# NanoTick OMS — Design & Implementation

A standalone, loosely-coupled, policy-based Order Management module for the nanotick C++17 engine

**Prepared:** 2026-07-08 · **Module:** `NTS::oms` · **Scope:** order request/response lifecycle (pre-trade risk deferred)

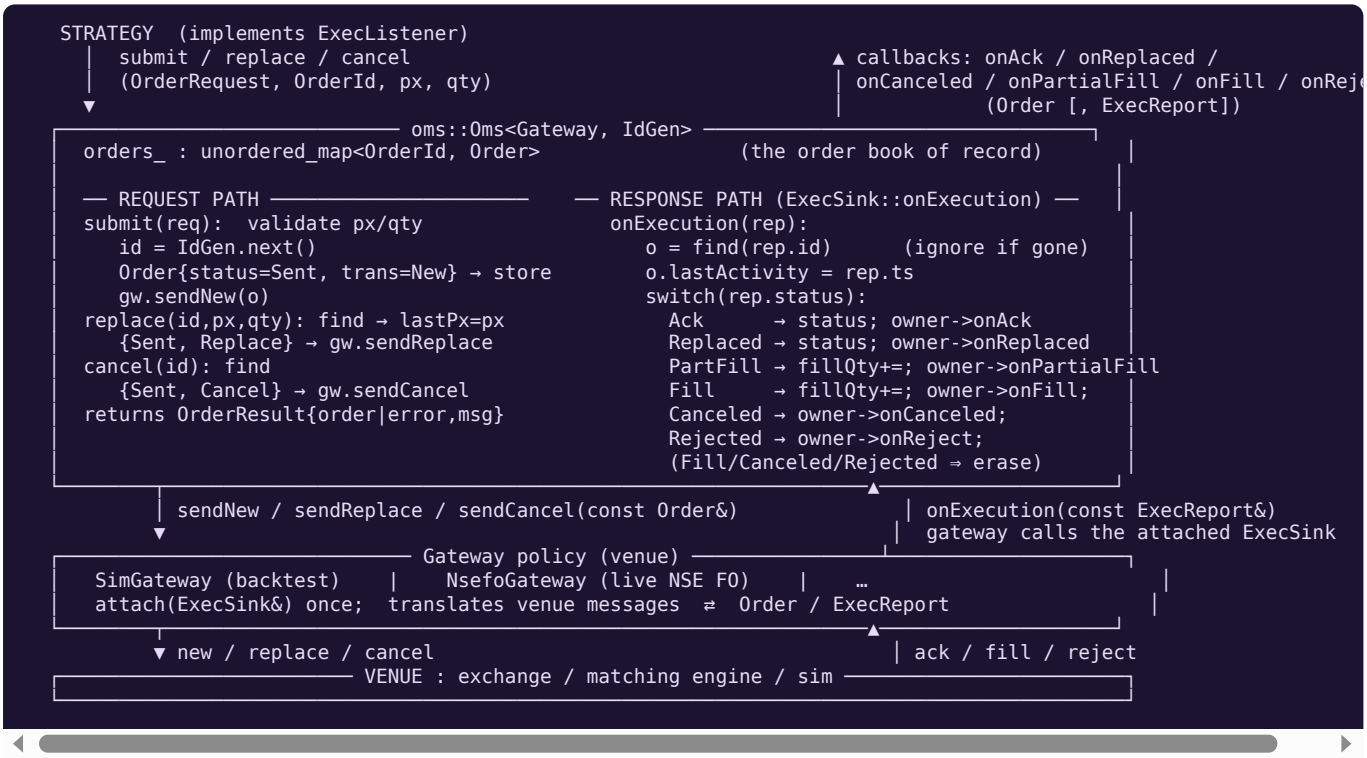
## 1. Overview & goals

`NTS::oms::Oms` is the order half of the trading stack: it accepts order intents from a strategy, tracks live order state, drives a venue gateway, and reports every execution event back to the strategy. It is designed to be:

- **Standalone** — a top-level `oms/` module depending only on `common/` (base types) and `core/` (time). It never references the feed, the trade engine, or any concrete strategy.
- **Loosely coupled** — it touches the outside world through exactly **two** narrow interfaces: `ExecSink` (gateway→OMS, one method) and `ExecListener` (OMS→strategy, six methods).
- **Policy-based** — composed at compile time (same idiom as `trd::TradeEngine<…>`): the **Gateway** (venue) and **IdGen** (order-id) are template policies; backtest vs live is a type swap, monomorphic hot path, no vtable per request.
- **Single-writer** — one `Oms` instance per engine thread; its order store needs no locking.

> **Deferred:** pre-trade risk (RMS) is intentionally out of scope. It re-attaches later as an optional `Risk` policy — a single `check(Order)` at the top of `submit()` — without touching anything else.

## 2. Comprehensive flow (request & response)




**Two directions, two contracts.** Down the stack the OMS calls the **Gateway** policy; up the stack the gateway calls back through the **ExecSink** it was handed (implemented by the OMS), and the OMS forwards to the per-order **ExecListener** (the strategy). Nothing here knows about the trade engine.

## 3. Module layout

```
include/oms/
  OmsTypes.h        // enums + POD entities (OrderRequest, Order, ExecReport, OrderResult)
  ExecSink.h        // gateway → OMS  (1 method)
  ExecListener.h    // OMS → strategy (6 callbacks)
  Oms.h             // the policy template (header-only)
  policy/SeqOrderId.h                   // default id policy
  gw/SimGateway.h   (+ src/oms/gw/SimGateway.cpp for any non-template bodies)
  gw/NsefoGateway.h (+ .cpp)            // live NSE FO (later)
```

## 4. Entity reference

### 4.1 Request-side entities

`OrderRequest` — the strategy's immutable *intent*, the input to `submit()`.

| Field | Type | Meaning |
|---|---|---|
| `instr` | InstrId | Instrument token. |
| `side` | OrderSide | BUY / SELL. |
| `px` | Price | Limit price (ticks). Must be > 0. |
| `qty` | Qty | Order quantity. Must be > 0. |
| `owner` | ExecListener* | Who receives this order's callbacks (usually the submitting strategy). |

`Order` — the OMS-owned *mutable state*, the record kept in `orders_`.

| Field | Type | Meaning |
|---|---|---|
| `id` | OrderId | OMS-assigned unique id (from IdGen). |
| `exchId` | ExchOrderId | Exchange order id (filled from the first ack). |
| `instr / side` | InstrId / OrderSide | Copied from the request. |
| `px` | Price | Current working price. |
| `lastPx` | Price | Price before the last replace. |
| `size` | Qty | Current working quantity. |
| `origSize` | Qty | Quantity at submit. |
| `fillQty` | Qty | Cumulative filled quantity. `leaves() = size − fillQty`. |
| `status` | OrderStatus | Lifecycle state (see §4.4). |
| `trans` | TransType | Last request kind sent (New / Replace / Cancel). |
| `lastUpdate` | Timestamp | When the last request was sent (µs). |
| `lastActivity` | Timestamp | When the last venue event arrived (µs). |
| `owner` | ExecListener* | Callback target for this order. |

### 4.2 Response-side entities

`ExecReport` — a single venue event, gateway→OMS via `ExecSink::onExecution`.

| Field | Type | Meaning |
|---|---|---|
| `id` | OrderId | Which OMS order this event is for. |
| `exchId` | ExchOrderId | Exchange id (0 if not applicable). |
| `status` | OrderStatus | Ack / Replaced / Canceled / PartFill / Fill / Rejected. |
| `execPx` | Price | Fill price (fills only). |
| `execQty` | Qty | Fill quantity this event (fills only). |
| `rej` | RejectReason | Reject reason (rejects only). |
| `passive` | bool | True if this fill was passive (maker). |
| `ts` | Timestamp | Event time (µs). |

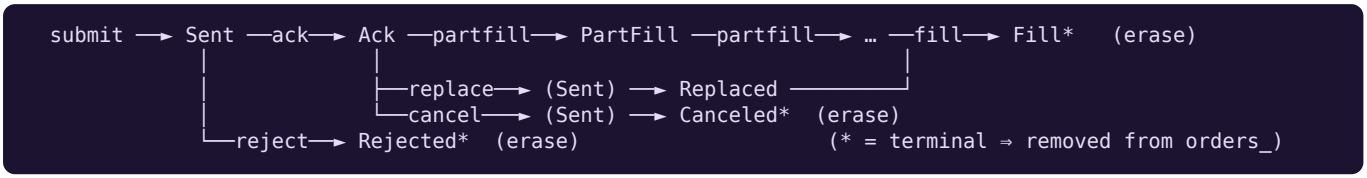
`OrderResult` — the synchronous return of `submit / replace / cancel`.

| Field | Type | Meaning |
|---|---|---|
| `order` | Order* | Pointer to the live order (nullptr on error). |
| `error` | bool | True if the request was rejected locally. |
| `msg` | const char* | Human-readable reason on error. |

### 4.3 Enums

| | |
|---|---|
| **`TransType`** | None, New, Replace, Cancel |
| **`OrderStatus`** | None, Sent, Ack, Replaced, Canceled, PartFill, Fill, Rejected |
| **`RejectReason`** | None, Dpr, InvalidPrice, InvalidQty, OrderChanged, NotFound, BelowMinSize, Unknown |

### 4.4 Order lifecycle (state machine)



## 5. Policy seams (contracts)

| Seam | Direction | Contract | Default / impl |
|---|---|---|---|
| `Gateway` | OMS → venue | `attach(ExecSink&)`, `sendNew/sendReplace/sendCancel(const Order&)→bool`, `poll()` | SimGateway; NsefoGateway (live) |
| `ExecSink` | gateway → OMS | `onExecution(const ExecReport&)` | Oms implements it |
| `ExecListener` | OMS → strategy | `onAck/onReplaced/onCanceled/onPartialFill/onFill/onReject` | strategy implements |
| `IdGen` | OMS internal | `next() → OrderId` | SeqOrderId |

## 6. C++ implementation

### 6.1 `oms/OmsTypes.h`

```
#pragma once
#include <cstdint>
#include "common/Types.h"   // InstrId, Price, Qty(Size), Timestamp, OrderSide

namespace NTS { namespace oms {

using OrderId     = uint64_t;
using ExchOrderId = uint64_t;

enum class TransType    : uint8_t { None, New, Replace, Cancel };
enum class OrderStatus  : uint8_t { None, Sent, Ack, Replaced, Canceled, PartFill, Fill, Rejected };
enum class RejectReason : uint8_t { None, Dpr, InvalidPrice, InvalidQty, OrderChanged, NotFound, BelowMinSize

inline bool isTerminal(OrderStatus s) {
    return s == OrderStatus::Canceled || s == OrderStatus::Fill || s == OrderStatus::Rejected;
}

class ExecListener;   // fwd

// Strategy intent — immutable input to submit().
struct OrderRequest {
    InstrId       instr = 0;
    OrderSide     side  = ORD_SIDE_BUY;
    Price         px    = 0;
    Qty           qty   = 0;
    ExecListener* owner = nullptr;
};

// OMS-owned mutable order state.
struct Order {
    OrderId       id           = 0;
    ExchOrderId   exchId       = 0;
    InstrId       instr        = 0;
    OrderSide     side         = ORD_SIDE_BUY;
    Price         px           = 0;   // working price
    Price         lastPx       = 0;   // price before last replace
    Qty           size         = 0;   // working qty
    Qty           origSize     = 0;   // qty at submit
    Qty           fillQty      = 0;   // cumulative filled
    OrderStatus   status       = OrderStatus::None;
    TransType     trans        = TransType::None;
    Timestamp     lastUpdate   = 0;
    Timestamp     lastActivity = 0;
    ExecListener* owner        = nullptr;
    Qty leaves() const { return size > fillQty ? size - fillQty : 0; }
};

// One venue event — gateway → OMS.
struct ExecReport {
    OrderId      id      = 0;
    ExchOrderId  exchId  = 0;
    OrderStatus  status  = OrderStatus::None;
    Price        execPx  = 0;
    Qty          execQty = 0;
    RejectReason rej     = RejectReason::None;
    bool         passive = false;
    Timestamp    ts      = 0;
};

// Synchronous result of submit/replace/cancel.
struct OrderResult {
    Order*      order = nullptr;
    bool        error = false;
    const char* msg   = "";
    static OrderResult ok(Order* o)        { return { o, false, "" }; }
    static OrderResult fail(const char* m) { return { nullptr, true, m }; }
};
```

```
}} // namespace NTS::oms
```

### 6.2 `oms/ExecSink.h` & `oms/ExecListener.h`

```
// ExecSink.h — the ONLY thing a gateway knows about the OMS.
#pragma once
#include "oms/OmsTypes.h"
namespace NTS { namespace oms {
class ExecSink {
 public:
    virtual ~ExecSink() = default;
    virtual void onExecution(const ExecReport&) = 0;
};
}}

// ExecListener.h — implemented by the consumer (strategy). Override what you need.
#pragma once
#include "oms/OmsTypes.h"
namespace NTS { namespace oms {
class ExecListener {
 public:
    virtual ~ExecListener() = default;
    virtual void onAck        (const Order&) {}
    virtual void onReplaced   (const Order&) {}
    virtual void onCanceled   (const Order&) {}
    virtual void onPartialFill(const Order&, const ExecReport&) {}
    virtual void onFill       (const Order&, const ExecReport&) {}
    virtual void onReject     (const Order&, const ExecReport&) {}
};
}}
```

### 6.3 `oms/policy/SeqOrderId.h`

```
#pragma once
#include "oms/OmsTypes.h"
namespace NTS { namespace oms {
class SeqOrderId {
 public:
    explicit SeqOrderId(OrderId start = 10000) : next_(start) {}
    OrderId next() { return ++next_; }
 private:
    OrderId next_;
};
}}
```

### 6.4 `oms/Oms.h` — core (header-only template)

```
#pragma once
#include <unordered_map>
#include "oms/OmsTypes.h"
#include "oms/ExecSink.h"
#include "oms/ExecListener.h"
#include "oms/policy/SeqOrderId.h"
#include "core/time/TimeUtil.h"   // NTS::core::time::nowUs()

namespace NTS { namespace oms {

// Gateway policy (duck-typed):
//   void attach(ExecSink&);
//   bool sendNew(const Order&); bool sendReplace(const Order&); bool sendCancel(const Order&);
//   void poll();
template<class Gateway, class IdGen = SeqOrderId>
class Oms : public ExecSink {
 public:
    Oms(Gateway& gw, IdGen& ids) : gw_(gw), ids_(ids) { gw_.attach(*this); }

    // ---- request path ----
    OrderResult submit(const OrderRequest& r) {
        if (r.px  == 0) return OrderResult::fail("submit: invalid price");
        if (r.qty == 0) return OrderResult::fail("submit: invalid qty");
        const OrderId id = ids_.next();
        if (orders_.count(id)) return OrderResult::fail("submit: duplicate id");

        Order& o = orders_[id];
        o.id = id; o.instr = r.instr; o.side = r.side;
        o.px = r.px; o.size = r.qty; o.origSize = r.qty; o.owner = r.owner;
        o.status = OrderStatus::Sent; o.trans = TransType::New;
        o.lastUpdate = core::time::nowUs();

        if (!gw_.sendNew(o)) { orders_.erase(id); return OrderResult::fail("submit: gateway send failed"); }
        return OrderResult::ok(&o);
    }

    OrderResult replace(OrderId id, Price newPx, Qty newQty) {
        auto it = orders_.find(id);
        if (it == orders_.end()) return OrderResult::fail("replace: not found");
        Order& o = it->second;
        o.lastPx = o.px; o.px = newPx; o.size = newQty; o.origSize = newQty;
        o.status = OrderStatus::Sent; o.trans = TransType::Replace;
        o.lastUpdate = core::time::nowUs();
        if (!gw_.sendReplace(o)) return OrderResult::fail("replace: gateway send failed");
        return OrderResult::ok(&o);
    }

    OrderResult cancel(OrderId id) {
        auto it = orders_.find(id);
        if (it == orders_.end()) return OrderResult::fail("cancel: not found");
        Order& o = it->second;
        o.status = OrderStatus::Sent; o.trans = TransType::Cancel;
        o.lastUpdate = core::time::nowUs();
        if (!gw_.sendCancel(o)) return OrderResult::fail("cancel: gateway send failed");
        return OrderResult::ok(&o);
    }

    // ---- response path: single fan-in from the gateway ----
    void onExecution(const ExecReport& e) override {
        auto it = orders_.find(e.id);
        if (it == orders_.end()) return;              // unknown or already terminal
        Order& o = it->second;
        o.lastActivity = e.ts;
        if (e.exchId) o.exchId = e.exchId;
        ExecListener* L = o.owner;

        switch (e.status) {
            case OrderStatus::Ack:      o.status = e.status; if (L) L->onAck(o);         break;
            case OrderStatus::Replaced: o.status = e.status; if (L) L->onReplaced(o);    break;
            case OrderStatus::PartFill: o.fillQty += e.execQty; o.status = e.status;
                                        if (L) L->onPartialFill(o, e);              break;
            case OrderStatus::Fill:     o.fillQty += e.execQty; o.status = e.status;
                                        if (L) L->onFill(o, e); orders_.erase(it);  break;
```

```
            case OrderStatus::Canceled: o.status = e.status;
                                        if (L) L->onCanceled(o); orders_.erase(it); break;
            case OrderStatus::Rejected: o.status = e.status;
                                        if (L) L->onReject(o, e); orders_.erase(it); break;
            default: break;                             // None/Sent not expected inbound
        }
    }

    void poll() { gw_.poll(); }
    const Order* find(OrderId id) const { auto it = orders_.find(id); return it == orders_.end() ? nullptr :
    std::size_t liveCount() const { return orders_.size(); }

 private:
    Gateway& gw_;
    IdGen&   ids_;
    std::unordered_map<OrderId, Order> orders_;   // order book of record (swap for object-pool later)
};

}} // namespace NTS::oms
```

### 6.5 `oms/gw/SimGateway.h` — backtest gateway (reference)

```
#pragma once
#include "oms/OmsTypes.h"
#include "oms/ExecSink.h"
#include "core/time/TimeUtil.h"
namespace NTS { namespace oms {

// Minimal backtest gateway: ACKs immediately; the sim/matching harness calls
// fill() to report (partial/full) fills. Satisfies the Gateway contract.
class SimGateway {
 public:
    void attach(ExecSink& s)        { sink_ = &s; }
    bool sendNew(const Order& o)     { emit(o.id, OrderStatus::Ack);      return true; }
    bool sendReplace(const Order& o) { emit(o.id, OrderStatus::Replaced); return true; }
    bool sendCancel(const Order& o)  { emit(o.id, OrderStatus::Canceled); return true; }
    void poll() {}                  // event-driven sim; nothing to pump

    void fill(OrderId id, Price px, Qty qty, bool full, bool passive = true) {
        ExecReport e; e.id = id; e.status = full ? OrderStatus::Fill : OrderStatus::PartFill;
        e.execPx = px; e.execQty = qty; e.passive = passive; e.ts = core::time::nowUs();
        if (sink_) sink_->onExecution(e);
    }
 private:
    void emit(OrderId id, OrderStatus s) {
        ExecReport e; e.id = id; e.status = s; e.ts = core::time::nowUs();
        if (sink_) sink_->onExecution(e);
    }
    ExecSink* sink_ = nullptr;
};
}}
```

### 6.6 Usage — a strategy wiring the OMS

```
class MyStrategy : public oms::ExecListener {
 public:
    explicit MyStrategy(oms::Oms<oms::SimGateway>& o) : oms_(o) {}

    void quote(InstrId instr, Price px, Qty qty) {
        oms::OrderRequest r; r.instr = instr; r.side = ORD_SIDE_BUY;
        r.px = px; r.qty = qty; r.owner = this;
        auto res = oms_.submit(r);
        if (res.error) { /* handle local reject: res.msg */ }
        else           workingId_ = res.order->id;
    }
    void onAck (const oms::Order& o) override { /* order live at o.px */ }
    void onFill(const oms::Order& o, const oms::ExecReport& e) override { /* position += e.execQty */ }
    void onReject(const oms::Order&, const oms::ExecReport& e) override { /* e.rej */ }
 private:
    oms::Oms<oms::SimGateway>& oms_;
    oms::OrderId workingId_ = 0;
};

// wiring
oms::SimGateway  sim;
oms::SeqOrderId  ids;
oms::Oms<oms::SimGateway> oms(sim, ids);   // backtest
MyStrategy strat(oms);
// live: swap the type — oms::Oms<oms::NsefoGateway, oms::ExchNonceId> oms(gw, ids);
```

## 7. Threading & conventions

- **One OMS per engine thread** — single writer to `orders_`; **no locking**. Fits the sharded/N:M engine model directly, and works unchanged single-threaded in replay.
- **C++17, no Boost** (std only). `Oms` is a template ⇒ header-only; non-template gateway bodies live in `src/oms/**.cpp`.
- **POD entities**, trivially copyable; no hot-path allocation beyond the map insert (swap the store for an object-pool + intrusive index later for zero-alloc).
- **No god-object**: the OMS holds only `Gateway&`, `IdGen&`, and its order map; the outside world is reached only via `ExecSink` / `ExecListener`.

## 8. Open decisions

- **Order store:** `unordered_map` (simple, shown) vs object-pool + intrusive index (zero-alloc, HFT-grade). Recommend map first.
- **Multi-strategy routing:** per-order `ExecListener*` (shown — supports N strategies per OMS) vs a single OMS-level listener.
- **Replace semantics:** amend-in-place (keep id — shown) vs cancel/replace (new id). Match the venue (NSE FO supports modify).
- **Local validations** (sec-ban / DPR / min-size): a thin check in `submit()` now; fold into the optional `Risk` policy when RMS is added.
- **Out of scope:** the matching/sim behind `SimGateway` (queue model, latency, fills) — a separate design; the OMS only speaks the Gateway contract.