Arch Documentation

- New Page

New Page

Architecture: current and target

This document describes the system's design: an **Engine**, any number of **Providers** and one **Exchange** service per trading venue. Phases 1–5 of the migration (§8) are implemented: the engine talks to the exchanges only through this contract, and §3 now describes the architecture before the migration. `docs/provider-protocol.md` stays the reference for the legacy `PricingInfo` subject that Internal and the bookmaker feeds still use (phase 6).

1. Purpose

Use exchanges and providers to find markets that are priced wrong, and buy them when they are cheaper than the real probability.

The system is run by **one person**: one operator, one wallet per exchange. There are no user accounts or roles.

2. Roles

Role	What it does	Talks over	Owns
Engine	Receives prices from exchanges and providers, matches feeds of the same game, builds the consensus line, runs strategies (paper and real), executes on an exchange, shows metrics (dashboard) and logs matched data to Influx. Never names a specific exchange in code.	NATS (in), HTTP (to exchanges), NATS WebSocket (dashboard)	`GlobalGameId`, game state (Redis), mappings, trades and paper fills (Mongo), price history (Influx)
Provider	Unmanaged price source, possibly on a remote network. Publishes its games and prices; managing it is optional.	NATS only (publish)	Its own `GameId`s

Role	What it does	Talks over	Owns
Exchange	A price provider that can also trade on its own platform (today: Polymarket). One instance per exchange. Publishes prices like a provider and serves an HTTP API for availability, subscriptions, trades and orders.	NATS (publish) + HTTP API	Its `GameId`s (e.g. a Polymarket slug and its `-more-markets` slug under one id), `AssetId`s, wallet, orders

What changes against today: trading (order placement, the wallet's fills, market resolution) moves from the engine into the exchange, and the engine stops knowing which exchange it talks to. Strategies, paper trading, matching and consensus stay in the engine.

3. Architecture before the migration

```
flowchart LR
    subgraph Providers
        INT["Internal (Override)"]
        BK["Bookmaker feeds (*-TX ...)"]
    end
    subgraph PP["PolymarketProvider"]
        GAM["Gamma discovery + mapper"]
        WS["CLOB WebSockets"]
        MON["Monitor :8080"]
    end
    subgraph ENG["ArbitrageEngine"]
        ROUTE["Routing / matching"]
        CONS["Consensus"]
        STRAT["Strategies"]
        PAPER["PaperTrader"]
        LIVE["LiveTrader / ManualTrader"]
        HIST["TradeHistory (wallet poll)"]
        RES["MarketResolver (Gamma)"]
        WEB["Dashboard :8088"]
    end
    EXEC["Python executor (FastAPI)"]
    PMAPI["Polymarket data-api / Gamma"]
    NATS(("NATS"))
    REDIS[("Redis")]
```

```
    MONGO[("Mongo")]
    INFLUX[("Influx")]

    INT -- "arbitrage.prices.internal" --> NATS
    BK -- "arbitrage.prices.*" --> NATS
    PP -- "prices, catalog, retired, trades" --> NATS
    NATS -- "subscriptions, refresh" --> PP
    NATS --> ROUTE --> CONS --> STRAT
    STRAT --> PAPER
    STRAT --> LIVE --> EXEC
    HIST --> PMAPI
    RES --> PMAPI
    ENG --> REDIS
    ENG --> MONGO
    ENG --> INFLUX
    WEB -. "arbitrage.ui.*" .- NATS
```

Today's control plane is all NATS (`ControlSubjects` in `Arbitrage/PolymarketSportFeed/Engine.cs`):

Subject	Direction	Content
`arbitrage.prices.<source>`	provider → engine	`PricingInfo`
`arbitrage.catalog.polymarket`	provider → engine	`CatalogMessage`, every 60 s
`arbitrage.control.polymarket.subscriptions`	engine → provider	`SubscriptionSet` (full wanted set, every 60 s and on change)
`arbitrage.control.polymarket.sync`	provider → engine	ask for the wanted set (startup)
`arbitrage.control.polymarket.refresh`	engine → provider	resend these events' books
`arbitrage.control.polymarket.retired`	provider → engine	events finished
`arbitrage.trades.polymarket`	provider → engine	`MarketTrade`, the public trade tape

Things in the current shape that the target removes:

- The engine trades directly: `ManualTrader` and `LiveTrader` call the Python executor, `TradeHistory` polls the wallet on Polymarket's data-api, `MarketResolver` asks Gamma for results. All of that is Polymarket-specific code inside the engine.
- The engine refers to `"Polymarket"` by name (strategies' target book, dashboard status, the on-demand subscription logic), and Polymarket order details (`ConditionId`, `NegRisk`, `TickSize`) travel on every price message.
- Feed ids are the provider's internal ids; Polymarket's `-more-markets` merge is a provider detail the engine partly knows about.

4. Target architecture (implemented)

```
flowchart LR
    subgraph PROV["Providers (unmanaged, may be remote)"]
        P1["Internal"]
        P2["Bookmaker feeds"]
    end
    subgraph EXS["Exchanges (one per venue)"]
        X1["Polymarket Exchange"]
        X2["Future exchange"]
    end
    subgraph ENG["Engine"]
        ROUTE["Matching"]
        CONS["Consensus"]
        STRAT["Strategies"]
        PAPER["Paper trading"]
        EXEC["Execution client"]
        RECON["Subscription reconcile"]
        ORD["Order tracking"]
        WEB["Dashboard"]
    end
    NATS(("NATS"))
    REDIS[("Redis")]
    MONGO[("Mongo")]
    INFLUX[("Influx")]

    P1 -- "Book / Update" --> NATS
    P2 -- "Book / Update" --> NATS
    X1 -- "Book / Update" --> NATS
    X2 -- "Book / Update" --> NATS
    NATS --> ROUTE --> CONS --> STRAT
    STRAT --> PAPER
    STRAT --> EXEC
    EXEC -- "POST /trade" --> X1
    ORD -- "GET /orders" --> X1
    RECON -- "GET /availability, POST /subscription" --> X1
    RECON -. "same API" .-> X2
    ENG --> REDIS
    ENG --> MONGO
```

```
    ENG --> INFLUX
```

- **Data plane (NATS)**: everyone publishes `Book` and `Update` messages in one envelope. Providers and exchanges look the same to the engine's router.
- **Control plane (HTTP)**: only exchanges have it. The engine needs each exchange's base URL in config; providers need nothing but the NATS URL.
- **Execution**: a strategy signal names an exchange and an `AssetId`. With several exchanges the engine can pick where to buy; the contract below already allows it.

5. Contracts

Enum integer values are part of the wire format: never reorder them, only append. The market enums (`Period`, `MarketType`, `MarketTarget`) and their conventions (home-margin spreads, 3-way moneylines, team totals) stay exactly as in `docs/provider-protocol.md`.

5.1 Exchange HTTP API

Method	Path	Purpose
GET	`/health`	Liveness, identity and money available
GET	`/availability`	Every game the exchange can open, with its state
POST	`/subscription`	Open or close one game's price stream
POST	`/trade`	Place an order
GET	`/orders`	Our orders, their fills and settlement
GET	`/markets?assetIds=a,b`	Market state of any assets (positions that are not orders: paper fills)

The API moves money: it must only be reachable from the engine's network (see §9). An exchange may also require a shared key in the `X-Api-Key` header on `/trade` and `/orders`.

`GET /health`

```
{ "time": "2026-10-24T18:02:11Z", "name": "Polymarket", "liquidityAvailable": 412.37 }
```

`liquidityAvailable` is the balance usable for trading, in the exchange's settlement currency (USDC for Polymarket).

`GET /availability`

```
[
  {
    "gameId": "pm-7f3a1c",
    "teamNames": ["Boston Celtics", "New York Knicks"],
    "matchDate": "2026-10-25T00:00:00Z",
    "startTimeUtc": "2026-10-25T23:30:00Z",
    "updateType": "Accumulative",
    "isEnabled": true
  }
]
```

- `gameId` is created and owned by the exchange, and must stay stable across restarts (the engine stores it in Redis, Mongo and Influx). Polymarket: one id covers the event slug and its `-more-markets` companion.
- `teamNames` are `[home, away]`; the engine only uses them to match the game.
- `startTimeUtc` is the scheduled kick-off when known (pregame/in-play rules depend on it).
- `updateType`: `Accumulative` (an `Update` carries only what changed) or `Override` (an `Update` is the full board for that game).
- `isEnabled`: the exchange is currently streaming this game.
- Optional, for display: `title`, `url` (the game's page on the exchange) and `manual` (opened by hand on the exchange: it streams whatever the engine asks).
- A game that is finished or decided disappears from the list: that is how the engine learns it is retired.

`POST /subscription`

```
{ "gameId": "pm-7f3a1c", "enable": true }
```

One call per game. `enable: true` opens the stream (if not open already) and the exchange starts publishing `Update`s for it; `false` stops it. The call is idempotent.

The engine does not trust its own memory of what is open: every 60 s it reads `/availability`, compares `isEnabled` with the games it wants, and sends `/subscription` for every difference. An exchange restart (everything `isEnabled: false`) therefore heals within a minute, and the exchange never has to persist subscriptions. The engine only disables games it enabled itself in this run, so a game enabled by hand on the exchange stays open.

`enable: true` on a game that already streams is a **resend**: the exchange forgets its last top-of-book for that game and publishes the whole book again (the engine asks for it after clearing a board, e.g. a series split, and at startup for the pregame games). Unknown `gameId` → 404.

`POST /trade`

Request (by `AssetId`, the normal path):

```
{
  "clientOrderId": "tennis-live-v2:9c1e...:1",
  "assetId": "71321045679252212594626385532706912750332728571942532289631379312455583992563",
  "amount": 5.0,
  "sell": false,
  "price": 0.43,
  "tradeType": "FAK",
  "expirationSeconds": 0
}
```

Or by market, when the caller has no `AssetId`: replace `assetId` with `"target": { "gameId": "pm-7f3a1c", "period": "Game", "type": "Moneyline", "targetType": "Home", "points": 0 }` and the exchange resolves it with its mapper.

Field	Meaning
`clientOrderId`	Generated by the engine (`ui:<guid>` for the dashboard, `live:<strategy>:<asset>:<ticks>` for real-money strategies). The exchange stores the order under it before sending it and answers `Declined: duplicate clientOrderId` (with the first order's `orderId`) to a second one, so a retry after a timeout can never buy twice.
`amount`	Buy: money to spend (before the exchange's fee). Sell: shares to sell. Each exchange also has its own per-trade cap.
`price`	Limit price in the asset's own terms (required). Rounded to the tick so the limit always holds: a buy is rounded down, a sell up.
`tradeType`	`FAK` (fill what the book has at or under the limit, cancel the rest), `FOK` (all or nothing), `GTD` (rest on the book until `expirationSeconds`).
`expirationSeconds`	Only for `GTD`. Polymarket clamps it to 120–600 s (the CLOB wants at least 180 s ahead and the executor adds 60 s).

Response:

```
{ "status": "Delayed", "message": "in-play order delay", "orderId": "0xab12...",
"clientOrderId": "tennis-live-v2:9c1e...:1", "price": 0.43 }
```

`status`: `Accepted` (matched or resting), `Declined` (rejected, `message` says why; a FAK/FOK that matched nothing is `Declined` on Kalshi), `Delayed` (outcome pending: Polymarket holds in-play sports orders before matching, and a delayed order may never fill; a timeout or lost answer is `Delayed` too, also on the engine's side when the exchange doesn't answer). `price` is the limit actually sent. The response never carries amounts: the final outcome always comes from `/orders`.

`GET /orders?since=<ISO time>`

`OrderStatus` adds `Resting` (a GTD order on the book with nothing matched yet).

```
[
  {
    "orderId": "0xab12...",
    "clientOrderId": "tennis-live-v2:9c1e...:1",
    "gameId": "pm-7f3a1c",
    "assetId": "7132...2563",
    "sell": false,
    "status": "Filled",
    "shares": 11.62,
    "avgPrice": 0.43,
    "filledAt": "2026-10-24T18:02:14Z",
    "marketStatus": "Resolved",
    "payout": 1.0,
    "currentPrice": 1.0
  }
]
```

Field	Meaning
`status`	`Delayed`, `Filled`, `Partial`, `Unfilled` (delayed or GTD order that expired without a match), `Rejected`, `Resting`
`marketStatus`	`Open`, `Closed` (trading stopped, result pending), `Resolved`
`payout`	Per share once resolved: `1` won, `0` lost, `0.5` void
`currentPrice`	Latest price of the token, to value open positions

This replaces the engine's old wallet poll and Gamma resolution. It also lists orders placed outside the engine (e.g. on the exchange's website), with no `clientOrderId`, so the Bets tab still sees the whole wallet. Fills never travel on NATS.

`GET /markets?assetIds=a,b`

```
[ { "assetId": "7132...2563", "marketStatus": "Resolved", "payout": 1.0, "currentPrice": 1.0,
"title": "Celtics vs. Knicks" } ]
```

The same market state as `/orders`, for assets that are not our orders: the engine resolves paper fills with it (`MarketResolver`, every 5 min, batches of 50).

5.2 NATS messages

Every message uses one envelope:

```
{ "messageType": "Book", "publisher": "Polymarket", "content": { } }
```

Subject: `arbitrage.<messageType lower case>.<publisher lower case>`, e.g. `arbitrage.update.polymarket`, so the engine subscribes to `arbitrage.book.>` and `arbitrage.update.>`.

Book

`content` is the same list as `/availability`. An exchange lists only its streaming games; a provider lists every game it has. Sent at startup, whenever the list changes (at most one per 5 s) and about every 5 minutes.

- **Provider**: its `Book` is its whole catalog. The engine routes it like `/availability`, and a game missing from 5 consecutive Books is retired (`CatalogMissesBeforeDrop`).
- **Exchange**: its `Book` only refreshes the names, date and kick-off of the games that stream (for their `Update`s) and never retires anything; an exchange's game is retired when it is missing from 5 consecutive `/availability` reads (about 5 minutes).
- The publisher's `updateType` (from its first entry) sets how the engine applies its `Update`s.

Update

`content` is one game: its `gameId` and the list of its markets that changed. One `Update` message per game.

```
{
  "messageType": "Update",
  "publisher": "Polymarket",
  "content": {
    "gameId": "pm-7f3a1c",
    "markets": [
      {
        "period": 1, "type": 1, "targetType": 3, "points": 0,
        "price": 0.43,
        "bestBid": 0.41,
        "timestampMilliseconds": 1792951334000,
        "assetId": "7132...2563",
        "reference": "Boston Celtics"
      }
    ]
  }
}
```

- The game id travels once per message, not on every market: a market item carries only the market fields (`period`, `type`, `targetType`, `points`, `price`, `bestBid`, `timestampMilliseconds`, `assetId`, `reference`).
- No team names: the engine matched the game through `Book`/`/availability`. An `Update` for a game it has not seen announced is routed from the feed's existing mapping (after an engine restart), or else dropped until the game's `Book` arrives.
- One message is one game, so the engine turns each `Update` into one `PricingInfo` for that `gameId`. A publisher with changes on several games sends several messages. If the same market (period, type, target, points) appears twice in one message, the later item wins.
- For an exchange, `price` is the **ask** (what a buy costs) and `bestBid` is optional. The engine uses the bid to skip empty books (ask − bid over 20¢ means the ask is not a real price). Providers send a fair price and no bid.
- `assetId` is the exchange's id for that outcome; the engine sends it back in `/trade`. Providers leave it null. `ConditionId`, `NegRisk`, `TickSize` and `BestAsk` no longer travel: they are the exchange's business.
- Override publishers: an `Update` replaces the game's full board. To empty a game's board, send its `Update` with an empty `markets` list.
- Compatibility: until 2026-10-27 `content` was a flat list of items, each carrying its own `gameId`. The engine still reads that shape (grouped per item, counted as `UpdatesLegacy` in the engine and logged once per publisher), so a publisher not moved yet keeps working; new publishers must send the shape above. In code: `Arbitrage.Contracts.GameUpdate` is the content, `UpdateBatch` collects quotes across games and `Outbox.EnqueueUpdates` sends one message per game.

5.3 Enums

Enum	Values
`UpdateType`	`Accumulative = 0`, `Override = 1`
`TradeType`	`FAK = 0`, `FOK = 1`, `GTD = 2`
`TradeStatus`	`Accepted = 0`, `Declined = 1`, `Delayed = 2`
`OrderStatus`	`Delayed = 0`, `Filled = 1`, `Partial = 2`, `Unfilled = 3`, `Rejected = 4`, `Resting = 5` (GTD on the book, nothing matched yet)
`MarketStatus`	`Open = 0`, `Closed = 1`, `Resolved = 2`

6. Sequences

6.1 Discovery, matching and subscription

```
sequenceDiagram
    participant X as Exchange
    participant P as Provider
    participant N as NATS
    participant E as Engine

    P->>N: Book (all its games)
    N->>E: Book
    E->>E: match games (team names + date)
    X->>N: Book (enabled games)
    E->>X: GET /availability
    X-->>E: games, isEnabled
    E->>E: match, decide wanted set (game also has an Internal feed, or opened by hand)
    E->>X: POST /subscription {gameId, enable: true}
    X->>N: Update (full book first, then changes)
    N->>E: Update
    E->>E: consensus, strategies

    loop every 60 s
        E->>X: GET /availability
        X-->>E: isEnabled per game
        E->>X: POST /subscription for each difference
    end

    Note over X: restart: nothing enabled
    E->>X: GET /availability (next loop)
    E->>X: POST /subscription again for every wanted game
```

6.2 A strategy buy

```
sequenceDiagram
    participant S as Strategy
    participant E as Engine (execution)
    participant X as Exchange
    participant M as Mongo

    S->>E: signal (exchange, assetId, ask, fair)
    E->>X: POST /trade {clientOrderId = order id, assetId, amount, price, FAK}
    X->>X: store order by clientOrderId, send it
```

```
    X-->>E: Accepted / Declined / Delayed
    E->>M: StrategyOrders (strategy, fair, outcome)
    loop every 60 s (exchange): wallet fills
        X->>X: fill → oldest open order of ours on that token and side
    end
    loop every 60 s (engine, TradeHistory)
        E->>X: GET /orders?since=now-2d
        X-->>E: shares, avgPrice per order (+ fills placed elsewhere)
        E->>M: Trades: new fill tagged with the order's strategy + fair (by clientOrderId)
    end
    Note over E: no fill 5 min after a Delayed order: Unfilled, token and game slot free again
(max 3 attempts)
    loop every 5 min (engine, MarketResolver)
        E->>X: GET /markets?assetIds=... (tokens not resolved yet, real and paper)
        X-->>E: marketStatus, payout, currentPrice
        E->>M: TradeAssets → Won / Lost / Void on the Bets tab
    end
```

6.3 Engine startup

```
sequenceDiagram
    participant E as Engine
    participant R as Redis
    participant X as Exchange
    participant N as NATS

    E->>X: GET /health (each host in "exchange", up to 10 s): its name
    E->>R: restore games and feed mappings
    E->>N: subscribe arbitrage.prices.>, arbitrage.book.>, arbitrage.update.>
    E->>X: GET /availability (reconcile loop, at once and every 60 s)
    Note over E: known games route at once (mapping restored)
    Note over E: a provider's new game waits for its next Book (at most ~5 min)
    E->>X: POST /subscription for wanted games not streaming
    E->>X: POST /subscription (resend) for the pregame games, 30 s later
```

7. Before ? after the migration

Before	Now
`Arbitrage/PolymarketProvider`	Polymarket Exchange (same discovery, mapper and WebSockets, plus the HTTP API)
`CatalogMessage` on `arbitrage.catalog.polymarket`	`Book` + `GET /availability`
`SubscriptionSet` on `...subscriptions` / `...sync`	`POST /subscription` + the engine's 60 s reconcile
`...retired`	game drops out of `Book` / `/availability`
`...refresh` (resend books)	`POST /subscription {enable: true}` on an already open game resends its full book
`arbitrage.prices.<source>` + `PricingInfo`	`Update` envelope; team names and dates move to `Book`
`PricingInfo.InternalId` (provider feed id)	`gameId` (owned by the publisher)
`MarketPrice.ConditionId/NegRisk/TickSize/BestAsk`	removed from the wire, kept inside the exchange
`ManualTrader` / `LiveTrader` → Python executor	`POST /trade` on the exchange; the executor becomes the exchange's internal detail
`TradeHistory` (data-api wallet poll)	`GET /orders`
`MarketResolver` (Gamma resolution)	`GET /orders` (`marketStatus`, `payout`)
`PaperTrader`	unchanged, stays in the engine
`arbitrage.trades.polymarket` (public tape) and `maker-v3`	removed; `maker-v3` is retired, its paper fills stay in Mongo
`"Polymarket"` checks in the engine (strategies, dashboard, subscriptions)	"is an exchange" (it has an HTTP API in config)
`Engine.UpdateModes` (hard-coded)	`updateType` in `Book` / `/availability`

8. Migration phases

Each phase deploys on its own and keeps today's behaviour working. **Status (2026-10-24): phases 1–5 are implemented together** (the engine reads only `Book`/`Update` from exchanges; the Polymarket provider is now `Arbitrage/PolymarketExchange`); phase 6 waits on the Internal feed, which lives outside this repo.

1. **Envelope alongside the old subjects.** The engine accepts `Book`/`Update` as well as `PricingInfo`. The Polymarket provider publishes both. `updateType` from `Book` replaces the hard-coded `UpdateModes` when present.
2. **Polymarket exchange API, read side.** `GET /health`, `GET /availability`, `POST /subscription` in `PolymarketProvider`, with a stable `gameId` per event + companion (mapped from today's ids so Redis/Mongo/Influx history keeps working).
3. **Engine reconciles over HTTP.** Exchange base URLs in config; the reconcile loop replaces `SubscriptionLoop`; catalog/subscriptions/sync/refresh/retired subjects are removed from both sides.

4. **Trading moves to the exchange.** `POST /trade` and `GET /orders` in the exchange (it takes over the executor client, the wallet poll and Gamma resolution). The engine's `LiveTrader`, `ManualTrader`, `TradeHistory` and `MarketResolver` call them instead. `maker-v3` is unregistered.
5. **Engine without exchange names.** Replace the `"Polymarket"` checks with the exchange role; signals carry the exchange name. Remove the Polymarket-only fields from `MarketPrice`.
6. **Providers on the envelope.** Internal and the bookmaker feeds switch to `Book`/`Update`; then the engine drops `PricingInfo` and `docs/provider-protocol.md` is rewritten from §5. *(Not done: the engine already routes a provider's `Book` as its full game list and accepts its `Update`s.)*

What the engine does with the contract

- **Config**: `"exchange": ["polymarket-provider:8080", "kalshi-exchange:8090"]` (or `engine:exchanges`), optional `engine:exchangeApiKey` sent as `X-Api-Key`. Each exchange's name comes from its `/health` (`ExchangeRegistry`, every 30 s).
- **Reconcile** (`Engine.ReconcileLoop`): every 60 s `/availability` per exchange → routing (`ProcessCatalog`), retirement after 5 missing reads, and one `POST /subscription` per difference between what streams and what the engine wants (an Internal feed in the game, or opened by hand). The engine only closes games it opened in this run. Enabling an open game = resend.
- **Prices**: `arbitrage.update.>` → one `PricingInfo` per message (`content` = `gameId` + `markets`; names/date from `Book` / `/availability`), same `ProcessNewData` as the legacy subject. Exchanges never count as books.
- **Trading**: dashboard Buy and `LiveTrader` → `POST /trade` (client order ids `ui:<guid>` / `live:<strategy>:<asset>:<ticks>`); `TradeHistory` reads `/orders` every minute into Mongo `Trades`; `MarketResolver` reads `/markets` into `TradeAssets`.
- **Strategies**: `EdgeOptions.Exchange` (default Polymarket); paper copies on every other exchange (`<profile>@kalshi`). `maker-v3` is retired (no trade tape in the contract).

9. Production concerns

- **Exchange API exposure.** `/trade` spends real money and has no auth in this design. Bind it to the Docker network the engine shares, or put a shared API key in front of it. The dashboard (`:8088`) has the same problem today: anyone who reaches it can trade up to the cap.
- **Build.** The engine references `Propagate.Network.dll` from a sibling repo by HintPath, so it only builds on a machine with that repo built in Release/x64. A NuGet package (Nexus already hosts the images) or a submodule makes the build reproducible.
- **Tests and CI.** There are none. The first candidates: the team matcher (`TeamPairMatcher`, with the prod name pairs as fixtures), consensus rules, strategy gates, and contract tests for the exchange API. Then a pipeline that builds and pushes the images instead of

`aut.bat`.

- **Alerting.** The Home tab shows health but nothing notifies the operator: a quiet feed, NATS drops, Influx queue drops, a failed or long-delayed real order, or exposure near `maxOpenUsdc`.
- **Secrets.** API keys and wallet access live in the mounted `config.json`. With trading moving into the exchange, the wallet key only needs to be in the exchange's config, not the engine's.

10. Implemented exchanges

Kalshi (`Arbitrage/KalshiExchange`)

The first service written against this contract (the shared types are in `Arbitrage/Contracts`). The engine consumes it like any exchange: its asks show on the dashboard and Top edges, and the paper strategies run on it (`<profile>@kalshi`); the dashboard's Buy is off for it unless `engine:trading:exchanges` lists it.

- **Discovery** every `kalshi:discoveryMinutes` (10): for each league prefix in `kalshi:leagues` (default KXNBA, KXNFL, KXMLB, KXNHL, KXNCAAF, KXEPL, KXLALIGA, KXUCL, KXATP, KXWTA), the open events of the series `<prefix>GAME` (tennis: `MATCH`), `SPREAD`, `TOTAL`, `TEAMTOTAL`, `1HSPREAD`, `1HTOTAL`, `2HSPREAD`, `2HTOTAL`. The events of one game share a suffix and become one game: `GameId` = prefix + suffix, e.g. `KXNBA-26OCT24MIACHA`. The winner event is required (it names the teams). The catalog is cached in Redis (`Arbitrage.Exchange.Kalshi.Games`) so `/availability` answers at once after a restart.
- **Home team**: Kalshi lists US games away team first and soccer home team first; the order comes from the sport in the markets' `custom_strike` (`soccer_team`, `tennis_competitor` = first listed is home), overridable per league (`{"prefix": "KXMLS", "homeFirst": true}`). Tennis: the first-listed player is Home, like the engine's convention.
- **Start time**: only when the ticker carries it (`26OCT242000` = 20:00 US Eastern). Kalshi's `occurrence_datetime` is about the expected end of the game, not the start, so it is not used.
- **Assets**: every market is a YES and a NO outcome, `AssetId` = `<ticker>:yes` / `<ticker>:no`. Moneylines publish the YES side of Home/Away/Draw only. Spreads ("X wins by more than n") are mapped to the home margin convention; totals and team totals to Over/Under. The NO quote mirrors the YES book: NO ask = 1 − YES bid, NO bid = 1 − YES ask.
- **Prices**: Kalshi's WebSocket (`ticker` channel, one subscription per enabled game) needs an API key even for public data; without a key the service polls the enabled games over REST every 10 s. Enabling a game publishes its full book from REST.
- **Trading** (needs `kalshi:keyId` + `kalshi:privateKey` or `privateKeyPath`, the Ed25519 or RSA key from Kalshi's API settings): `POST /portfolio/events/orders` on the single YES book (buying NO = an ask on YES at 1 − price, rounded so the asset's limit is never exceeded), whole contracts, buys capped by `kalshi:maxUsdcPerTrade` (10). FAK = `immediate_or_cancel`, FOK = `fill_or_kill`, GTD = `good_till_canceled` with an expiration. A FAK/FOK that

matches nothing is `Declined`; a timeout is `Delayed` (only `/orders` can tell). Kalshi's taker fee is charged on top of the amount. Every order is saved in Mongo `Arbitrage.KalshiOrders` keyed by `clientOrderId` before it is sent, which makes retries idempotent.

- **Orders**: `/orders?since=` reads Kalshi's orders and fills (average price in the asset's terms) and each market's state (finalized markets cached in Mongo `Arbitrage.KalshiMarkets`): `Resolved` only once `finalized`, payout 1/0 from the result, or the settlement value for a scalar result. Orders sent from elsewhere (website) are listed too, without a `clientOrderId` .
- **Config** (`config/appsettings.json`, same `ConnectionStrings` and `environment` as the other services; environment variables like `kalshi__keyId` and `--kalshi:demo=true` override it): `kalshi {demo, keyId, privateKey | privateKeyPath, port (8090), leagues[], maxUsdcPerTrade, apiKey, discoveryMinutes}`. Image: `aut.bat` → `nexus.home/kalshi-exchange`.
- **Monitor page** at `/` (outside the contract, like the Polymarket monitor; Kalshi-green theme): metric cards (feed, NATS out, catalog, Kalshi API calls and 429s, account balance, process), the availability list filterable by state and league with an Enable/Disable button per game (calls `/subscription`), and each game's markets with bid/ask (grey = discovery snapshot, not yet published), plus the last orders sent. `#game=<id>` opens a game. It polls `/api/state` every 2 s; `/api/game?id=` and `/api/orders/recent` feed the details. `/status` is a short JSON summary.

Polymarket (`Arbitrage/PolymarketExchange`)

The former PolymarketProvider. `GameId` = the Gamma event id (the feed id the engine already had, so Redis, Mongo and Influx history carry on); the `-more-markets` companion is merged under it.

- **API** on the monitor's port (8080, `ExchangeApi.cs`): the contract plus `/api/exchange` for the monitor's strip. `/trade` and `/orders` need `X-Api-Key` when `polymarket:apiKey` is set.
- **Prices**: CLOB WebSocket per enabled game (or manual slug) → `Update` on `arbitrage.update.polymarket`; `Book` of the streaming games every 5 min and on changes (at most one per 5 s).
- **Trading** (`PolymarketTrading.cs`): the Python executor (`PolymarketSidecarClient`), the account from MySQL; the executor's url in `polymarket:clobHost`; `polymarket:trading` or, falling back, `engine:trading` (`accountId`, `maxUsdcPerTrade`, `expirationSeconds`, `address`). Orders in Mongo `PolymarketOrders` keyed by `clientOrderId` before they are sent; the wallet's fills (data-api, every minute) in `PolymarketFills`, each given to the oldest open order of ours on that token and side whose window holds it; an order with no fill is `Delayed` for 3 min (GTD: `Resting` until it expires), then `Unfilled`. Market state from Gamma by token (resolved ones cached in `PolymarketMarkets`).
- **Monitor page** (white Polymarket theme): channels, manual slugs, and a strip with trading state, balance, games streaming, updates/s and Books published.

Betfair (`Arbitrage/BetfairExchange`)

Betfair Exchange (global API-NG endpoints), port 8092, publisher `Betfair`. Paper only in the engine (the `@betfair` strategy copies); its own `/trade` is off unless `betfair:trading` is true.

- **Login**: an app key plus username/password, by certificate (`certPath` + `certKeyPath`, or a `.pfx` with `certPassword`; Betfair's recommendation for bots) or interactive login. The session is kept alive every 2 h and renewed on `INVALID_SESSION_INFORMATION`; a failed login is not retried for a minute (Betfair locks accounts after repeated failures).
- **Catalog** (`listMarketCatalogue`, every 10 min, 6-hour windows of up to 1000 markets per market type): `GameId` = the Betfair event id. Markets: `MATCH_ODDS` / `MONEY_LINE` (moneyline; soccer's draw = `Draw`; a US game with both uses the money line), soccer `OVER_UNDER_05..55` (totals) and `ASIAN_HANDICAP` half lines only (spreads from the home team's margin: whole and quarter lines refund part of the stake, which a binary price can't express). Home = the first name in "A v B", the second in "A @ B" (US sports). Asset ids: `<marketId>:<selectionId>`, plus `:<handicap>` on Asian handicap lines.
- **Prices**: `listMarketBook` (best offer, 40 markets per call) every `pollSeconds` (5) for the enabled games. Commission on net winnings (`commission`, 5%) is folded into the prices so they compare with the other exchanges: `price` (back at odds O) = 1 / (1 + (O − 1)(1 − c)), `bestBid` (lay at odds L) = (1 − c) / (L − c). A suspended market or a runner with no back offer is not published.
- **Trading** (off by default): a buy is a BACK; the limit price becomes the lowest acceptable odds, rounded up Betfair's ladder. FAK = `FILL_OR_KILL` with a minimum fill, FOK = `FILL_OR_KILL`, GTD = a `LAPSE` order cancelled at its expiry. Selling (laying) is not supported yet. Orders in Mongo `BetfairOrders` (client order id → a 32-char `customerRef`), settled markets in `BetfairMarkets`; `/orders` reads `listCurrentOrders` + `listClearedOrders`, with shares = payout net of commission.

SX Bet (`Arbitrage/SxBetExchange`)

SX Bet V3 order book (`https://api.sx.bet`, or the Toronto testnet with `sxbet:testnet`), port 8080 (`sxbet:port`), publisher `SxBet`. Paper only in the engine (the `@sxbet` strategy copies); its own `/trade` is off unless `sxbet:trading` is true and `sxbet:privateKey` is set. SX Bet is an on-chain peer-to-peer exchange: every market is a binary question with two outcomes, odds are implied probabilities on a 10^20 scale quoted from the **maker's** side, and the taker of a level pays `1 − maker`. Funds live in a proxy wallet (USDC on SX's chain) deployed from sx.bet.

- **Keys**: `sxbet:apiKey` (header `x-sx-api-key`, generated on sx.bet › Account › Overview; only one key is active at a time) is needed for the order book, the account routes and the realtime token. Discovery is public. Without a key the service answers `/availability` but publishes no price. `sxbet:privateKey` is the EOA that signs orders (the account's "maker"); the proxy wallet it owns must be deployed (`GET /user/proxy` says `deployed`) and funded.
- **Catalog** (`GET /markets/active` per sport, filtered to the mapped market types and paged by `paginationKey`, every 10 min): `GameId` = `sportXeventId` (e.g. `L18876509`), teams = `teamOneName` (SX's scheduled home) and `teamTwoName`, start = `gameTime`. Market types: the 2-way moneyline (226 "12 incl. OT", else 52), soccer's 1X2 (type 1: one market per outcome, "Toulouse" / "Not Toulouse", "Tie" / "Not tie", whose outcome one is our Home /

Draw / Away; soccer's type 52 is draw-no-bet and is skipped), totals (2, 28, tennis games 166), spreads (3, 342, tennis games 201: `line` is team one's handicap, the home-margin convention), first-half lines (63, 77, 53) and the period lines (202/21/64, 203/45/65, 204/46/66, 205) as quarters for basketball and American football and as sets for tennis. Only half lines: whole lines push and quarter lines (`isQuarterLineMarket`) half-win, which a binary price can't express. Asset ids: `<marketHash>:1` (outcome one) and `:2`. Cached in Redis (`Arbitrage.Exchange.SxBet.Games`). The `/markets/*` family is limited to 500 calls per minute per IP; a refresh of four sports is ~40 calls (2026-10-25: 88 games, 3,457 assets from 2,972 markets in 7 s).

- **Prices**: top of book from `GET /orders-v3/odds/best` (100 markets per call) every `pollSeconds` (5) for the enabled games, and from the Centrifugo channel `best_odds_v3:global` (one publication per top-of-book change, any market; `SxBetRealtime` is a minimal protocol-v2 JSON client: connect with the token from `GET /user/realtime-token-v3/api-key`, subscribe, answer the empty-frame pings, refresh the token before its ttl, reconnect with a backoff) while it is connected, the poll then only re-seeding every 30 s. For asset `:1` the ask is `1 − q` of the best maker level on outcome two and the bid is the best maker level on outcome one (and the reverse for `:2`); a level whose taker capacity `size × (1 − q) / q` is under `sxbet:minSize` (1 USDC) counts as empty. The account's **taker payout fee** (`GET /user/fees-v3`, charged on the profit of a winning bet; `sxbet:takerFee`, 2%, until the key answers) is folded into both so SX's prices compare with the other exchanges': `price = p / (p + (1 − p)(1 − c))`.
- **Trading**: a buy is a taker order on the asset's outcome (`POST /orders-v3`, `timeInForce` IOC for FAK, FOK, or GTC with the signed `expiry` = `expirationSeconds` away for GTD: SX expires it, nothing to cancel). The limit price (fee removed) becomes `percentageOdds`, the highest implied probability we pay, snapped **down** the odds ladder (`oddsLadderStepSize` × 10^15, 0.125% today) so the price paid never exceeds the limit; the stake is `totalBetSize` in base units (6 decimals; minimum `limits.orderSizeMinimumBaseUnits`). The eight-field EIP-712 `Order` (domain from `GET /metadata/obv3`) is signed with Nethereum; the order id is the digest. Sent with `waitForOutcome` (15 s), the answer carries the fill: `FULLY_FILLED` / `PARTIAL_FILL_*` = Accepted with the matched stake and blended odds, `RESTED` = Accepted (resting), `CANCELLED` (no liquidity) = Declined, `TIMEOUT` or a transport error = Delayed (only `/orders` can tell). SX's `clientOrderId` allows only `[A-Za-z0-9_-]`, so the engine's id is sent as `arb-<sha256 prefix>` and kept in the order document. Selling is not supported: a position is closed by betting the other outcome. A taker order on a live market waits the market's betting delay (`bettingDelay` in the metadata, 3–10 s) and matches against the book as it stands then. Orders in Mongo `SxBetOrders` (keyed by the engine's client order id before they are sent, so retries are idempotent), reported markets in `SxBetMarkets`. `dotnet run -- --selftest` prints a fixed order's address, digest and signature; checked on 2026-10-25 against `eth_account` (`encode_typed_data`) for the same inputs.
- **Orders**: `/orders?since=` joins our orders with `GET /trades-v3` (one row per bet; `ownOrderIds` carries our order ids) for the matched stake, `totalReturn` (shares = the net payout after the fee on profit, average price = stake / shares) and the settlement (`outcome` 1 / 2 / 0 = void → payout 1, 0, 0.5); a GTC with no bet yet is `Resting` until its expiry, an IOC/FOK with none is `Delayed` for 3 min, then `Unfilled`; `FAILED` (the on-chain

lock failed) is `Rejected`. Bets placed elsewhere on the account are listed without a client order id. `/markets` asks `GET /markets/find` (reported markets carry `reportedDate` and `outcome`).

- **Config** (`config/appsettings.json`, same `ConnectionStrings` and `environment` as the other services; environment variables like `sxbet__apiKey` and `--sxbet:trading=true` override it): `sxbet {apiKey, privateKey, testnet, sports[] (soccer, tennis, basketball, americanfootball, hockey, baseball, mma, esports…: SX sport ids behind), leagues[] (SX league ids, optional allowlist), daysAhead (2), pollSeconds (5), takerFee (0.02), minSize (1), trading (false), maxPerTrade (10), port (8080), serviceKey (our X-Api-Key for /trade and /orders), discoveryMinutes (10)}`. Image: `aut.bat` → `nexus.home/sxbet-exchange`.
- **Verified on 2026-10-25** with the user's API key: discovery, `/availability`, `/subscription`, `/markets`, the best-odds poll, the realtime channel (9,000 publications in a minute, ~130 for one enabled NFL game), the fee (1% on this account) and the signing (equal to eth_account's). Order placement and `/orders` have not been run: they need `sxbet:privateKey` and a funded proxy; try them on the Toronto testnet first.
- **Monitor page** at `/`: the same layout as the other monitors, in sx.bet's own dark palette (warm grey surfaces, mint `#5dfcce` for bids as on the site, coral for asks, Manrope and Geist Mono from Google Fonts with system fallbacks): cards for the feed, the realtime channel, NATS out, the catalog, the SX API (calls, errors, 429s), the account (balance, taker fee, proxy state) and the process; the availability list with Enable/Disable per game and each game's markets with the taker's implied probability, the size that can be taken, bid/ask and the main-line tag; the last orders sent.

Limitless (`Arbitrage/LimitlessExchange`)

Limitless Exchange (`https://api.limitless.exchange`, Base mainnet, chain 8453, no testnet), port 8080 (`limitless:port`), publisher `Limitless`. Paper only in the engine (the `@limitless` strategy copies); its own `/trade` is off unless `limitless:trading` is true and `limitless:privateKey` is the EOA of the profile behind the token. Limitless is a CLOB prediction market: every market is a YES/NO pair of conditional tokens backed by 1 USDC, quoted 0–1 in USDC per share on one order book (the YES side; a YES bid at q is a NO ask at 1 − q). Its sports markets are replications of Polymarket's and of a sports data feed (`metadata.externalProvider`).

- **Keys**: a scoped API token (limitless.exchange › profile › API tokens › Derive: `tokenId` = `limitless:apiKey`, the base64 `secret` = `limitless:secret`, shown once). Every signed request carries `lmts-api-key`, `lmts-timestamp` (ISO-8601, within 30 s of the server) and `lmts-signature` = base64 HMAC-SHA256 with the decoded secret over `timestamp\nMETHOD\npath?query\nbody` (`LimitlessClient.SignedHeaders`). Market data and order books are public: without a token the service lists games and publishes prices; the token is needed for `/profiles/me`, orders and the portfolio. `limitless:privateKey` is the EOA that signs orders; the profile must be in `eoa` trade-wallet mode (`PUT /profiles {tradeWalletOption: eoa}`), else `/trade` is declined with the reason.
- **Catalog** (`LimitlessCatalog`, every 10 min): each configured market page (`limitless:pages`, `/sport` by default; `/esports` is the other) is resolved with `GET /market-pages/by-path` and listed with `GET /market-pages/{id}/markets?limit=100&page=n`. A listed

market is a match market when its metadata carries `homeTeam`, `awayTeam` and `startMatchTimestampInUTC` and its tab is matches / games / fights / props; markets are grouped into games by the provider's event id (`fixtureId`, `eventId`, `externalMatchId`...), `GameId` = `<sport-type>:<event id>` (`basketball:1215090`), start = the kick-off, league = `leagueNameFull` / `tournamentName`. Mapped: the 2-way moneyline (a `single` market: YES pays on `outcomeLabels[0]`, the home team / player 1 unless the labels name the away team; NO on the other side; also the untyped "Home vs. Away" replications), soccer's 1X2 (a negRisk `group` of one YES/NO market per outcome titled after the team or "Draw": only the YES tokens, as every 3-way moneyline in the model), the total-goals props (`binaryMarketType: total_goals`, "3 or more goals" = Over 2.5 YES / Under NO) and the spread props (`spreads`, "X to win by 2 or more" = a ±1.5 line in the home-margin convention). Futures, specials, transfers, "both teams to score", map winners and F1 are skipped (77 of 217 listed on 2026-10-26: 137 games, 311 assets, 4.5 s). Asset ids: `<marketSlug>:yes` / `:no`; each asset keeps its token id and the market its venue (`exchange`, `adapter` for negRisk, `feeModule`), `takerDelayMs` and `minSize`. Cached in Redis (`Arbitrage.Exchange.Limitless.Games`).

- **Prices** (`LimitlessFeed`): the whole book of every market of an enabled game from `GET /markets/{slug}/orderbook` (public, one call per market, 100 ms apart) when the game is enabled and every `pollSeconds` (5), and from the Socket.IO feed (`LimitlessRealtime`: a minimal Engine.IO v4 client over a raw WebSocket on `wss://ws.limitless.exchange/socket.io/?EIO=4&transport=websocket`, namespace `/markets`, answering the pings; `subscribe_market_prices {marketSlugs}` replaces the subscription, so the full set of enabled slugs is re-sent on every change and after a reconnect, and the server answers with every book; `orderbookUpdate` is the whole book with a `version`, frames older than the one held are dropped) while it is connected, the poll then only re-seeding every 60 s. For `:yes` the ask is the best ask and the bid the best bid; for `:no` the ask is 1 − best bid and the bid 1 − best ask; levels under `limitless:minSize` shares (1) are skipped. The taker fee of the current venues (`feeModuleType: COLLATERAL`: 5% × (1 − price) of the USDC value, paid in USDC on top; makers pay nothing; `limitless:takerFee`) is folded into both so Limitless's prices compare with the other exchanges': ask = p (1 + c (1 − p)), bid = p (1 − c (1 − p)).
- **Trading** (`LimitlessTrading`): a buy is a taker order on the asset's token (`POST /orders`, which waits for the settlement): the limit (fee removed, inverse of the fold, 3 decimals rounded down) and the shares the budget buys, in multiples of 0.001 share so `price × shares` is an exact integer in base units (makerAmount = USDC, takerAmount = shares; the reverse for a sell); FAK fills what the book has and cancels the rest, FOK is a market order for the whole budget (`makerAmount` = the budget, `takerAmount` = 1), GTD is a GTC that the service cancels (`DELETE /orders/{id}`) after `expirationSeconds`, since a signed expiration is rejected. The 12-field CTF Exchange `Order` (domain `Limitless CTF Exchange` v1 on 8453, verifying contract = the market's `venue.exchange`; `expiration` and `nonce` 0, `feeRateBps` 0 on COLLATERAL venues else the profile's `effectiveFeeRateBps`, a `FEE_RATE_MISMATCH` answer re-signs once with the expected rate; `signatureType` 0) is signed with Nethereum; `dotnet run -- --selftest` prints a fixed order's signature, equal to eth_account's `encode_typed_data` on 2026-10-26. The answer's `execution.settlementStatus`: `MATCHED` / `MINED` / `CONFIRMED` = Accepted with `totalsRaw` (shares = `contractsNet`, cost = `usdGross`

+ `usdFee`), `UNMATCHED` = resting for a GTC, else Declined, `CANCELED` / `FAILED` = Declined unless something filled, `DELAYED` (a marketable order on a sports market is held `settings.takerDelayMs`, 250–1,250 ms, and matched against the book as it stands then) and `RETRYING` = Delayed; a 409 (the client order id was already submitted) = Delayed, a transport error = Delayed. The engine's client order id is sent as Limitless's `clientOrderId` (≤ 128 chars). Orders in Mongo `LimitlessOrders` (keyed by the engine's client order id before they are sent, so retries are idempotent), resolved markets in `LimitlessMarkets`. The balance behind `/health` is the EOA's USDC on Base read with `eth_call balanceOf` through `limitless:rpcUrl` (`https://mainnet.base.org`), cached 30 s.
- **Orders**: `/orders?since=` refreshes our open orders from `POST /orders/status/batch` (30 per call; the cumulative `orderExecution`), and lists the account's other CLOB fills from `GET /portfolio/history` (entries whose `strategy` ends in Buy / Sell and whose `orderId` is not ours; `outcomeIndex` 0 = YES). A GTC with nothing matched is `Resting` until its expiry, a FAK/FOK with nothing is `Delayed` for 3 min, then `Unfilled`. `/markets` asks `GET /markets/{slug}`: `RESOLVED` with `winningOutcomeIndex` (0 = YES wins) or `payoutNumerators` (a 50/50 void) → the payout; `LOCKED` / expired → Closed; `FUNDED` → Open with the feed's ask (or `prices` folded).
- **Config** (`config/appsettings.json`, same `ConnectionStrings` and `environment` as the other services; environment variables like `limitless__apiKey` and `--limitless:trading=true` override it): `limitless {apiKey, secret, privateKey, pages[] (/sport), daysAhead (3), pollSeconds (5), takerFee (0.05), minSize (1, shares), trading (false), maxPerTrade (10), port (8080), serviceKey (our X-Api-Key for /trade and /orders), discoveryMinutes (10), rpcUrl}`. Image: `aut.bat` → `nexus.home/limitless-exchange`.
- **Verified on 2026-10-26** without a token: discovery (137 games), `/availability`, `/subscription`, `/markets`, the order-book poll, the Socket.IO feed (books for 3 enabled games, every update applied) and the signing (equal to eth_account's). The signed routes (`/profiles/me`, `/orders`, `/portfolio/history`, order placement) have not been run: they need the token (and a funded EOA for orders).
- **Monitor page** at `/`: the same layout as the other monitors in a dark theme with a lime accent (lime bids, rose asks, Inter and JetBrains Mono from Google Fonts with system fallbacks): cards for the feed, the realtime connection (subscribed markets, events applied / stale), NATS out, the catalog (pages, markets listed / not mapped), the Limitless API (calls, errors, 429s), the account (balance, taker fee, profile, why trading is blocked) and the process; the availability list with Enable/Disable per game and each game's markets with the raw book level, its size, bid/ask, the taker delay and a link to the market; the last orders sent.

Novig (`Arbitrage/NovigExchange`)

Novig's exchange API v3 (`https://api.novig.com`, or the play-money exchange `https://api.paper.novig.com` with `novig:paper`; docs at docs.novig.com), port 8080 (`novig:port`), publisher `Novig`. Paper only in the engine (the `@novig` strategy copies); its own `/trade` is off unless `novig:trading` is true and the key is a trading key. Novig is a US peer-to-peer sports exchange: every market has two outcomes, each with its own ladder of **bids** (an order always buys an outcome), a price is the outcome's probability on a grid (0.001 steps up to 0.050 and from 0.950,

0.005 between) and one contract pays 1¢, so 100 contracts are one share of the model. Takers pay `c × P (1 − P)` per contract (`fee.coefficient`, 0.06 on every game market) but only while the event is in play (`charged = WHEN_LIVE`); makers pay nothing.

- **Keys** (`novig:keyId` + `novig:privateKey` or `privateKeyPath`, a PKCS#8 PEM, Ed25519 or P-256): every request is signed (`NOVIG-V3`: six LF-joined lines `NOVIG-V3`, Unix ms, METHOD, path, canonical query, lowercase SHA-256 of the body; headers `Novig-Key-Id`, `Novig-Timestamp`, `Novig-Signature`), ±30 s of the server's clock (`GET /v3/time` measures the skew). Two kinds: the **management** key from Profile › Settings › Novig API reads the catalog, the balance and every wallet's orders and fills but can never place an order or open the socket; a subaccount's **trading** key does both. The service tells them apart at startup (`GET /v3/account/subaccounts/{keyId}/balance` answers 400 for a management key) and `POST /v3/echo` checks the signature. `dotnet run -- --echo` prints the result; `dotnet run -- --open-subaccount <label>` generates an Ed25519 keypair, opens a subaccount with it (`POST /v3/account/subaccounts`, management key) and prints the trading key id and PEM to put in the config; `dotnet run -- --selftest [signing-vectors.json]` checks the signer against Novig's published vectors (all 30 pass on 2026-10-26). Without a key the catalog and the prices come from the public mirror (`/v3/public/catalog/…`, cached at the edge): no realtime, no account. Every signed call spends a server-side token bucket (read 64 / 16 Hz, account 64 / 8 Hz, history 512 / 4 Hz, place 256 / 8 Hz, cancel 256 / 16 Hz, stream 512 / 32 Hz; `GET /v3/limits`); the client models them (`Bucket`) and queues a call that would empty one, a 429 is retried after `Retry-After`.
- **Catalog** (every `novig:discoveryMinutes`, 10): `GET /v3/catalog/events` (statuses OPEN_PREGAME, OPEN_INGAME, DELAYED; kick-offs from 8 h ago to `daysAhead` days ahead; `novig:leagues` allowlist, else every league), `GET /v3/catalog/participants` (home/away competitor ids with name, short name and symbol; signed routes only) and `GET /v3/catalog/markets` filtered to the game-line types, paged at 2,000. `GameId` = the event id. Mapped: `MONEY` (2-way; not in soccer, where it is draw-no-bet), `SPREAD` (`strike` = the home side's handicap: the outcome named with it is Over = home covers), `TOTAL`, `TEAM_TOTAL` (the team from the description), `MONEY_1H` / `SPREAD_1H` / `TOTAL_1H` as the first half, and soccer's `MONEYLINE_3_WAY_WIN` per team and `MONEYLINE_3_WAY_DRAW` (their Yes outcome only: No is "any other result"). Half lines only (a whole line pushes). Asset ids `<marketId>:<outcomeId>`. Without participants the sides come from the event description ("Away @ Home", a tennis round sometimes appended) and the spread outcomes' names ("WSH +5.5" with strike 5.5 names the home side), and outcome names match by symbol, short name, full name or "M. Tobon" for "Miguel Tobon". Cached in Redis (`Arbitrage.Exchange.Novig.Games`). On 2026-10-26 (production, no key): 237 games, 7,591 assets from 3,911 markets of 237 events in the 3-day window (ATP/WTA, NBA, NHL, MLS, Serie A, NCAAF, NFL…).
- **Prices**: for an asset the bid is its outcome's best bid and the ask is 1 − the other outcome's best bid (a level under `novig:minQty` contracts, 100 = $1, counts as empty). The fee is folded in while the event is in play (ask = p + c p (1 − p), bid = p − c p (1 − p)) so Novig compares with the other exchanges; pregame the prices are the book's. Top of book from `GET /v3/catalog/markets?event=` (every market of the game with each outcome's best `bid`, one read token) when a game is enabled and every `pollSeconds` (5),

and from the WebSocket (`GET /v3/ws`, the upgrade signed like any route, trading key only) on the `bbo` channel, one subscription per mapped market (8 stream tokens each, sent in chunks of 48 as the bucket allows; a seq gap asks a `snapshot`), the poll then only re-seeding every 60 s. `GET /v3/catalog/events?status=OPEN_INGAME` every 15 s (and the lifecycle transitions GOLIVE / UNLIVE on the socket) says which enabled games charge the fee; a change republishes the game.

- **Trading** (off by default): a buy is `POST /v3/orders {outcomeId, price, qty, tif, ttl?, clientId}` with the limit (fee removed while live) snapped **down** to the grid and the contracts the budget buys at that price plus the fee; FAK = `IOC`, FOK = `FOK`, GTD = `GTT` with the expiration as `ttl` (Novig expires it). Selling is not supported (close by buying the other outcome). Novig's 201 means accepted and queued, not filled: the outcome comes from the private `orders` channel (`open` / `fill` / `cancel` / `reject`; an IOC that matches nothing is a `reject`) and from `GET /v3/orders/{id}`, polled for 4 s before `/trade` answers (Accepted with the fill, Declined on reject / cancel with nothing matched, Delayed otherwise). `clientId` is only a label (a replay places again), so the order is saved in Mongo `NovigOrders` before it is sent and a retry of the same client order id is `Declined: duplicate` ; the uuid sent is derived from the engine's id. `/orders?since=` refreshes our orders (`GET /v3/orders/{id}` + `GET /v3/portfolio/fills?order=` for the price and fee of each fill; shares = contracts / 100, average price = dollars paid with the fee per share) and lists the subaccount's other fills (`GET /v3/portfolio/fills?startsAfter=`) one record per order. `/markets` asks `GET /v3/history/markets/{id}`: `SETTLED` → the outcome's grade (WIN 1, LOSS 0, PUSH 0.5 = the model's void, an FMV grade = that price), kept in Mongo `NovigMarkets`; `CLOSED` → Closed; open markets carry the feed's ask.
- **Config** (`config/appsettings.json`, same `ConnectionStrings` and `environment` as the other services; environment variables like `novig__keyId` and `--novig:trading=true` override it): `novig {keyId, privateKey | privateKeyPath, paper (false), leagues[], daysAhead (3), pollSeconds (5), minQty (100 contracts), trading (false), maxPerTrade (10, dollars), port (8080), serviceKey (our X-Api-Key for /trade and /orders), discoveryMinutes (10), realtime (true)}`. Image: `aut.bat` → `nexus.home/novig-exchange`.
- **Verified on 2026-10-26** without a key (public mirror): discovery, `/availability`, `/subscription`, the poll (an in-play NBA game: 153 of 210 assets quoted with the fee folded, a pregame ATP match with none) and the signer (37 of 37 vectors). The signed routes, the socket, the private channel and order placement have not been run: they need the key (orders: a funded subaccount's trading key; try Paper first).
- **Monitor page** at `/`: the same layout as the other monitors in a navy theme with a Novig-green accent (green bids, coral asks, Inter and JetBrains Mono): cards for the feed (poll, live check), the realtime connection (subscribed markets, messages applied / gaps, stream tokens), NATS out, the catalog (games, in play, markets listed / not mapped, whether the sides came from participants), the Novig API (calls, errors, 429s, bucket tokens, key kind and type, clock skew), the account (balance, exchange status, fee, why trading is blocked) and the process; the availability list (filters: enabled, in play, today, league) with Enable/Disable per game and each game's markets with both outcomes' best bids and sizes, bid/ask and the Novig market; the last orders sent.

Pinnacle (`Arbitrage/PinnacleExchange`)

Pinnacle's official REST API (`https://api.pinnacle.com`, or `pinnacle:baseUrl` for a white-label such as ps3838), port 8093 (`pinnacle:port`), publisher `Pinnacle`. Paper only in the engine (the `@pinnacle` strategy copies); its own `/trade` is off unless `pinnacle:trading` is true. Pinnacle closed new API access to the public in July 2025: this needs an account that already has it (HTTP Basic auth, `pinnacle:username` / `password`). A 401/403 stops every call until a restart (repeated failed logins lock the account). Pinnacle is a bookmaker, not an order book: one price per outcome, its margin included, and a bet is taken whole at the current line or not at all.

- **Catalog** (`GET /v3/fixtures` per sport every `discoveryMinutes`, 10; `sports` = soccer, tennis, basketball, football, hockey, baseball by default (also mma, esports), optional `leagues` allowlist of league ids, `daysAhead` 2): `GameId` = the pregame event id; a live copy (an event whose `parentId` is that id) is folded into the same game. Only `resultingUnit` Regular (corners, bookings, set/game handicap events are left out). Home/away as Pinnacle names them. Cached in Redis (`Arbitrage.Exchange.Pinnacle.Games`).
- **Prices** (`GET /v3/odds?oddsFormat=Decimal` per sport while any of its games is enabled, every `pollSeconds`, 5 at least): a full snapshot first and every 10 min, otherwise the changes since Pinnacle's `last` cursor. Each period sent replaces what we had for it; a line it no longer lists or a period offline (status 2) is dropped and not published. Periods: 0 = Game; soccer 1 = 1st half; basketball/football 1–2 halves, 3–6 quarters; tennis 1–3 sets; others game only. Markets: moneyline (soccer's draw = `Draw`), spreads (`hdp` = home's handicap → Over/Under at home's line), totals, team totals; half lines only, alternate lines included. `price` = 1 / odds; `bestBid` = 1 − the other side's price on two-way markets (betting the other side is how a position is sold), none on a 3-way moneyline. No fee on top. Asset ids `<eventId>:<period>:<ml|s|t|tt>:<side>[:<line>]` (e.g. `1612345678:0:s:away:-1.5`, line = home's handicap on both sides of a spread), so a bet needs only the id and the catalog.
- **Trading** (off by default): `GET /v2/line` first (price, `lineId`, stake limits); the bet is placed (`POST /v2/bets/straight`, RISK stake, `acceptBetterLine`) only if 1 / odds is at or under the limit. FAK cuts the stake to the line's maximum, FOK and GTD decline over it (nothing rests on a book). In play Pinnacle holds a bet a few seconds (`PENDING_ACCEPTANCE`): answered `Delayed`. An accepted bet answers `filledShares` = stake × odds. Selling is not possible. Orders in Mongo `PinnacleOrders` keyed by the client order id, with `uniqueRequestId` a GUID derived from it (Pinnacle's idempotency key). `/orders` reads `GET /v3/bets?betlist=ALL` (30-day windows): shares = stake × odds, payout per share = (risk + winLoss) / shares. `/markets` resolves from `GET /v3/fixtures/settled` (per sport every 5 min, Pinnacle keeps 24 h; Mongo `PinnacleSettled`): the period's score decides the asset, a cancelled period or a tie on a two-way moneyline is void (0.5).
- **Config** `pinnacle:{username, password, baseUrl, sports[], leagues[], daysAhead, pollSeconds, trading, maxPerTrade, port, apiKey (our X-Api-Key for /trade and /orders), discoveryMinutes}`; `config/` is gitignored.
- **Monitor page** (Pinnacle navy and orange): Feed, NATS out, Catalog, Pinnacle API, Account, Process cards; games with enable/disable and their lines (odds, bid, ask, max stake).