Reservations Service – API-First Specification (Schema.org-aligned)

1. Overview

1.1 Purpose

This project delivers a standalone **Reservations Service** that supports planning and allocation of fleet equipment for scheduled events and ASAP batch requests.

The service is **API-first**: all functionality is available via HTTP APIs, and side effects are driven via a standardized **Trigger** system. A UI may be added later but is not required for initial value.

1.2 Design goals

- **Interoperability:** Use **Schema.org JSON-LD** objects where appropriate (notably `RentAction`).
- **API-first:** All functionality is accessible via HTTP APIs; UI is optional.
- **Deterministic contracts:** Standardize API endpoints and Trigger event types.
- **Configurable automation:** Trigger actions are configurable per environment (webhooks, transforms, routing, etc.).
- **Reliability:** Trigger delivery is resilient and replayable (Outbox pattern).
- **Planning-aware:** Predict future device needs from known events/reservations, compare to expected supply, and surface or automate procurement/build actions.
- **Operational history:** Maintain a maintenance log per asset, including photo/document uploads.

1.3 Non-goals

- No full refurbishment/lifecycle execution engine (owned by LMT), beyond **planning signals** and **maintenance logging**.
- No deep manufacturing/BOM execution logic (owned by LMT + InvenTree). This service may **request** builds via standardized triggers.
- No remote management logic (owned by LMT + MeshCentral).
- No UI requirements beyond API discoverability (UI is an optional later phase).

---

2. Schema.org alignment

2.1 Primary interoperable object: `RentAction`

A reservation is represented externally as a Schema.org `RentAction` in JSON-LD.

Minimum mapping (API payloads): - `@context`: `https://schema.org` - `@type`: `RentAction` - `identifier`: service reservation ID - `agent`: requester (`Person` or `Organization`) - `object`: the

thing being rented/reserved (`Product` or `IndividualProduct`) - `startTime`, `endTime`: reservation window - `actionStatus`: mapped from internal status - `description` / `purpose`: optional

2.2 Quantities and allocation fields

Schema.org does not standardize "reserve N units" for `RentAction` in a way that fits all use cases. The service therefore uses a small set of **extension fields** alongside JSON-LD: - `requestedQuantity` (integer) - `allocatedQuantity` (integer) - `lineItems` (array; see Section 3.4)

2.3 Operational actions (maintenance)

Maintenance work can optionally be expressed using Schema.org action types (JSON-LD) in event payloads: - `RepairAction` (preferred when applicable) - `UpdateAction` (for software/firmware/config updates)

The service standardizes its own maintenance log schema (Section 3.9+) and includes an optional `schema_org` JSON-LD field for interoperability.

2.4 How catalog entities map

- **Item Type / SKU:** `Product`
- **Serialized asset / unit:** `IndividualProduct` (with `serialNumber` or internal `identifier`) and `isVariantOf` → `Product`
- **Kit template:** `Product` + extension field `kitComponents`

---

3. Core concepts and schema

> NOTE: Types are expressed in SQL-like form for clarity; implement with the chosen framework's migrations.

3.1 Terminology

- **Item Type**: a reservable type/category (e.g., TimerNode v2).
- **Asset**: a serialized physical unit of an item type.
- **Stock Pool**: fungible quantity for un-serialized items.
- **Kit Template**: composition of item types and quantities.
- **RentAction / Reservation**: time-window request for quantities.

3.2 `item_types`

Represents reservable categories (serialized, fungible, or kit).

```
CREATE TABLE item_types (
  id BIGINT PRIMARY KEY,
  code VARCHAR(64) UNIQUE NOT NULL,
  name VARCHAR(191) NOT NULL,
```

```
  kind VARCHAR(32) NOT NULL, -- 'serialized', 'fungible', 'kit'
  is_active BOOLEAN NOT NULL DEFAULT TRUE,
  metadata JSON NULL,
  created_at TIMESTAMP NULL,
  updated_at TIMESTAMP NULL
);
```

3.3 `assets`

Serialized reservable units.

```
CREATE TABLE assets (
  id BIGINT PRIMARY KEY,
  item_type_id BIGINT NOT NULL,
  asset_tag VARCHAR(128) UNIQUE NULL,
  serial_number VARCHAR(128) NULL,
  status VARCHAR(32) NOT NULL DEFAULT 'available',
  location VARCHAR(191) NULL,
  assigned_to VARCHAR(191) NULL,
  mesh_node_id VARCHAR(191) NULL,
  wireguard_hostname VARCHAR(191) NULL,
  metadata JSON NULL,
  created_at TIMESTAMP NULL,
  updated_at TIMESTAMP NULL,
  CONSTRAINT fk_assets_item_type FOREIGN KEY (item_type_id) REFERENCES
item_types(id)
);

CREATE INDEX idx_assets_item_type ON assets(item_type_id);
CREATE INDEX idx_assets_status ON assets(status);
```

3.4 `stock_pools`

Fungible quantities (optionally per location).

```
CREATE TABLE stock_pools (
  id BIGINT PRIMARY KEY,
  item_type_id BIGINT NOT NULL,
  location VARCHAR(191) NULL,
  quantity_total INT NOT NULL DEFAULT 0,
  is_active BOOLEAN NOT NULL DEFAULT TRUE,
  metadata JSON NULL,
  created_at TIMESTAMP NULL,
  updated_at TIMESTAMP NULL,
  CONSTRAINT fk_stock_item_type FOREIGN KEY (item_type_id) REFERENCES
```

```
item_types(id)
);

CREATE INDEX idx_stock_item_type ON stock_pools(item_type_id);
```

3.5 `kit_templates` and `kit_template_items`

Kit templates are compositional bundles.

```
CREATE TABLE kit_templates (
  id BIGINT PRIMARY KEY,
  code VARCHAR(64) UNIQUE NOT NULL,
  name VARCHAR(191) NOT NULL,
  is_active BOOLEAN NOT NULL DEFAULT TRUE,
  metadata JSON NULL,
  created_at TIMESTAMP NULL,
  updated_at TIMESTAMP NULL
);

CREATE TABLE kit_template_items (
  id BIGINT PRIMARY KEY,
  kit_template_id BIGINT NOT NULL,
  component_item_type_id BIGINT NOT NULL,
  quantity_per_kit INT NOT NULL,
  created_at TIMESTAMP NULL,
  updated_at TIMESTAMP NULL,
  CONSTRAINT fk_kti_kit FOREIGN KEY (kit_template_id) REFERENCES
kit_templates(id),
  CONSTRAINT fk_kti_item FOREIGN KEY (component_item_type_id) REFERENCES
item_types(id)
);

CREATE INDEX idx_kti_kit ON kit_template_items(kit_template_id);
```

3.6 `rent_actions` (reservations)

Top-level reservation (Schema.org `RentAction`).

```
CREATE TABLE rent_actions (
  id BIGINT PRIMARY KEY,

  requester_ref VARCHAR(191) NOT NULL, -- can be a user id, email, or service
principal
  created_by_ref VARCHAR(191) NOT NULL,
  approved_by_ref VARCHAR(191) NULL,
```

```
  status VARCHAR(32) NOT NULL,         --
'draft','pending','approved','rejected','cancelled','fulfilled'
  priority VARCHAR(32) NOT NULL DEFAULT 'normal',

  start_time TIMESTAMP NOT NULL,
  end_time TIMESTAMP NOT NULL,
  is_asap BOOLEAN NOT NULL DEFAULT FALSE,

  description TEXT NULL,

  external_source VARCHAR(191) NULL,
  external_ref VARCHAR(191) NULL,

  schema_org JSON NULL,     -- stored JSON-LD representation (optional but
useful)
  metadata JSON NULL,

  approved_at TIMESTAMP NULL,
  rejected_at TIMESTAMP NULL,
  cancelled_at TIMESTAMP NULL,

  created_at TIMESTAMP NULL,
  updated_at TIMESTAMP NULL
);

CREATE INDEX idx_ra_interval ON rent_actions(start_time, end_time);
CREATE INDEX idx_ra_status ON rent_actions(status);
CREATE INDEX idx_ra_external ON rent_actions(external_source, external_ref);
```

3.7 `rent_action_items`

Line items: what is being reserved and how many.

```
CREATE TABLE rent_action_items (
  id BIGINT PRIMARY KEY,
  rent_action_id BIGINT NOT NULL,

  item_kind VARCHAR(32) NOT NULL, -- 'item_type','kit_template'
  item_id BIGINT NOT NULL,

  requested_quantity INT NOT NULL,
  allocated_quantity INT NOT NULL DEFAULT 0,

  notes TEXT NULL,
  metadata JSON NULL,
```

```
  created_at TIMESTAMP NULL,
  updated_at TIMESTAMP NULL,

  CONSTRAINT fk_rai_action FOREIGN KEY (rent_action_id) REFERENCES
rent_actions(id)
);

CREATE INDEX idx_rai_action ON rent_action_items(rent_action_id);
CREATE INDEX idx_rai_kind_id ON rent_action_items(item_kind, item_id);
```

3.8 `rent_action_allocations`

Concrete allocations.

```
CREATE TABLE rent_action_allocations (
  id BIGINT PRIMARY KEY,
  rent_action_item_id BIGINT NOT NULL,

  -- For serialized allocations
  asset_id BIGINT NULL,

  -- For fungible allocations
  stock_pool_id BIGINT NULL,
  quantity INT NOT NULL DEFAULT 1,

  status VARCHAR(32) NOT NULL DEFAULT 'allocated', --
'allocated','in_use','returned','cancelled'
  allocated_from TIMESTAMP NULL,
  allocated_to TIMESTAMP NULL,
  metadata JSON NULL,

  created_at TIMESTAMP NULL,
  updated_at TIMESTAMP NULL,

  CONSTRAINT fk_raa_item FOREIGN KEY (rent_action_item_id) REFERENCES
rent_action_items(id)
);

CREATE INDEX idx_raa_item ON rent_action_allocations(rent_action_item_id);
CREATE INDEX idx_raa_asset ON rent_action_allocations(asset_id);
CREATE INDEX idx_raa_stock ON rent_action_allocations(stock_pool_id);
```

3.9 Demand planning and build requests

3.9.1 `demand_signals`

Represents forecast inputs for future demand. This can be sourced from: - External event systems (e.g., SGL), - Pre-created reservations (`RentAction`), - Manual planning inputs.

```
CREATE TABLE demand_signals (
  id BIGINT PRIMARY KEY,
  source VARCHAR(64) NOT NULL,         -- 'sgl','manual','import',...
  source_ref VARCHAR(191) NULL,        -- event ID, batch ID, etc.

  start_time TIMESTAMP NOT NULL,
  end_time TIMESTAMP NOT NULL,

  item_type_id BIGINT NOT NULL,
  quantity_required INT NOT NULL,

  location VARCHAR(191) NULL,
  priority VARCHAR(32) NOT NULL DEFAULT 'normal',

  status VARCHAR(32) NOT NULL DEFAULT 'active', --
'active','superseded','cancelled'
  metadata JSON NULL,

  created_at TIMESTAMP NULL,
  updated_at TIMESTAMP NULL,

  CONSTRAINT fk_ds_item_type FOREIGN KEY (item_type_id) REFERENCES
item_types(id)
);

CREATE INDEX idx_ds_window ON demand_signals(start_time, end_time);
CREATE INDEX idx_ds_item_type ON demand_signals(item_type_id);
CREATE INDEX idx_ds_source ON demand_signals(source, source_ref);
```

3.9.2 `supply_snapshots`

Optional cached computation of expected supply available for a time window. Useful for reporting and repeatable “why was a build triggered?” audits.

```
CREATE TABLE supply_snapshots (
  id BIGINT PRIMARY KEY,
  as_of_time TIMESTAMP NOT NULL,
  start_time TIMESTAMP NOT NULL,
```

```
  end_time TIMESTAMP NOT NULL,
  item_type_id BIGINT NOT NULL,

  total_capacity INT NOT NULL,
  reserved_capacity INT NOT NULL,
  available_capacity INT NOT NULL,

  metadata JSON NULL,
  created_at TIMESTAMP NULL,

  CONSTRAINT fk_ss_item_type FOREIGN KEY (item_type_id) REFERENCES
item_types(id)
);

CREATE INDEX idx_ss_window ON supply_snapshots(start_time, end_time);
CREATE INDEX idx_ss_item_type ON supply_snapshots(item_type_id);
```

3.9.3 `build_requests`

Represents a request to manufacture/procure additional units due to predicted shortfall.

```
CREATE TABLE build_requests (
  id BIGINT PRIMARY KEY,

  item_type_id BIGINT NOT NULL,
  quantity INT NOT NULL,
  needed_by TIMESTAMP NULL,            -- when supply must exist

  reason VARCHAR(64) NOT NULL,          --
'shortfall','buffer','replacement',...
  status VARCHAR(32) NOT NULL DEFAULT 'requested', --
'requested','submitted','accepted','rejected','fulfilled','cancelled','failed'

  -- linkage back to what triggered it
  trigger_event_id VARCHAR(191) NULL,
  demand_signal_id BIGINT NULL,

  -- external manufacturing system linkage (e.g., InvenTree)
  external_system VARCHAR(64) NULL,     -- 'inventree'
  external_ref VARCHAR(191) NULL,       -- build order ID

  metadata JSON NULL,
  created_at TIMESTAMP NULL,
  updated_at TIMESTAMP NULL,

  CONSTRAINT fk_br_item_type FOREIGN KEY (item_type_id) REFERENCES
```

```
item_types(id)
);

CREATE INDEX idx_br_status ON build_requests(status);
CREATE INDEX idx_br_item_type ON build_requests(item_type_id);
CREATE INDEX idx_br_external ON build_requests(external_system, external_ref);
```

3.10 Maintenance logging and attachments

3.10.1 `maintenance_logs`

Log of work performed on a specific asset (inspection, repair, refurbishment step, software update, etc.).

```
CREATE TABLE maintenance_logs (
  id BIGINT PRIMARY KEY,
  asset_id BIGINT NOT NULL,

  performed_by_ref VARCHAR(191) NOT NULL, -- user/service principal
  performed_at TIMESTAMP NOT NULL,

  kind VARCHAR(32) NOT NULL,              --
'inspection','repair','update','clean','refurb','qa','other'
  summary VARCHAR(191) NOT NULL,
  details TEXT NULL,

  outcome VARCHAR(32) NULL,               -- 'pass','fail','n_a'
  next_action VARCHAR(64) NULL,           --
'return_to_pool','send_refurb','retire','hold',...

  schema_org JSON NULL,                   -- optional JSON-LD (e.g.,
RepairAction)
  metadata JSON NULL,

  created_at TIMESTAMP NULL,
  updated_at TIMESTAMP NULL,

  CONSTRAINT fk_ml_asset FOREIGN KEY (asset_id) REFERENCES assets(id)
);

CREATE INDEX idx_ml_asset ON maintenance_logs(asset_id);
CREATE INDEX idx_ml_time ON maintenance_logs(performed_at);
```

3.10.2 `attachments`

Generic attachment table for photos/documents. Photo upload support is implemented via pre-signed upload URLs or direct multipart upload.

```
CREATE TABLE attachments (
  id BIGINT PRIMARY KEY,
  owner_kind VARCHAR(32) NOT NULL,  --
'maintenance_log','asset','rent_action',...
  owner_id BIGINT NOT NULL,

  file_name VARCHAR(255) NOT NULL,
  content_type VARCHAR(128) NOT NULL,
  byte_size BIGINT NOT NULL,

  storage_provider VARCHAR(32) NOT NULL, -- 's3','gcs','local'
  storage_key VARCHAR(512) NOT NULL,

  checksum VARCHAR(128) NULL,
  metadata JSON NULL,

  created_at TIMESTAMP NULL
);

CREATE INDEX idx_att_owner ON attachments(owner_kind, owner_id);
```

---

4. Availability and conflict rules

4.1 Interval overlap

Two windows overlap if:

```
(startA < endB) AND (endA > startB)
```

4.2 Capacity by item type

4.2.1 Serialized item types

- `total_capacity` = count of reservable `assets` for the `item_type_id`.
- `reserved_capacity(window)` = count of those assets allocated to overlapping **approved** rent actions.
- `available = total_capacity - reserved_capacity`.

Optionally subtract assets already in non-reservable states (`maintenance`, `retired`, etc.).

4.2.2 Fungible item types

- `total_capacity` = sum of `stock_pools.quantity_total` for relevant pools (optionally filtered by location).
- `reserved_capacity(window)` = sum of allocated quantities (or committed `allocated_quantity`) for overlapping **approved** rent actions.
- `available = total_capacity - reserved_capacity`.

4.2.3 Kits

Kit availability is derived from limiting components:

1. For each kit component `C` with `quantity_per_kit`:
2. Compute `available_component_units(window)` using serialized/fungible rules.
3. `kits_supported_by_C = floor(available_component_units / quantity_per_kit)`
4. `available_kits = min(kits_supported_by_C)`

Recommendation: allocate at kit-level during planning; optionally allocate component assets at fulfillment time.

4.3 Approval/Allocation policy

On approval or allocation, each line item may: - **Hard-fail** if insufficient availability, or - **Allow under-allocation**: approve but leave `allocated_quantity < requested_quantity` and emit a shortfall trigger.

---

5. API endpoints (standardized)

5.1 Conventions

- Versioned base path: `/v1/...`
- Request/response bodies support JSON and JSON-LD.
- For `RentAction` resources, the service SHOULD accept and emit JSON-LD using Schema.org.
- Idempotency: POST endpoints SHOULD accept `Idempotency-Key`.

5.2 Catalog

- `GET  /v1/catalog/item-types`
- `POST /v1/catalog/item-types`
- `GET  /v1/catalog/item-types/{id}`
- `PATCH /v1/catalog/item-types/{id}`
- `GET  /v1/catalog/assets?itemTypeId=&status=&location=`

- `POST /v1/catalog/assets`
- `GET /v1/catalog/assets/{id}`
- `PATCH /v1/catalog/assets/{id}`
- `GET /v1/catalog/stock-pools?itemTypeId=&location=`
- `POST /v1/catalog/stock-pools`
- `PATCH /v1/catalog/stock-pools/{id}`
- `GET /v1/catalog/kit-templates`
- `POST /v1/catalog/kit-templates`
- `GET /v1/catalog/kit-templates/{id}`
- `PATCH /v1/catalog/kit-templates/{id}`

5.3 Availability

- `GET /v1/availability?itemTypeId=&startTime=&endTime=&quantity=&location=`
- `GET /v1/availability/kits?kitTemplateId=&startTime=&endTime=&quantity=&location=`
- `GET /v1/availability/assets?itemTypeId=&startTime=&endTime=&location=`
- `GET /v1/availability/asset?assetId=&startTime=&endTime=`

5.4 Reservations (`RentAction`)

- `POST /v1/rent-actions` (create)
- `GET /v1/rent-actions?status=&startFrom=&endUntil=&externalRef=`
- `GET /v1/rent-actions/{id}`
- `PATCH /v1/rent-actions/{id}` (edit window, metadata, quantities)

State transitions: - `POST /v1/rent-actions/{id}/approve` - `POST /v1/rent-actions/{id}/reject` - `POST /v1/rent-actions/{id}/cancel`

Allocation helpers: - `POST /v1/rent-actions/{id}/allocate` (auto) - `POST /v1/rent-actions/{id}/release-allocations` - `GET /v1/rent-actions/{id}/allocations`

5.5 Demand planning

- `GET /v1/planning/demand-signals?startFrom=&endUntil=&itemTypeId=&source=`
- `POST /v1/planning/demand-signals` (create/import)
- `PATCH /v1/planning/demand-signals/{id}`
- `DELETE /v1/planning/demand-signals/{id}` (optional; or mark cancelled)
- `GET /v1/planning/shortfalls?startFrom=&endUntil=&itemTypeId=&location=`

- returns computed shortfall windows and recommended build quantities

- `POST /v1/planning/build-requests` (create a build request)

- `GET /v1/planning/build-requests?status=&itemTypeId=&neededBy=`
- `GET /v1/planning/build-requests/{id}`
- `PATCH /v1/planning/build-requests/{id}` (update status, external refs)

5.6 Maintenance logs and uploads

- `GET /v1/assets/{assetId}/maintenance-logs`
- `POST /v1/assets/{assetId}/maintenance-logs`
- `GET /v1/maintenance-logs/{id}`

Attachments: - `POST /v1/attachments/initiate` (returns upload URL + attachment draft) - `POST /v1/attachments/complete` (finalize after upload, optional) - `GET /v1/attachments/{id}` (metadata + download URL)

5.7 Trigger management

- `GET /v1/triggers`
- `POST /v1/triggers`
- `PATCH /v1/triggers/{id}`

- `DELETE /v1/triggers/{id}`

- `GET /v1/webhooks`

- `POST /v1/webhooks`
- `PATCH /v1/webhooks/{id}`
- `DELETE /v1/webhooks/{id}`

Optional event feed: - `GET /v1/events?since=&types=`

---

6. Triggers (standardized events; configurable actions)

6.1 Event types (contract)

Catalog: - `catalog.itemType.created` - `catalog.itemType.updated` - `catalog.asset.created` - `catalog.asset.updated` - `catalog.stockPool.updated` - `catalog.kitTemplate.updated`

RentAction lifecycle: - `rentAction.created` - `rentAction.updated` - `rentAction.status.changed` - `rentAction.approved` - `rentAction.rejected` - `rentAction.cancelled`

Allocations: - `rentAction.allocation.created` - `rentAction.allocation.released` - `rentAction.allocation.changed`

Time-based: - `rentAction.starts.soon` - `rentAction.ends.soon` - `rentAction.overdue`

Planning: - `availability.shortfall.detected` - `buildRequest.requested` - `buildRequest.submitted` - `buildRequest.failed` - `buildRequest.fulfilled`

Maintenance: - `maintenanceLog.created` - `attachment.created`

6.2 Event envelope (standard)

All trigger deliveries use a consistent envelope:

```
{
  "id": "evt_...",
  "type": "rentAction.approved",
  "occurredAt": "2026-02-03T12:34:56Z",
  "subject": {"type": "RentAction", "id": "ra_123"},
  "data": {
    "@context": "https://schema.org",
    "@type": "RentAction",
    "identifier": "ra_123",
    "startTime": "...",
    "endTime": "...",
    "actionStatus": "ActiveActionStatus",
    "object": {"@type": "Product", "identifier": "itemType_7", "name":
"TimerNode v2"},
    "requestedQuantity": 12,
    "allocatedQuantity": 10,
    "externalSource": "sgl",
    "externalRef": "event-123"
  },
  "meta": {
    "tenant": "default",
    "correlationId": "...",
    "version": 1
  }
}
```

6.3 Trigger rules

A trigger rule binds: - `eventTypes`: list of event types - `filter`: optional conditions over envelope fields - `actions`: one or more configurable actions

6.4 Trigger sources (event-driven and scheduled)

The system supports two standardized trigger sources:

1. **Event-driven triggers** (default)
2. Fire in response to one or more event types emitted by the Outbox.
3. Example: on `rentAction.approved`, call LMT webhook.
4. **Scheduled triggers**
5. Fire on a schedule (cron/rrule) and execute one or more actions.
6. Scheduled triggers can optionally **emit** a synthetic event into the Outbox (recommended) so deliveries remain auditable and replayable.

Recommended scheduled use cases: - Nightly/weekly **planning runs** to compute shortfalls and create `build_requests`. - Reminder workflows (starts soon / ends soon) if not implemented as separate time-based workers. - Periodic synchronization/import tasks (if needed).

6.5 Scheduled trigger configuration

Scheduled triggers extend the trigger rule with a `schedule` and a `jobType`:

- `schedule`:
- `cron` (preferred) or `rrule`.
- Optional timezone.
- `jobType` (standardized):
- `planning.shortfall_scan`
- `planning.build_request_submit`
- `notifications.starts_soon_scan`
- `notifications.ends_soon_scan`
- `maintenance.overdue_scan` (optional)

Example (cron-based):

```
{
  "name": "Nightly shortfall scan",
  "enabled": true,
  "source": "schedule",
  "schedule": {"type": "cron", "expr": "0 2 * * *", "timezone": "UTC"},
  "jobType": "planning.shortfall_scan",
  "filter": {"meta.tenant": "default"},
  "actions": [
    {"type": "invoke_internal", "config": {"handler":
"planning.shortfall_scan"}},
    {"type": "webhook", "config": {"url": "https://lmt.internal/webhooks/
planning"}}
```

```
  ]
}
```

Implementation note: - For auditability, scheduled runs SHOULD write an outbox event (e.g., `planning.shortfall_scan.completed`) including summary results.

6.6 Action types (configurable) (configurable)

- **WebhookAction**: POST event to URL (HMAC signing, retries, backoff).
- **TransformAction**: transform payload (redaction/mapping) before delivery.
- **RouteAction**: publish to queue/topic.
- **InvokeInternalAction**: call internal integration (e.g., LMT handler).
- Recommended use: submit `build_requests` to InvenTree and write back `external_ref`.
- **Filter/RateLimit**: guard rails.

---

7. Reliability and delivery

7.1 Outbox pattern

All trigger emissions are written to an **outbox** within the same DB transaction as the state change. A worker processes outbox entries and performs deliveries.

Benefits: - Reservation writes do not fail due to webhook downtime. - Deliveries are retryable and auditable. - Events can be replayed.

7.2 Delivery records

Store delivery attempts (success/failure, response codes, timestamps) for observability and debugging.

---

8. Security and access control

8.1 Authentication

Initial recommendation: - Service-to-service API keys or signed JWT.

8.2 Authorization

Roles/scopes: - `planner`: create/update rent actions - `approver`: approve/reject/cancel - `allocator`: allocate/release allocations - `catalog_admin`: manage catalog objects - `admin`: all permissions

---

9. Optional UI (later)

A future UI may provide: - calendar and capacity views - approval queue - allocation dashboard

UI is out of scope for initial delivery; all UI should rely exclusively on the standardized APIs above.