.md` — feature overview + run steps.
- `docker-compose.yml` — two services: `backend:8001` and `nginx frontend:80`.
- `init_database.py` — creates DB tables.
- `backend_test.py` — HTTP smoke tests for endpoints.
- `tests/` — currently empty besides `__init__.py`.
- `memory/PRD.md` — product requirements and implementation checklist.

backend/

- `server.py`
- Defines FastAPI app and routes under `/api`.
- Defines WebSocket endpoint `/ws`.
- Implements `ConnectionManager` (active websocket list + broadcast).
- Startup: `init_db` + `load_config`.
- `bot_service.py`
- Lazy constructs `TradingBot`.
- `start_bot/stop_bot/squareoff_position` call engine.
- `get_bot_status/get_market_data/get_position/get_config` expose state for UI.
- `update_config_values` applies updates + persists.
- `trading_bot.py`
- `TradingBot.run_loop`: 1-second loop building candles and applying strategy.
- `process_signal_on_close`: primary entry/exit state machine.
- `check_tick_sl/check_trailing_sl_on_close`: risk exits.

- enter_position/close_position: creates/clears positions + orders + DB writes.
- dhan_api.py
- Wraps dhanhq SDK.
- Fetches quotes/option chain.
- Resolves option security id and expiry.
- Places market orders.
- Optionally verifies order fills by polling.
- indicators.py
- SuperTrend(period, multiplier) and MACD(fast, slow, signal).
- database.py
- SQLite schema init.
- load_config/save_config.
- save_trade/update_trade_exit.
- get_trade_analytics (aggregations).
- save_candle_data/get_candle_data.
- config.py
- bot_state (runtime) + config (settings) + DB_PATH.
- utils.py
- is_market_open/can_take_new_trade/should_force_squareoff (IST time helpers).
- indices.py
- per-index metadata + strike rounding.
- models.py
- ConfigUpdate request model.

frontend/

- nginx.conf
- Proxies /api and /ws to backend service in Docker.
- src/App.js
- Global app context state.
- Fetches initial data via REST.
- Connects to WebSocket and updates UI state.
- Routes: / (Dashboard), /settings, /analysis.
- src/pages/Dashboard.jsx
- Composes panels: TopBar, PositionPanel, ControlsPanel, NiftyTracker, TradesTable, DailySummary, LogsViewer.
- src/pages/Settings.jsx
- Credentials and risk parameter forms; posts config updates.
- src/pages/TradesAnalysis.jsx
- Fetches /api/analytics and implements filters + derived statistics.

4. Execution flow

What happens when the app starts

- Backend uvicorn starts FastAPI app.
- Startup hook initializes DB and loads persisted config.
- Bot engine is idle until started via API.

Entry point

- Backend: uvicorn server:app (port 8001 in Docker).
- Frontend: nginx serves React build; in dev use craco start.

Runtime flow step-by-step

- 1) UI loads and calls /api/status, /api/config, /api/trades, etc.
- 2) UI opens WebSocket to /ws.
- 3) User clicks Start → /api/bot/start.
- 4) TradingBot.start() spawns TradingBot.run_loop().
- 5) run_loop each tick:
 - market gating
 - fetch quotes (spot, option when in a position)
 - update candle aggregates
 - tick-level risk exits
 - candle-close indicator updates
 - reversal exit / new entry decision
 - broadcast state_update

Background processes / loops

- Backend: trading loop asyncio task; background DB updates via asyncio.create_task.
- Frontend: WebSocket reconnect + 5s REST polling.

5. Configuration & environment

Env variables

- Frontend: REACT_APP_BACKEND_URL (API base, WebSocket base).
- Backend: loads backend/.env if present, but most config is stored in SQLite.

Config files

- SQLite config table persists config values.
- frontend/nginx.conf provides reverse proxy in Docker.

Secrets

- Dhan access token + client id are saved via Settings and persisted in SQLite as plaintext.

6. External dependencies & integrations

- Broker API: DhanHQ (dhanhq Python SDK).
- Database: SQLite via aiosqlite.
- No queues/message brokers.

7. Core business logic

Decision making logic

- Candle timeframe in seconds (default 5s).
- Signal: SuperTrend direction.
- Optional MACD confirmation.
- Optional HTF SuperTrend alignment (60s).

Entry rules (summary)

- Must be within market hours + entry cutoff.
- Must respect max trades/day.
- Must respect flip-only and order cooldown.

- If enabled: MACD confirms direction.
- If enabled: HTF filter aligns direction.

Exit rules (priority)

Tick-level:

- 1) daily max loss breach
- 2) max loss per trade
- 3) target points
- 4) trailing SL

Candle-level:

- exit immediately on SuperTrend reversal (unless min_hold_seconds blocks it)

Scheduled:

- force squareoff at 15:25 IST.

8. Error handling & logging

- Bot loop catches exceptions and continues after sleep.
- Logging to stdout + backend/logs/bot.log.
- UI fetches logs via /api/logs.

9. Performance & scalability

- Dhan SDK calls are synchronous and may block the async event loop.
- WS broadcasts are sequential per client.
- Not designed for multiple concurrent bots.

10. Security considerations

- No authentication/authorization on API.
- CORS allows *.
- Secrets stored plaintext in SQLite.

11. How to run & test locally

Docker:

- docker-compose up -d --build
- Frontend: http://localhost
- Backend: http://localhost:8001/api

Local dev:

- cd backend && python -m uvicorn server:app --reload --port 8001
- cd frontend && yarn start

12. Known issues / code smells

- backend/database.py: save_trade() appears to use db without opening a connection.
- Risk sizing qty semantics inconsistent (lots vs absolute qty).
- bypass_market_hours does not bypass TradingBot.is_within_trading_hours.
- Potential for multiple bot instances (global TradingBot plus lazy singleton).
- Settings page hardcodes lot sizes that may not match backend indices.

13. Improvement suggestions

- Fix DB trade insert path first (save_trade connection bug).
- Standardize quantity units and position sizing.
- Offload broker calls to threads to avoid blocking event loop.
- Add authentication for control/config endpoints.
- Encrypt or externalize secrets; restrict CORS in production.
- Add pytest tests for entry/exit rules and analytics.