

Product Panel Spec 0.4

Note: This document is the sole normative contract for the Product Panel. Any other documents are explanatory only and MUST NOT override this spec.

Architecture Context: This spec is designed for DayOf's specific tech stack—assumptions about deployment, rendering, and state management are baked into every decision. Understanding the architecture explains **why** the contract looks the way it does.

Key Stack Facts:

- **Framework:** TanStack Start v1 (SSR-first, server functions, not REST API)
- **Deployment:** Vercel with **atomic deploys** (server + client = one build, one version, always in sync)
- **Rendering:** Isomorphic execution (same code runs server-side for SSR and client-side for hydration)
- **State:** Jotai atoms + TanStack Query (derived state from server facts, no prop drilling)
- **Validation:** Zod 4 schemas validate **twice**: server (SSR/loaders) and client (hydration/forms)
- **Data Flow:** Server functions → server state → atoms derive presentation → React renders

How This Impacts The Contract:

1. **No version skew** → Strict validation (`.strict()`) catches bugs; unknown fields are errors, not forward-compat concerns
2. **SSR + hydration** → Same schemas run server + client; consistency is critical to prevent hydration mismatches
3. **Atomic deploys** → Schema changes deploy atomically; no "old client + new server" scenarios
4. **Server functions** → No separate API versioning; contract is isomorphic (server and client share types)
5. **Derived state** → Client never computes business logic; atoms transform server facts into UI state
6. **No caching across deploys** → No stale data surviving deployments; every deploy = fresh contract

What This Means for Implementers:

- Unknown fields = validation errors (fail fast, not graceful degradation)
- Business fields are **required** (no `.default()` hiding server bugs)
- Enums are **strict** (coordinated deploys are fine; catch typos immediately)
- Template checks happen at **render time** (schema layer doesn't know about template registry)
- Currency/money handled via **Dinero.js V2 snapshots** (nothing happens to money outside Dinero utils)

This is **not** a traditional REST API contract designed for independent client/server versioning. It's an **isomorphic TanStack Start contract** where server and client are two execution contexts of the same codebase.

See §14 "Architecture Context: TanStack Start & Validation Strategy" for detailed rationale and comparison with traditional API design patterns.

Sections

1. Purpose & Scope

2. Terminology & Lexicon (*authoritative names; forbidden → preferred table*)

3. State Model (Orthogonal Axes)

- 3.1 Temporal
- 3.2 **Supply** (*canonical; no "availability/inventory"*)
- 3.3 **Gating** (*with `listingPolicy`*)
- 3.4 Demand Capture
- (*Admin axis removed—disabled never sent*)

4. Top-Level Contract Shape (*Context, Sections, Items, Pricing*)

- **Context includes:** `orderRules`, `gatingSummary`, `panelNotices`, `copyTemplates`, `clientCopy`, `tooltips/hovercards`, `prefs`

5. Preferences & Copy *(incl. payment plan banner rule)*

6. Item Structure *(product, variant, fulfillment, commercial clamp)*

7. Messages (Unified)

- Replace `reasonTexts` + `microcopy` split with `state.messages[]` + optional `copyTemplates`

8. Rendering Composition (Derived Atoms Only)

- Row presentation (normal/locked)
- Purchasable boolean
- CTA decision tree
- Quantity & price visibility rules

9. Gating & Unlock (No Leakage)

- `listingPolicy="omit_until_unlocked"` default; `gatingSummary` is the only hint

10. Relations & Add-ons *(selection vs ownership; applyQuantity/matchBehavior)*

11. Pricing Footer *(server math; inclusions flags)*

12. Reason Codes Registry *(machine codes; copy via messages/templates)*

13. Invariants & Guardrails *(client MUST NOT compute X)*

14. Zod/TS Schemas *(single source of truth)*

1. Purpose & Scope

Normative

- **Purpose:** Define the JSON **contract** between the server and the Product Panel UI. The server is the **single source of truth** for all business decisions; the client is a **pure view** that derives presentation from server facts.
- **What this contract MUST cover (inclusive scope):**
 - **Axes of state** per item: `temporal`, `supply`, `gating`, `demand`.
 - **Selection rules** for the panel: `context.orderRules` (types/tickets mins & composition).
 - **Row-level messages:** `state.messages[]` (the only inline text channel per item).
 - **Panel-level notices:** `context.panelNotices[]` (the only banner channel).
 - **Gating behavior:** `gating.required | satisfied | listingPolicy`, plus `context.gatingSummary` for zero-leak hints.
 - **Authoritative clamps & money:** `commercial.maxSelectable`, `pricing` (server-computed).
 - **Display hints:** badges/placements/hovercards/tooltips as provided in `context/display`.
- **What is explicitly out of scope (client MUST NOT implement):**
 - Payments, checkout orchestration, taxes/fees math, discounts, installment scheduling.
 - Identity/auth, membership verification, abuse/rate-limit policies.
 - Seat maps, seat adjacency rules, hold management.
 - Business policy recomputation (sale windows, availability, purchase limits).
 - Any approval/request workflow (not part of this contract).
- **Panel scope boundary:**
 - The Product Panel covers everything **up to** adding tickets and add-ons to an order.
 - It ensures the user has selected what they need (meeting min/max rules from `orderRules`) and validates selection completeness.
 - The panel then **hands off** to the checkout process with the finalized selection.
 - Payment processing, seat assignment (reserved seating), and authentication flows are **outside** this contract.
 - This separation keeps panel logic focused on **product selection** rather than transaction orchestration.

- **Contract guarantees:**

- **Orthogonality:** Each axis is independent; causes go in `reasons[]`; user-facing text goes in `messages[]` / `panelNotices[]`.
- **Zero-leak gating:** Default sendability is governed by `gating.listingPolicy`; omitted items **do not** appear in `items[]`.
- **Single clamps:** UI quantity controls enforce **only** `commercial.maxSelectable`.
- **Single speech channel per level:** rows speak via `state.messages[]`; the panel speaks via `context.panelNotices[]`. No other strings.
- **Replace over infer:** When new payloads arrive, the client **replaces** derived state from server facts; it does not back-compute truth.

- **Validation & compatibility (TanStack Start):**

- Clients **MUST** validate payloads strictly on both server (SSR/loaders) and client (hydration). Unknown fields are **validation errors**.
- Servers **MUST** send all business-meaningful fields explicitly; clients **MUST NOT** invent defaults that change business meaning.
- All machine codes and enums are authoritative; clients **MUST NOT** transform codes into UI text without payload strings/templates.

Rationale (why these boundaries exist)

- The panel must be predictable, testable, and secure. Keeping **decisions on the server** and **derivations in the client** prevents drift, reduces combinatorial bugs, and blocks leakage for gated SKUs.
- One row text channel and one banner channel avoid precedence fights, duplicated copy, and translation churn.
- A single clamp and server-computed pricing eliminate the gray area where client math contradicts business policy.

Real-time synchronization

- The client receives the **authoritative state** from the server at load time.
- Real-time updates (via polling, WebSocket, or server push) ensure the client always reflects current state without local guesswork.
- When the payload refreshes (e.g., inventory changes, sale window transitions, access code unlocked), atoms **re-derive** UI state from the new server facts.
- The client does **not** maintain a separate "source of truth" or attempt to predict state transitions; it derives presentation from what the server sends and requests fresh data when the user acts or time elapses.

Flow: Server fact changes → new payload arrives → atoms re-run derivations → React re-renders UI. No prediction, no back-calculation.

Examples (compact)

Inclusive scope, minimal top-level shape

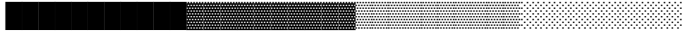
```
{
  "context": {
    "orderRules": {
      "types": "multiple",
      "typesPerOrder": "multiple",
      "ticketsPerType": "multiple",
      "minSelectedTypes": 0,
      "minTicketsPerSelectedType": 0
    },
    "gatingSummary": { "hasHiddenGatedItems": true },
    "panelNotices": [
      {
        "code": "requires_code",
```

```

        "text": "Enter access code to view tickets",
        "variant": "info",
        "priority": 90
      }
    ],
    "effectivePrefs": {
      "showTypeListWhenSoldOut": true,
      "displayPaymentPlanAvailable": false
    }
  },
  "sections": [{ "id": "main", "label": "Tickets", "order": 1 }],
  "items": [
    {
      "product": {
        "id": "prod_ga",
        "name": "General Admission",
        "type": "ticket"
      },
      "state": {
        "temporal": { "phase": "during", "reasons": [] },
        "supply": { "status": "available", "reasons": [] },
        "gating": {
          "required": false,
          "satisfied": true,
          "listingPolicy": "omit_until_unlocked",
          "reasons": []
        },
        "demand": { "kind": "none", "reasons": [] },
        "messages": []
      },
      "commercial": {
        "price": {
          "amount": 5000,
          "currency": { "code": "USD", "base": 10, "exponent": 2 },
          "scale": 2
        },
        "feesIncluded": false,
        "maxSelectable": 6
      },
      "display": { "badges": [] }
    }
  ],
  "pricing": {
    "currency": { "code": "USD", "base": 10, "exponent": 2 },
    "lineItems": [
      {
        "code": "TOTAL",
        "label": "Total",
        "amount": {
          "amount": 0,
          "currency": { "code": "USD", "base": 10, "exponent": 2 },
          "scale": 2
        }
      }
    ]
  },
  "mode": "reserve"
}

```

Out of scope (intentionally absent) No local countdown strings, no client-invented banners, no approval flows, no price math.



2. Terminology & Lexicon (authoritative names)

Normative (Terminology & Lexicon)

- **Field naming:** use **camelCase** for all JSON payload fields.
- **Machine codes:** use **snake_case** for all machine codes (e.g., reason codes, message codes, enum literals).
- Availability axis name is **supply**.
- **Axes (canonical):**
 - **temporal**
 - **supply**
 - **gating**
 - **demand**
- **Supply axis:**
 - `supply.status` ∈ "available" | "none" | "unknown".
 - Use `display.showLowRemaining: boolean` for urgency styling; `supply.remaining` for copy interpolation.
 - `supply.reasons` is an array of machine codes (e.g., ["sold_out"]).
 - `supply.remaining` MAY be included (number) but is **not** used for business logic.
- **Gating axis:**
 - `gating.required: boolean`
 - `gating.satisfied: boolean`
 - `gating.listingPolicy` ∈ "omit_until_unlocked" (default) | "visible_locked"
 - Access-code validation is server-side only.
- **Demand axis:** `demand.kind` ∈ "none" | "waitlist" | "notify_me".
- **Unified messages:** row-level display text comes **only** from `state.messages[]` Each message MAY include { `code`, `text?`, `params?`, `placement?`, `variant?`, `priority?` }.
- **Panel banners:** come **only** from `context.panelNotices[]` (priority-ordered; informational only, not primary CTAs).
- **Copy registries:** optional `context.copyTemplates[]` (templates) and `context.clientCopy` (client-reactive microcopy).
- **Gating hint:** `context.gatingSummary.hasHiddenGatedItems: boolean` is the **only** allowed hint that hidden gated items exist.
- **Reason codes:** machine codes are **snake_case** (e.g., `sold_out`, `requires_code`, `sales_ended`). UI strings are supplied by the payload; the client does not hardcode.
- **Display flag naming:** boolean flags controlling UI presentation use consistent prefixes:
 - **show*** for visibility toggles (e.g., `showTypeListWhenSoldOut`, `showLowRemaining`)
 - **display*** for feature availability flags (e.g., `displayPaymentPlanAvailable`, `displayRemainingThreshold`)

Why no hardcoded copy?

All user-facing text and labels come from the server or configurable copy dictionaries—**nothing is hardwired** in the client code. This includes:

- Status messages: "Sold out", "Sales start in...", "Requires code"
- Button labels: "Add", "Join Waitlist", "Notify Me"
- Error prompts: "Max 4 tickets per order", "Minimum 1 ticket required"
- Notices: "Payment plans available at checkout"

Benefits:

- **Consistency** across platforms (web, mobile, kiosk)
- **Localization** without client releases
- **A/B testing** of copy variants server-side
- **Rapid iteration** on messaging (no deployment for text changes)

The server provides either **exact text** ("Sold Out") or **templates** with params ("Only {count} left!"). The client **never invents** strings or derives them from codes alone.

Absolute forbidden terms (contract & code comments)

- **inventory**, **stock** (use **supply.remaining**, and copy like "Only N left" supplied by server).
- **availability** as an axis/field name (use **supply**).
- "view model" / MVVM jargon (refer to **derived state** if needed in client docs).
- Any client-invented banners or row text.

Tests

- **Schema & Naming Compliance**
 - Payloads **MUST** validate with **supply.*** fields present and **no availability.*** or **inventory.*** fields.
 - All machine codes and enums **MUST** use **snake_case** (e.g., **sold_out**, **requires_code**, **sales_ended**).
- **Message Channels**
 - The client **MUST** render messages from **state.messages[]** and **context.panelNotices[]** only; no other strings may be introduced.

Examples (compact & canonical)


```
// Supply axis (sold out)
"supply": { "status": "none", "reasons": ["sold_out"] }

// Gating axis (secure default)
"gating": { "required": true, "satisfied": false, "listingPolicy": "omit_until_unlocked" }

// Demand axis
"demand": { "kind": "waitlist", "reasons": [] }

// Unified row messages
"state": {
  "messages": [
    { "code": "sold_out", "text": "Sold Out", "placement": "row.under_quantity", "priority": 100 }
  ]
}

// Panel notices
"context": {
  "panelNotices": [
    { "code": "payment_plan_available", "variant": "info", "text": "Payment plans available at checkout", "priority": 50 }
  ]
}
```

 A full "forbidden → preferred" mapping table belongs in the **Appendix: Terminology & Migration Notes** (for authoring hygiene). Section 2 remains purely normative.

Client State Architecture (derived state, not ViewModel)

- **Terminology (normative):** Use the word **state**, not “view model”. The client maintains and consumes **derived state** computed from the server contract. No two-way bindings.
- **One-way flow:** contract → derive/compose via atoms → client state → React components. No props drilling for panel data.

State Flow (one direction)

```

Server Contract (PanelItem)
    ↓
Atoms (derive + compose)
    ↓
Client State (RowState, SectionState, PanelState)
    ↓
React Components (consume via useAtomValue)

```

No props, no drilling (pattern)

```

function ProductPanel() {
  const panelState = useAtomValue(panelStateAtom);
  return panelState.sections.map((section) => (
    <Section key={section.id} id={section.id} />
  ));
}

function Section({ id }: { id: string }) {
  const section = useAtomValue(sectionStateAtom(id));
  return section.rows.map((row) => (
    <ProductRow key={row.key} rowId={row.key} />
  ));
}

function ProductRow({ rowId }: { rowId: string }) {
  const rowState = useAtomValue(rowStateAtom(rowId));
  const [quantity, setQuantity] = useAtom(selectionFamily(rowId));
  // rowState: { presentation, quantityUI, priceUI, cta, ... }
}

```

State Types Hierarchy (authoritative names)

```

export type RowState = {
  key: string;
  presentation: RowPresentation;
  quantityUI: QuantityUI;
  priceUI: PriceUI;
  cta: RowCTA;
  maxSelectable: number;
};

export type SectionState = {
  id: string;
  label: string;
  rows: RowState[];
};

export type PanelState = {
  sections: SectionState[];
  allOutOfStock: boolean;
}

```

```
anyLockedVisible: boolean;
};
```

State Atoms Pattern (derive → compose → consume)

```
export const productPanelQueryAtom =
  atomWithQuery<ProductPanelPayload>(/* ... */);

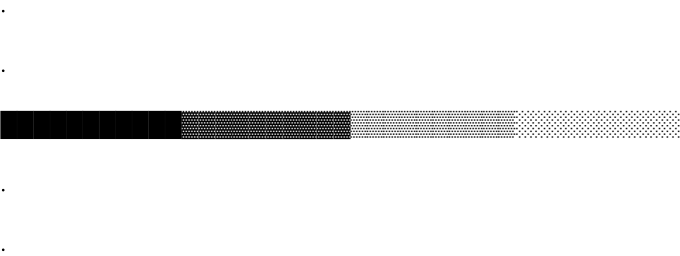
export const rowStatesAtom = atom((get) => {
  const payload = get(productPanelQueryAtom);
  return payload.items.map((item) => deriveRowState(item, payload));
});

export const panelStateAtom = atom<PanelState>((get) => {
  const rowStates = get(rowStatesAtom);
  const sections = get(sectionsAtom);
  // compose PanelState from rowStates + sections + rollups
  return derivePanelState({ rowStates, sections });
});

export const selectionFamily = atomFamily((rowKey: string) =>
  atom(0, (get, set, qty: number) => {
    const row = get(rowStateAtom(rowKey));
    const clamped = Math.max(0, Math.min(qty, row.maxSelectable));
    set(selectionFamily(rowKey), clamped);
  })
);
```

Why “State” not “ViewModel”

Reason	Explanation
React convention	“Derived state” is standard React terminology.
Jotai philosophy	Atoms hold state; components consume state.
No MVVM baggage	ViewModel implies two-way binding; we use one-way flow.
Clear boundaries	Contract (server) → State (client-derived) → Components (render).
Future-proof	“Selection state”, “gate state”, “pricing state” all fit.



3. State Model (Orthogonal Axes)

The server sends **independent axes**; the client composes them into presentation. Axes: **temporal**, **supply**, **gating**, **demand**. (Admin axis removed — unpublished/disabled items are never sent.)

Why Axes?

Each product's status is broken down into **independent dimensions** (timing, supply, access, demand). This allows the server to explicitly signal *why* an item is or isn't purchasable at any moment—for example, distinguishing "sold out" from "not on sale yet"

from "requires access code."

The client composes these orthogonal facts into presentation states through pure view mapping—**no business logic, no schedule math, no price computation**. A ticket might be temporally available (`phase="during"`) but supply-constrained (`status="none"`) and offering a waitlist (`kind="waitlist"`)—three independent server decisions that together determine the row's CTA.

This separation makes the system **testable** (toggle one axis at a time), **extensible** (add geo-gating or age verification later without touching existing axes), and **secure** (client cannot recompute or bypass server decisions).

Normative (global to the state model)

- Each item **MUST** include:

```
"state": {
  "temporal": { ... },
  "supply":   { ... },
  "gating":   { ... },
  "demand":   { ... },
  "messages": [ /* optional, row display-only */ ]
}
```

- Independence:** Axes **MUST NOT** encode each other's decisions. Causes go in `reasons[]` (machine codes, `snake_case`). User text **MUST** come from `state.messages[]` (or `context.copyTemplates`), not from `reasons[]`.
- No leakage:** Only `gating.listingPolicy` and server omission control whether an item is sent. Client **MUST NOT** infer missing items except via `context.gatingSummary.hasHiddenGatedItems`.
- Purchasability (derived later):** "Purchasable" is a client-derived boolean: `temporal.phase="during" AND supply.status="available" AND (gating.required=false OR gating.satisfied=true) AND commercial.maxSelectable > 0`. The derivation lives in §8; this section defines the axes that make it possible.

3.1 Temporal

Normative

- `temporal.phase` **MUST** be one of: `"before"` | `"during"` | `"after"`.
- `temporal.reasons[]` **MAY** include machine codes such as:
 - `outside_window` (phase=`before`)
 - `sales_ended` (phase=`after`)
- The server **MUST** decide the phase; the client **MUST NOT** compute from timestamps or clocks.
- Any human-readable timing text (e.g., "On sale Fri 10 AM CT") **MUST** be delivered via `state.messages[]` or `context.copyTemplates`.
- `temporal.currentWindow` and `temporal.nextWindow` **MAY** provide ISO timestamp objects for display metadata only:
 - These are **not** for client computation of `phase` or countdown rendering.
 - The client **MAY** format timestamps for locale display only (e.g., converting ISO to local time format).
 - The client **MUST NOT** compute countdowns, sale windows, or phase transitions from timestamps.
 - If countdown UI is desired, the server **MUST** send refreshed text or trigger client re-fetch on schedule transitions.
 - Recommended:** Send pre-formatted text whenever possible to avoid client-side time formatting complexity.

Rationale

- Time math (TZ, DST, pauses) belongs on the server; clients only reflect the call.

Examples

```
// Not on sale yet
"temporal": { "phase": "before", "reasons": ["outside_window"] }

// On sale
"temporal": { "phase": "during", "reasons": [] }

// Sales ended
"temporal": { "phase": "after", "reasons": ["sales_ended"] }
```

Tests

- Given `phase="before"`, purchasability derivation **MUST** be false regardless of other axes.
- Given `phase` toggles `"before"→"during"` in a refresh, purchasability **MUST** re-compute to true iff other axes permit (no extra client logic).
- Client **MUST NOT** display a countdown unless a message is provided; no clock math is allowed.

3.2 Supply

Normative

- `supply.status` **MUST** be one of: `"available"` | `"none"` | `"unknown"`.
 - `"available"`: stock exists (use `supply.remaining` and `display.showLowRemaining` for urgency)
 - `"none"`: sold out
 - `"unknown"`: status cannot be determined (e.g., external seat map not ready)
- When sold out, `supply.reasons` **SHOULD** include `sold_out`.
- `supply.remaining` **MAY** be present (number) for copy/urgency, but **MUST NOT** drive business rules.
- Authoritative clamp**: selection controls enforce **only** `commercial.maxSelectable` (see §13 guardrails). Supply does **not** set the clamp; it explains it.

Rationale

- Counts can be stale, seat maps can constrain selection, and limits apply; a single clamp avoids contradictions.

Examples

```
// In stock (count optional)
"supply": { "status": "available", "remaining": 5, "reasons": [] }

// Sold out
"supply": { "status": "none", "reasons": ["sold_out"] }

// Unknown (e.g., seat provider session not ready)
"supply": { "status": "unknown", "reasons": [] }
```

Tests

- With `supply.status="none"`, the row **MUST** not be purchasable (CTA decided in §8).
- With `supply.status="available"` and `commercial.maxSelectable=0`, the row **MUST** not render quantity controls (clamp wins).
- Client **MUST NOT** infer `sold_out` from `remaining=0`; only `status`/messages govern display.
- Clamp authority**: Given `commercial.maxSelectable=2` and any `supply.remaining` value (including 0), the quantity UI **MUST** clamp at 2. The client **MUST NOT** derive limits from `remaining` or other fields.
- MaxSelectable vs remaining**: Given `commercial.maxSelectable: 2` and `supply.remaining: 0`, quantity UI **MUST** allow up to 2 (clamp is authoritative; no inference from counts).

- **Authoring note:** The "remaining=0 but maxSelectable>0" fixture is didactic (proves clamp authority). In production payloads, prefer keeping `remaining` omitted or consistent with clamp unless there is an intentional reason (e.g., async seat batch allocation), and accompany edge cases with clear `state.messages[]`.

3.3 Gating (with `listingPolicy`)

Normative

- Gated items **MUST** set `gating.required=true`.
- If gate unsatisfied:
 - `listingPolicy="omit_until_unlocked"` (default) ⇒ item **MUST NOT** be sent in `items[]`.
 - `listingPolicy="visible_locked"` ⇒ item **MUST** be sent as **locked**; price **MUST** be masked; quantity UI **MUST** be hidden.
- Client **MUST NOT** infer omitted items. Only use `context.gatingSummary.hasHiddenGatedItems`.
- Access code validation **MUST** occur server-side; client **MUST NOT** validate or rate-limit.
- `gating.requirements[]` **MAY** provide structured metadata about gate constraints:
 - `kind`: type of gate (e.g., `"unlock_code"`)
 - `satisfied`: whether this specific requirement is met
 - `validWindow`: optional time window when gate is active
 - `limit`: optional usage constraints (`maxUses`, `usesRemaining`)
 - The server uses this metadata to compute `gating.satisfied`; the client uses it **only** for explanatory messages (e.g., "Code expired", "Code max uses reached").
 - The client **MUST NOT** use `requirements[]` to decide purchasability; only `gating.satisfied` governs that.
- On successful unlock, previously omitted items **MUST** appear with `gating.satisfied=true` (and become purchasable only if other axes allow).

Rationale

- Zero-leak default protects presales/secret SKUs. `visible_locked` is an explicit tease mode.

Examples

```
// Hidden until unlock (default)
"gating": { "required": true, "satisfied": false, "listingPolicy": "omit_until_unlocked" }

// Visible locked (tease)
"gating": {
  "required": true,
  "satisfied": false,
  "listingPolicy": "visible_locked",
  "reasons": ["requires_code"]
}

// Visible locked with requirements metadata (for error messaging)
"gating": {
  "required": true,
  "satisfied": false,
  "listingPolicy": "visible_locked",
  "requirements": [{
    "kind": "unlock_code",
    "satisfied": false,
    "validWindow": {
      "startsAt": "2025-10-22T00:00:00Z",
      "endsAt": "2025-10-25T23:59:59Z"
    }
  }]
}
```

```

    },
    "limit": {
      "maxUses": 100,
      "usesRemaining": 23
    }
  }],
  "reasons": ["requires_code"]
}

// Unlocked
"gating": { "required": true, "satisfied": true, "listingPolicy": "omit_until_unlocked" }
```

Tests

- Given `omit_until_unlocked`, item **MUST** be absent from `items[]`; `context.gatingSummary.hasHiddenGatedItems=true`.
- Given `visible_locked`, row renders **locked**; price masked; quantity UI hidden.
- After unlock, `gating.satisfied=true`; CTA resolves to **Purchase** iff `temporal="during"` and `supply="available"`.
- Omit enforcement:** If an item has `gating.required=true`, `satisfied=false`, and `listingPolicy="omit_until_unlocked"`, it **MUST NOT** be present in `items[]`.
- Hint presence:** When items are omitted due to gating, `context.gatingSummary.hasHiddenGatedItems` **MUST** be `true` iff any omitted item has stock.
- No leakage:** The client **MUST NOT** display, cache, or log any information about omitted items beyond the boolean `hasHiddenGatedItems` hint.
- Price masking:** Locked rows (`gating.required=true` && `satisfied=false`) **MUST** mask price and hide quantity controls.

UX nuances (non-normative, clarifying)

- Post-unlock confirmation:** If a user enters a valid access code but the unlocked inventory is already sold out, the UI should still confirm success.
 - For previously visible-locked items, keep the row visible but disabled with a sold-out message (confirms the code worked).
 - For items previously omitted via `omit_until_unlocked`, show a panel-level confirmation (e.g., a short notice indicating the code was valid) if no new rows appear; avoid a “dead end” feeling.
- Price visibility:** Prices are masked only when a row is locked. For sold-out or otherwise non-purchasable rows (including when `commercial.maxSelectable=0`), price **MUST NOT** be shown per §8 price visibility rules. The normative rules above still govern masking when locked.
- Public sold-out + hidden gated stock:** When `context.gatingSummary.hasHiddenGatedItems === true`, prefer an access-code prompt over a terminal “Event Sold Out” state to guide the user toward unlocking.

3.4 Demand (alternate actions)

Normative

- `demand.kind` **MUST** be one of: `"none"` | `"waitlist"` | `"notify_me"`.
- `demand.reasons[]` **MAY** annotate machine facts (e.g., `waitlist_available`).
- Demand **does not** override gating: if `gating.required` && `!satisfied`, the client **MUST NOT** surface demand CTAs that would leak locked inventory.
- CTA mapping from `demand.kind` is defined in §8 (Rendering/CTA Decision).

Rationale

- Demand expresses the server-chosen fallback when direct purchase is not available. Gating precedence prevents leakage and UX confusion.

Examples

```
// Waitlist offered (e.g., sold out)
"demand": { "kind": "waitlist", "reasons": ["waitlist_available"] }

// Notify-me (before sale)
"demand": { "kind": "notify_me", "reasons": [] }

// No alternate
"demand": { "kind": "none", "reasons": [] }
```

Tests

- With `supply.status="none"` and `demand.kind="waitlist"` and gate satisfied (or not required), CTA **MUST** resolve to **Join Waitlist** (per §8).
- With `temporal.phase="before"` and `demand.kind="notify_me"`, CTA **MUST** resolve to **Notify Me** (per §8).
- With `gating.required=true` and `satisfied=false`, demand CTAs **MUST NOT** appear (until unlocked).
- **No approval CTAs:** The client **MUST NOT** render any "Request Access" or approval-related CTAs; these are not part of this contract.

3.5 Cross-Axis Invariants (quick)

Normative

- **No duplication:** A cause appears once (in its axis' `reasons[]`); display text appears once (`state.messages[]`) or as `context.panelNotices[]`.
- **Gating precedence:** If `gating.required=true` AND `gating.satisfied=false`, demand CTAs (waitlist/notify_me) **MUST NOT** be shown, even if `demand.kind` is set.
 - **Rationale:** Prevents leaking locked inventory. A waitlist CTA for an unsatisfied gated item would reveal its existence to unauthorized users.
 - **Example:** A members-only ticket sells out. The server sets `demand.kind="waitlist"` but keeps `gating.satisfied=false` for non-members. The client shows neither purchase nor waitlist CTAs until the gate is satisfied.
- **Visibility vs. lock:** Visibility is controlled by **sendability** (`omit_until_unlocked` or server omission). Locking is a **rendered state** of a **sent** item.
- **No admin axis:** Items not for sale (unpublished/disabled) are omitted server-side; the client will never see them.

Tests

- A gated, unsatisfied item **cannot** be both omitted and visible: `listingPolicy` defines exactly one behavior.
- Given `gating.required=true`, `satisfied=false`, `demand.kind="waitlist"`: row **MUST** show lock state with **no** waitlist CTA (gating precedence).
- **Banner exclusivity:** With an empty `context.panelNotices[]`, the panel **MUST NOT** render any top banners—even if all visible rows are sold out. All panel-level notices **MUST** come from `context.panelNotices[]` only.
- Unknown fields in `state` are **invalid** under strict validation.

Tiny end-to-end example (axes only, minimal messages)

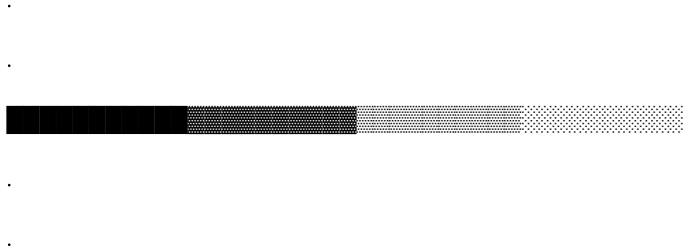
```
{
  "product": { "id": "prod_ga", "name": "General Admission", "type": "ticket" },
  "state": {
    "temporal": { "phase": "during", "reasons": [] },
    "supply": { "status": "available", "reasons": [] },
    "gating": {
      "required": false,
      "satisfied": true,
      "listingPolicy": "omit_until_unlocked",
      "reasons": []
    }
  }
}
```

```

    },
    "demand": { "kind": "none", "reasons": [] },
    "messages": []
  },
  "commercial": { "maxSelectable": 6 },
  "display": { "badges": [] }
}

```

This section defines **what the server says**. §7 specifies how rows speak (`messages[]`). §8 defines **how the client renders** from these axes (row presentation, CTA table, quantity/price visibility).



4. Top-Level Contract Shape (*Context, Sections, Items, Pricing*)

Normative

Root object

- The payload **MUST** be an object with exactly these top-level keys:
 - `context` (*required*)
 - `sections` (*required, array*)
 - `items` (*required, array*)
 - `pricing` (*required*)
- Clients **MUST** validate root keys strictly; unknown top-level keys are **invalid**.
- Machine codes (e.g., in `reasons[]` and `messages[].code`) **MUST** be `snake_case`. Field names remain camelCase.

4.1 `context` (server-authored panel configuration & copy)

The server is the single source for configuration, banner text, and copy artifacts.

- MUST** include:
 - `orderRules` — selection composition for the panel:
 - `types`: "single" | "multiple"
 - `typesPerOrder`: "single" | "multiple"
 - `ticketsPerType`: "single" | "multiple"
 - `minSelectedTypes`: integer ≥ 0
 - `minTicketsPerSelectedType`: integer ≥ 0
 - `panelNotices[]` — panel-level banners, ordered by descending `priority`.
 - Each notice **MAY** specify `{ code, text?, params?, variant?, priority?, action?, expiresAt? }`.
 - `effectivePrefs` — display preferences (non-business, UI hints):
 - SHOULD** at least provide:
 - `showTypeListWhenSoldOut`: boolean:
 - `true`: when all items are sold out, still render the list of ticket types (with "Sold out" labels)

- **false**: when all items are sold out, hide the list entirely and show only a panel notice (e.g., "Event Sold Out")
 - Default recommended: **true** (transparency)
 - **displayPaymentPlanAvailable**: **boolean**: informational flag; if **true**, the server typically includes a **panelNotices[]** entry about payment plans at checkout
 - **displayRemainingThreshold?**: **number**: optional threshold for urgency styling (e.g., if remaining \leq this value, set **display.showLowRemaining=true**)
- **MUST/SHOULD** behavior:
 - **gatingSummary**:
 - **MUST** be present **iff** any gating is configured for the event.
 - **MUST** include **hasHiddenGatedItems**: **boolean** (the **only** hint that omitted, gated SKUs exist).
 - **MAY** include **hasAccessCode**: **boolean** (general capability hint).
 - Copy registries:
 - **copyTemplates[]** **MAY** supply templates with **{ key, template }** structure, where **key** matches **messages[].code** for interpolating **messages[].params**.
 - **clientCopy** **MAY** provide client-reactive strings (e.g., min/max selection errors).
 - Help text:
 - **tooltips[]** and/or **hovercards[]** **MAY** provide referenced explainer content for badges and UI affordances.
- **MUST NOT** contain **reasonTexts** for row rendering (row text comes from **state.messages[]**; see §7).

4.2 sections (grouping/navigation)

- **sections[]** **MUST** be sent to group rows for display organization.
 - Each section: **{ id, label, order, labelOverride? }**.
 - The server determines section IDs, labels, and order; the client **MUST NOT** assume any specific section IDs or names.
- Items assign themselves to sections via **display.sectionId**.
 - If **display.sectionId** is absent, the item's section placement is undefined; the client **SHOULD** render it in the first section by **order**.
- Empty sections (no items assigned) **MAY** be hidden by the client.

4.3 items (the list of purchasable & related products)

Each element in **items[]** is a **Panel Item** the client renders verbatim (no client filtering beyond derivation rules).

- **MUST** include:
 - **product**: **{ id, name, type }** where **type** \in "ticket" | "digital" | "physical".
 - **product.id** **MUST** be unique across the payload.
 - **variant** **MAY** exist (forward-compat; empty object acceptable).
 - **state**: **MUST** include the four orthogonal axes **plus** the unified message channel:

Axes (decision facts):

- **temporal**: **{ phase: "before" | "during" | "after", reasons[] }**
 - See §3.1 for full spec; **currentWindow/nextWindow** metadata **MAY** be present.
- **supply**: **{ status: "available" | "none" | "unknown", reasons[], remaining? }**
 - See §3.2 for full spec.

- `gating: { required: boolean, satisfied: boolean, listingPolicy: "omit_until_unlocked" | "visible_locked", reasons[], requirements?[] }`
 - See §3.3 for full spec including `requirements[]` shape (kind, validWindow, limit).
- `demand: { kind: "none" | "waitlist" | "notify_me", reasons[] }`
 - See §3.4 for full spec.

Display text channel (composes axis info into UI strings):

- `messages[]`: array of `{ code, text?, params?, placement?, variant?, priority? }`
 - The **only** row-level text channel; composes information from all axes into user-facing strings.
 - **Not** an axis; it's the presentation layer that derives from axis `reasons[]`.
 - `variant`: explicit styling (e.g., `"warning"`, `"error"`, `"info"`, `"neutral"`) for icon/color
 - `priority`: for sorting when multiple messages exist (higher numbers first)
 - **Example relationship:**

```
// Axis has machine code
"supply": { "status": "none", "reasons": ["sold_out"] }

// Messages has display text for that code
"messages": [
  { "code": "sold_out", "text": "Sold Out", "placement":
    "row.under_quantity", "variant": "info" }
]
```

◦ `commercial`: authoritative clamps & money:

- **MUST** include `{ price, feesIncluded, maxSelectable, limits? }`.
- `price`: **Dinero.js V2 snapshot object** with `{ amount, currency, scale }`:
 - `amount`: integer in minor units (e.g., `5000` for \$50.00)
 - `currency`: full currency object `{ code, base, exponent }` (e.g., `{ code: "USD", base: 10, exponent: 2 }`)
 - `scale`: precision scale (typically `2` for cents)
 - **Rationale**: All money is transported as Dinero snapshots; nothing happens to money outside Dinero.js V2 and Dinero utils. Client **MAY** display without Dinero utils, but transport is always Dinero objects.
- `feesIncluded: boolean`: whether `price` includes fees or fees are added separately
- `maxSelectable: computed` effective cap that accounts for current stock, `limits.perOrder`, `limits.perUser`, and any other server-side constraints (holds, fraud rules, etc.).
 - When `0`, the item cannot be selected (sold out, locked, or otherwise unavailable).
 - UI quantity controls **MUST** enforce **only** this value; never derive limits from other fields.
- `limits.perOrder / limits.perUser`: **optional** business rules; informational for display (e.g., "Max 4 per order"), but `maxSelectable` is authoritative for UI enforcement.

◦ `display`: view hints:

- **SHOULD** include `{ badges[], sectionId? }`.
- **MAY** include `{ badgeDetails: { [badge]: { kind: "tooltip"|"hovercard", ref } }, showLowRemaining? }`.
- **Field explanations:**
 - `badges[]`: short labels for this specific product (e.g., `["Popular", "Members"]`); product-level, not section-level.
 - `badgeDetails`: optional tooltips/hovercards explaining badges (references `context.tooltips[]` or `context.hovercards[]`).
 - `showLowRemaining: boolean`: when `true`, signals the UI should apply urgency styling **for this item** (e.g., highlight the row, pulse animation). The actual count comes from `supply.remaining`; this is just a

presentation flag.

- **Not** a section-level FOMO notice; for event-wide urgency ("Only 20 tickets left!"), use `context.panelNotices[]`.
 - `sectionId: string`: assigns this item to a section from `sections[]`. If absent, client renders in first section by `order`.
- `relations` **MAY** express add-on dependencies:
 - `parentProductIds?: string[]` — IDs of products this item depends on. If empty/absent, the item has no dependencies.
 - `matchBehavior?: "per_ticket" | "per_order"`:
 - `"per_ticket"`: this add-on can be selected once per parent ticket (e.g., meal voucher per attendee).
 - `"per_order"`: this add-on is limited to the order level regardless of parent quantity (e.g., one parking pass per order).
 - **Note:** An "add-on" is not a separate `product.type`; it's any product with `parentProductIds[]` populated. The backend manages junction table relationships; the panel receives the derived `relations` object for UI enforcement.
- **MUST NOT** include any "admin/approval" axis (unpublished items are never sent; approval flows are out of scope).
- **Gating sendability:**
 - If `required=true` and `satisfied=false` and `listingPolicy="omit_until_unlocked"` → item **MUST NOT** appear in `items[]`.
 - If `visible_locked`, the item **MUST** appear locked; price masked; quantity controls hidden.

4.4 pricing (server-computed money summary)

⚠ **Work in Progress:** The pricing contract structure is evolving. The `mode` field, `lineItems[]` granularity, and breakdown format may change as implementation progresses. The server may ultimately send a simpler structure (subtotal, fees, taxes, total) rather than detailed line items. This section will be updated as the pricing contract stabilizes over the next week.

- The pricing footer is **authoritative** and **server-computed**; clients **MUST NOT** perform price math.
- **MUST** include:
 - `currency: Dinero.js V2 currency object { code, base, exponent }` (e.g., `{ code: "USD", base: 10, exponent: 2 }`)
 - Applies to all monetary values in this pricing object
 - `mode?: "reserve" | "final"` (*provisional field; may be removed*):
 - `"reserve"`: interim pricing during selection (face value + estimated fees); may change as user selects items
 - `"final"`: locked-in price after all discounts, taxes, payment plan adjustments applied
 - **Breakdown structure** (*provisional; exact fields TBD*):
 - May use `lineItems[]` with detailed breakdown: `{ code, label, amount }` where each `amount` is a Dinero snapshot
 - May use simpler fields like `subtotal`, `fees`, `taxes`, `total` (all Dinero snapshots)
 - Common line item codes (if used): `"TICKETS"`, `"FEES"`, `"TAX"`, `"DISCOUNT"`, `"TOTAL"`
 - Negative amounts allowed for discounts
- **Stable architectural principles** (*regardless of final structure*):
 - All monetary amounts are **Dinero.js V2 snapshot objects** `{ amount, currency, scale }`
 - Server computes all totals, fees, taxes, discounts; client receives Dinero snapshots
 - Client **MAY** use Dinero utils for display formatting or **MAY** format directly from snapshot values
 - Client **MUST NOT** perform arithmetic on money; all calculations happen server-side and are returned as new Dinero snapshots
 - Nothing happens to money outside Dinero.js V2 and Dinero utils
- **Dynamic updates:** As the user changes selection quantities, the client **requests** a new payload from the server (or receives a push update). The server recalculates `pricing` including all fees, taxes, discounts, payment plan effects, etc. The client

replaces the footer with the new **pricing** object.

- **Always present:** **pricing** is always included in the payload. If there is nothing to display yet (e.g., no selection), send `{ currency, lineItems: [] }`. The client **MUST NOT** compute or backfill totals; it renders only what the server provides.

Rationale

- This shape keeps **configuration, copy, and security** at the top (**context**), **facts** in the middle (**items[]**.state, **commercial**), and **money** at the bottom (**pricing**).
- **sections** provide grouping without coupling to item structure; the server decides section IDs and labels.
- Unified **state.messages[]** + **context.panelNotices[]** prevents copy collisions and implicit heuristics.
- Authoritative **maxSelectable** + server **pricing** eliminate client/server drift from local math.
- **Dinero.js V2 architecture** ensures all money operations are safe, precise, and server-computed:
 - All amounts transported as Dinero snapshots (persisted the same way)
 - Client never performs arithmetic; only formats for display
 - Eliminates floating-point errors and currency conversion bugs

How the Pieces Relate (*mental model*)

Understanding how context, items, and pricing work together:

Context provides the rules and UI scaffolding:

- **orderRules** tells the client how selections can be composed (one type vs many, minimums, etc.)
- **gatingSummary.hasHiddenGatedItems** hints that omitted items exist (without leaking details)
- **panelNotices[]** provides event-level banners ("Payment plans available", "Enter access code")
- **effectivePrefs** controls display behaviors (show/hide sold-out list)
- **copyTemplates**, **tooltips**, **hovercards** supply reusable text artifacts

Items provide the facts about each product:

- **product.id** is the unique key
- **state** (four axes + messages) describes **why** a product is/isn't purchasable
- **commercial.maxSelectable** is the authoritative clamp (accounts for stock + limits + holds + fraud rules)
- **display** provides view hints (badges, showLowRemaining flag, sectionId assignment)
- **relations** defines add-on dependencies (parentProductIds, matchBehavior)

Pricing provides the computed money summary:

- Updated by the server as the user changes selections
- Client **replaces** footer with new **lineItems**; never computes totals locally
- **mode**: **"reserve"** during selection, **"final"** after discounts/taxes applied

Example flow:

1. User loads panel → server sends **context** (rules + notices), **sections** (grouping), **items[]** (facts), **pricing** (initial \$0)
2. User selects 2 GA tickets → client POSTs selection → server responds with updated **items[]** (maxSelectable adjusted?) + new **pricing** (tickets + fees)
3. User enters access code → server responds with previously omitted items now in **items[]**, **gating.satisfied=true**, updated **gatingSummary**
4. User clicks checkout → panel validates against **orderRules**, proceeds if valid

Examples (*tiny, canonical*)

Note: Pricing examples show **lineItems[]** structure as provisional. Actual implementation may use simpler or more complex breakdown fields.

Minimal viable payload (single section)

```

{
  "context": {
    "orderRules": {
      "types": "multiple",
      "typesPerOrder": "multiple",
      "ticketsPerType": "multiple",
      "minSelectedTypes": 0,
      "minTicketsPerSelectedType": 0
    },
    "gatingSummary": { "hasHiddenGatedItems": false },
    "panelNotices": [],
    "effectivePrefs": {
      "showTypeListWhenSoldOut": true,
      "displayPaymentPlanAvailable": false
    }
  },
  "sections": [{ "id": "main", "label": "Tickets", "order": 1 }],
  "items": [
    {
      "product": {
        "id": "prod_ga",
        "name": "General Admission",
        "type": "ticket"
      },
      "variant": {},
      "state": {
        "temporal": { "phase": "during", "reasons": [] },
        "supply": { "status": "available", "reasons": [] },
        "gating": {
          "required": false,
          "satisfied": true,
          "listingPolicy": "omit_until_unlocked",
          "reasons": []
        },
        "demand": { "kind": "none", "reasons": [] },
        "messages": []
      },
      "commercial": {
        "price": {
          "amount": 5000,
          "currency": { "code": "USD", "base": 10, "exponent": 2 },
          "scale": 2
        },
        "feesIncluded": false,
        "maxSelectable": 10
      },
      "display": { "badges": ["Popular"] }
    }
  ],
  "pricing": {
    "currency": { "code": "USD", "base": 10, "exponent": 2 },
    "mode": "reserve",
    "lineItems": [
      {
        "code": "TOTAL",
        "label": "Total",
        "amount": {
          "amount": 0,
          "currency": { "code": "USD", "base": 10, "exponent": 2 },
          "scale": 2
        }
      }
    ]
  }
}

```

```

    }
  }

```

With multiple sections, locked row, panel banner, templates

```

{
  "context": {
    "orderRules": {
      "types": "multiple",
      "typesPerOrder": "multiple",
      "ticketsPerType": "multiple",
      "minSelectedTypes": 0,
      "minTicketsPerSelectedType": 0
    },
    "gatingSummary": { "hasHiddenGatedItems": true },
    "panelNotices": [
      {
        "code": "requires_code",
        "variant": "info",
        "text": "Enter access code to view tickets",
        "priority": 90
      }
    ],
    "effectivePrefs": {
      "showTypeListWhenSoldOut": true,
      "displayPaymentPlanAvailable": true
    },
    "copyTemplates": [
      { "key": "remaining_low", "template": "Only {count} left!" }
    ],
    "hovercards": [
      {
        "id": "members_info",
        "title": "Members Only",
        "body": "Unlock with a valid access code."
      }
    ]
  },
  "sections": [
    { "id": "primary", "label": "Tickets", "order": 1, "labelOverride": null },
    { "id": "addons", "label": "Add-ons", "order": 2, "labelOverride": null }
  ],
  "items": [
    {
      "product": {
        "id": "prod_locked",
        "name": "Members Presale",
        "type": "ticket"
      },
      "variant": {},
      "state": {
        "temporal": { "phase": "during", "reasons": [] },
        "supply": { "status": "available", "reasons": [] },
        "gating": {
          "required": true,
          "satisfied": false,
          "listingPolicy": "visible_locked",
          "reasons": ["requires_code"]
        },
        "demand": { "kind": "none", "reasons": [] },
        "messages": [
          {

```

```

        "code": "requires_code",
        "text": "Requires access code",
        "placement": "row.under_title",
        "variant": "info",
        "priority": 80
      }
    ]
  },
  "commercial": {
    "price": {
      "amount": 9000,
      "currency": { "code": "USD", "base": 10, "exponent": 2 },
      "scale": 2
    },
    "feesIncluded": false,
    "maxSelectable": 0
  },
  "display": {
    "badges": ["Members"],
    "badgeDetails": {
      "Members": { "kind": "hovercard", "ref": "members_info" }
    },
    "sectionId": "primary"
  }
},
],
"pricing": {
  "currency": { "code": "USD", "base": 10, "exponent": 2 },
  "mode": "reserve",
  "lineItems": [
    {
      "code": "TOTAL",
      "label": "Total",
      "amount": {
        "amount": 0,
        "currency": { "code": "USD", "base": 10, "exponent": 2 },
        "scale": 2
      }
    }
  ]
}
]
}
}

```

Tests

- **Root keys present**

- Given a payload without **context**, **sections**, **items**, or **pricing**, the client **MUST** reject the payload (validation error).
- Given an unknown top-level key, the client **MUST** reject the payload (validation error).

- **Context rules respected**

- Given **context.orderRules** with **typesPerOrder="single"**, the client **MUST** enforce single-type selection when enabling the bottom CTA (no extra business logic).
- Given **context.gatingSummary.hasHiddenGatedItems=true**, the client **MUST NOT** invent row placeholders; it **MAY** show only the banner(s) in **panelNotices**.

- **Sections rendering**

- When an item lacks **display.sectionId**, the client **SHOULD** render it in the first section by **order**.
- Empty sections (no items) **MAY** be hidden by the client.

- **Items integrity**

- Given two `items[]` with the same `product.id`, the client **MUST** treat this as invalid (duplicate row key).
- Given an item with `gating.required=true`, `satisfied=false`, `listingPolicy="omit_until_unlocked"`, the item **MUST NOT** be present in `items[]`.

- **Pricing authority**

- Given `pricing.lineItems=[]`, the footer **MUST** render no lines (no client recomputation); changing quantities **MUST** request new server pricing rather than compute locally.
- `pricing` is always present; the client **MUST NOT** compute totals locally under any circumstances.

Author note (for implementers): Lock these shapes into your Zod/TS schemas. Keep `messages[]` and `panelNotices[]` as the only speech channels; keep clamps (`maxSelectable`) and money (`pricing`) server-authored.

5. Preferences & Copy (*incl. payment plan banner rule*)

Normative

5.1 `context.effectivePrefs` (UI hints only; never business logic)

The server **MAY** include UI preferences that shape **presentation**, not policy. Clients **MUST NOT** use them to compute availability, prices, or clamps.

- `showTypeListWhenSoldOut: boolean` —
 - `true` ⇒ keep rows visible (disabled) when everything's sold out.
 - `false` ⇒ collapse to a compact sold-out panel.
 - Default is server-defined; client renders exactly what is sent. Clients **MUST NOT** assume UI defaults for absent fields.
 - **Rationale:** Different events benefit from different treatments. Showing sold-out types builds context and FOMO (driving waitlist signups), while hiding them provides a cleaner UX for postponed/cancelled events. This flexibility is independent of the strict price-hiding policy (see §6/§8).
- `displayPaymentPlanAvailable: boolean` — indicates that installment plans exist at checkout.
 - **MUST NOT** by itself render any banner or badge. See 5.3 for the banner rule.
- `displayRemainingThreshold?: number` — optional urgency threshold that the **server** uses to decide when to set `display.showLowRemaining=true` (e.g., when `supply.remaining ≤ threshold`).
 - **Informational only:** tells the client what threshold the server is using; the client does not perform the comparison.
 - This flag **MUST NOT** alter purchasability or client logic.
- Unknown prefs are **invalid** (validation error).

5.2 Copy channels (single source at each level)

- **Row-level copy:** `items[].state.messages[]` is the **only** inline text channel per row. Each entry **MAY** include `{ code, text?, params?, placement?, variant?, priority? }`.
 - `variant`: explicit styling variant (e.g., `"warning"`, `"error"`, `"info"`, `"neutral"`) for icon/color selection
 - `priority`: for sorting when multiple messages exist (higher numbers first)

- **Panel-level banners:** `context.panelNotices[]` for informational banners only (not CTAs). Each notice **MAY** include `{ code, text?, params?, variant?, priority?, action?, expiresAt? }`.
 - **Not for primary actions:** Access code entry and waitlist signup are **panel-level CTAs**, not notices (see below).
- Clients **MUST NOT** render any text invented locally (no hardcoded "Sold out", no client-authored banners).
- `context.reasonTexts` is **not part of this contract**; row text comes from `state.messages[]` only (see §4.1).

How the two channels work together:

The separation of panel vs row notices ensures broad messages don't get lost among the list, and each item's specific status is clearly labeled right next to it. Key scenarios:

- **Visible but locked item:** The lock state is conveyed as a **row-level message** (e.g., "Requires access code" with lock icon). The user sees the item but knows a code is needed. The **access code CTA** appears below the `PanelActionButton`.
- **Hidden until unlocked:** A **panel-level notice** MAY appear at top (e.g., "Enter access code to view tickets") for guidance. The **access code CTA** appears below the `PanelActionButton`. After successful unlock, the notice may be removed; unlocked items appear with their own row-level status.
- **Event sold out with waitlist:** The **PanelActionButton changes to "Join Waitlist"** (see §5.3a for derivation). A **panel notice** MAY announce "All tickets sold out" for context. Individual sold-out rows show their own **row messages** ("Sold out").
- **Event not on sale with waitlist:** The **PanelActionButton becomes "Join Waitlist"** (see §5.3a). A **panel notice** MAY provide timing info ("Tickets on sale Friday at 10 AM").
- **Multiple concerns:** An item can have multiple row messages (e.g., "Requires access code" + "Sales end tomorrow"). The panel can have multiple notices (e.g., "Payment plans available" + informational alerts). Priority sorting ensures critical messages appear first.

5.3 Payment plan banner rule (authoritative)

- `effectivePrefs.displayPaymentPlanAvailable=true` **MUST NOT** auto-render any banner or badge.
- A payment-plan banner **MUST** be rendered **only** when the server sends a notice:

```
{
  "code": "payment_plan_available",
  "variant": "info",
  "text": "Payment plans available at checkout",
  "priority": 50
}
```

- Per-row "Payment Plan" badges **MUST NOT** be rendered. The concept is order-level and surfaces only via `panelNotices[]`.
- Clients **MUST** sort `panelNotices[]` by **descending numeric priority** (higher numbers first).
 - Default priority is `0` if omitted.
 - Ties preserve server order.
- **Notice actions** (*optional, for secondary actions only*): Notices **MAY** include an `action: { label, kind, target? }` for supplementary interactive elements:
 - `kind: "link"` → open external URL (e.g., "Learn More" link about payment plans)
 - `kind: "drawer"` → open internal info panel/modal (e.g., "Why join the waitlist?" explainer)
 - The client renders the action as a secondary button/link using the provided `label`.
 - **Not for primary CTAs:** Waitlist signup and access code entry use panel-level CTAs (`PanelActionButton`, `AccessCodeCTA`), not notice actions.

5.3a Panel-level CTA derivation (`PanelActionButton` & `AccessCodeCTA`)

The panel has two derived CTAs that are **not** `panelNotices[]`:

PanelActionButton (main button at bottom):

- **States:**

- **"continue"**: Default when items are purchasable and selection is valid → button shows "Continue" (or "Checkout")
- **"waitlist"**: When no items purchasable but waitlist available → button shows "Join Waitlist"
- **"disabled"**: When selection invalid (doesn't meet **orderRules**) or nothing purchasable with no waitlist → button grayed out

- **Waitlist derivation logic:**

- **IF** all visible items are **not** purchasable (no item has **temporal.phase="during"** AND **supply.status="available"** AND gating satisfied)
- **AND** any **eligible visible** item (gate satisfied or not required; not hidden by **omit_until_unlocked**) has **demand.kind="waitlist"**
- **THEN** **PanelActionButton** state = **"waitlist"** (button shows "Join Waitlist")
- **Note:** Respects gating precedence (§3.5)—locked items with waitlist do not trigger panel waitlist CTA until unlocked.

- **Common scenarios:**

- Event not on sale yet (**temporal.phase="before"**) + waitlist enabled → "Join Waitlist"
- All tickets sold out (**supply.status="none"**) + waitlist → "Join Waitlist"
- Sales ended (**temporal.phase="after"**) + waitlist for next event → "Join Waitlist"

- **Label source:** **PanelActionButton** labels **MUST** come from server-provided copy (e.g., **context.clientCopy**) or app-level copy that is sourced from the server (e.g., **panel_cta_continue**, **panel_cta_waitlist**, **panel_cta_disabled**). Clients **MUST NOT** hardcode these strings. See §5.4/§7.1 for template resolution rules.

AccessCodeCTA (positioned below **PanelActionButton**):

- **Appears when:**

- **context.gatingSummary.hasHiddenGatedItems=true**, OR
- Any visible item has **gating.required=true** && **satisfied=false**

- **Typical UI:** Input field + "Apply Code" button (or similar)
- **Not a **panelNotice****; it's a persistent UI element derived from gating state
- A panel notice **MAY** provide additional guidance at the top (e.g., "Enter access code to view tickets")

Key principle: These CTAs are **derived from panel/item state**, not configured via notices. Notices provide context; CTAs provide actions.

5.4 Templates & interpolation

- **context.copyTemplates[]** **MAY** define templates { **key**, **template**, **locale?** }.
- A row message **MUST** resolve its display text as:
 1. use **messages[].text** if provided; else
 2. find **copyTemplates[].key === messages[].code**, interpolate {**...messages[].params**}, and render; else
 3. omit the message (no fallback strings).
- **panelNotices[]** follow the same rule: if **text** is absent and a template with **key === code** exists, interpolate; otherwise omit the notice.
- Placeholders use {**name**} syntax. Unknown placeholders **MUST** resolve to "" (empty string), not the literal token.

5.5 Client-triggered copy (**context.clientCopy**)

Row messages can be **server-authored** (included in **state.messages[]**) or **client-triggered** (using **context.clientCopy** templates):

- **Server-authored messages:** Static or dynamic row status from the server (e.g., "Sold out", "Only 5 left!", "Includes VIP lounge access"). These are always in **state.messages[]**.

- **Client-triggered messages:** Validation errors the client shows in response to user actions (e.g., trying to checkout without meeting `orderRules`, exceeding `maxSelectable`). These use `context.clientCopy` templates.

`context.clientCopy` strings:

The server **MAY** provide strings for client-triggered validation/errors, e.g.:

- `selection_min_reached` — "Please select at least one ticket."
- `selection_max_types` — "You can only select one ticket type."
- `quantity_min_reached` — "Minimum {min} tickets required for this type."
- `quantity_max_reached` — "Maximum {max} tickets for this type."

Usage:

- Clients **MUST** use these strings verbatim when they initiate the message (e.g., on invalid checkout press). No local wording.
- These strings are **templates** that can include placeholders (e.g., `{max}`, `{min}`); the client interpolates them with current values.

5.6 Tooltips & hovercards (progressive disclosure)

- `context.tooltips[]` / `context.hovercards[]` **MAY** define reusable explainer content.
- `items[].display.badgeDetails[badge]` **MAY** reference `{ kind: "tooltip" | "hovercard", ref }`.
- Clients **MUST** resolve these references and render the supplied content; no local prose.

5.7 Variant & placement

- **variant** **SHOULD** be one of: `"neutral"` | `"info"` | `"warning"` | `"error"`; unrecognized values default to `"info"`.
 - `"neutral"`: informational, no urgency; subtle styling (e.g., "Includes VIP lounge access")
 - `"info"`: standard status; default styling (e.g., "Only 5 left!")
 - `"warning"`: urgent or time-sensitive; attention-grabbing styling (e.g., "Sales end in 1 hour!")
 - `"error"`: problem requiring user action; error styling (e.g., "Invalid access code")
- **Purpose:** Explicit styling control for icon, color, and urgency level. The server chooses the variant; the client applies corresponding styles.
- Recommended `messages[].placement` slots:
 - `row.under_title` — Small text just below the row title
 - `row.under_price` — Inline with or under price
 - `row.under_quantity` — Under the quantity control / CTA area
 - `row.footer` — At bottom of the row
 - `row.cta_label` — CTA button label text for the row
- Clients **MUST NOT** display messages outside the declared placements.

Multiple messages per row:

- An item **MAY** have multiple messages (e.g., "Requires access code" + "Sales end tomorrow").
- Clients **MUST** display all messages with valid text/templates, sorted by descending `priority`.
- If placement conflicts (two messages for same slot), stack them vertically or use the highest-priority one (implementation choice).
- The client **collates** messages and displays them in designated spots (typically as smaller, styled text under the relevant UI element).

Rationale

- **One voice per layer.** Row facts speak via `messages[]`; panel-wide context speaks via `panelNotices[]`. This prevents precedence fights and localization churn.
- **Server controls tone & timing.** Copy changes, A/B tests, and localization land on the server—no client releases or guesswork.

- **Security & clarity.** No implicit banners ("maybe there's a code?"). If the server wants the user to see it, the server sends it.

Examples

A) Preferences with no banners (no auto-render)

```
"context": {
  "effectivePrefs": {
    "showTypeListWhenSoldOut": true,
    "displayPaymentPlanAvailable": true,
    "displayRemainingThreshold": 10
  },
  "panelNotices": []
}
```

Result: no payment-plan banner. No badges. Nothing is shown unless the server sends a notice.

B) Payment plan banner present (authoritative)

```
"context": {
  "effectivePrefs": { "displayPaymentPlanAvailable": true },
  "panelNotices": [
    {
      "code": "payment_plan_available",
      "variant": "info",
      "text": "Payment plans available at checkout",
      "priority": 50
    }
  ]
}
```

B2) Panel notice with additional guidance (not primary CTA):

```
"context": {
  "panelNotices": [
    {
      "code": "event_sold_out_info",
      "variant": "info",
      "text": "All tickets are sold out. Join our waitlist to be notified if tickets become available.",
      "priority": 100
    }
  ]
}
// Note: The PanelActionButton itself becomes "Join Waitlist" (not via this notice).
// This notice provides context; the button provides the action.
```

C) Using templates for row messages (no **text** on the message)

```
"context": {
  "copyTemplates": [
    { "key": "remaining_low", "template": "Only {count} left!" }
  ]
},
"items": [{
```

```

    "state": {
      "supply": { "status": "available", "remaining": 3, "reasons": [] },
      "messages": [
        { "code": "remaining_low", "params": { "count": 3 }, "placement":
"row.under_quantity", "priority": 60 }
      ]
    }
  }
}]

```

C2) Informational message (neutral variant):

```

"items": [{
  "state": {
    "messages": [
      {
        "code": "vip_includes",
        "text": "Includes VIP lounge access",
        "placement": "row.under_title",
        "variant": "neutral",
        "priority": 50
      }
    ]
  }
}]

```

Rendering: Displayed as subtle, non-urgent informational text (e.g., gray, smaller font). Not a warning or error.

D) Two banners, ordered by **priority**

```

"context": {
  "panelNotices": [
    { "code": "requires_code", "text": "Enter access code to view tickets", "variant":
"info", "priority": 90 },
    { "code": "payment_plan_available", "text": "Payment plans available at checkout",
"variant": "info", "priority": 50 }
  ]
}

```

Rendering order: "Enter access code ..." above "Payment plans ...".

E) Multiple messages per row (priority sorted)

```

"items": [{
  "state": {
    "gating": { "required": true, "satisfied": false, "reasons": ["requires_code"] },
    "temporal": { "phase": "during", "reasons": ["sales_end_soon"] },
    "messages": [
      {
        "code": "requires_code",
        "text": "Requires access code",
        "placement": "row.under_title",
        "variant": "info",
        "priority": 90
      },
      {
        "code": "sales_end_soon",
        "text": "Sales end tonight at 11:59 PM",
        "placement": "row.under_title",

```

```

        "variant": "warning",
        "priority": 80
      }
    ]
  }
}]

```

Rendering: Both messages appear under title, sorted by priority (requires_code first). Client may stack them or show highest priority only.

F) Badge detail via hovercard

```

"context": {
  "hovercards": [
    { "id": "members_info", "title": "Members Only", "body": "Unlock with a valid access code." }
  ],
  "items": [{
    "display": {
      "badges": ["Members"],
      "badgeDetails": { "Members": { "kind": "hovercard", "ref": "members_info" } }
    }
  }]
}

```

Invariants & guardrails (REMEMBER)





- **No auto-banners.** `effectivePrefs.*` never creates UI by itself. Banners come **only** from `panelNotices[]`.
- **No per-row payment-plan badges.** Ever. It's an order-level concept.
- **Messages require text or a template.** If neither is present, **omit** the message/notice.
- **Templates are not business logic.** Interpolate parameters; do not compute counts, dates, or phases. Those originate server-side.
- **Unknown placeholders resolve to empty string.** When interpolating templates, any unknown `{placeholder}` **MUST** resolve to `""`.
- **Respect placements and priorities.** Don't reshuffle banners or move messages to new locations.
- **No `reasonTexts` mapping:** This is not part of the contract. Use `messages[]` and `copyTemplates[]` for all display text.

Client do / don't

Do

- Render banners strictly from `panelNotices[]`, sorted by descending `priority`.
- Use `messages[].text` or `copyTemplates` interpolation; otherwise omit.
- Use `clientCopy` for client-triggered validations (min/max selection, etc.).
- Treat `effectivePrefs` as presentation hints only.

Don't

- Don't hardcode strings like "Sold out", "Enter access code", "Join Waitlist".
- Don't auto-show payment-plan messaging from `displayPaymentPlanAvailable` flag alone.
- Don't compute parameters for templates:
 -  Don't build `{date_local}` from timestamps (server sends formatted dates)
 -  Don't create "Only N left!" by checking `supply.remaining` (server sends the message)
 -  Don't compute `{max}` for "Max {max} per order" from `limits` (use `clientCopy` template with current `maxSelectable`)
 -  **Do** interpolate server-provided params into server-provided templates

- Don't surface approval/request CTAs—approval flows are out of scope.
- Don't invent new placement slots beyond the defined ones (`row.under_title`, etc.).
- Don't ignore `priority` values; always sort descending (higher first).

Edge cases & tests

- **No banner without notice** Given `displayPaymentPlanAvailable=true` and `panelNotices=[]` Expect no payment-plan banner is rendered.
- **Banner appears with notice** Given `panelNotices=[{ code:"payment_plan_available", text:"...", priority:50 }]` Expect a single info banner rendered at panel top; no per-row badges.
- **Priority ordering** Given `panelNotices` includes both `{ code:"requires_code", priority:90 }` and `{ code:"payment_plan_available", priority:50 }` Expect "requires_code" banner above the payment-plan banner.
- **Template resolution (row)** Given a row message `{ code:"remaining_low", params:{count:2} }` and `copyTemplates[{key:"remaining_low","template":"Only {count} left!"}]` Expect rendered text "Only 2 left!" at `row.under_quantity`.
- **Template missing** Given a row message `{ code:"foo_bar" }` with no `text` and no matching template Expect no message rendered (no fallback text).
- **Client-triggered copy** Given `clientCopy.selection_min_reached="Please select at least one ticket."` and user taps Checkout with no selection Expect that exact string to appear; no client-invented phrasing.
- **Prefs don't change business** Given `displayRemainingThreshold=5` and `supply.remaining=3` but `commercial.maxSelectable=0` Expect urgency styling may apply, but quantity controls remain hidden (clamp wins).
- **Panel notice vs PanelActionButton** Given all items sold out with `demand.kind="waitlist"` and a panel notice `{ code:"event_sold_out_info", text:"All tickets sold out..." }` Expect the notice banner appears at top (informational), and the PanelActionButton at bottom changes to "Join Waitlist" (primary action).
- **Multiple messages per row** Given an item with two messages at same placement, priorities 90 and 80 Expect both messages rendered (stacked or highest-priority only), sorted by descending priority.
- **showTypeListWhenSoldOut behavior** Given all items have `supply.status="none"`, `showTypeListWhenSoldOut=false`, and a panel notice `event_sold_out` Expect the item list is hidden; only the panel notice and its action (if any) are visible. Given same scenario but `showTypeListWhenSoldOut=true` Expect the item list remains visible (with "Sold out" row messages), and the panel notice appears above it.

6. Item Structure (*product, variant, fulfillment, commercial clamp*)

What this section does: pins down the shape and meaning of the **thing you render per row**. Identity and delivery live in `product/variant/fulfillment`. What a user can actually select is governed by the **commercial clamp** (`commercial.maxSelectable`) plus the state axes from §3. **Copy lives elsewhere:** row text is `state.messages[]`; panel banners are `context.panelNotices[]`.

6.1 Normative (contract you can validate)

Each `items[]` element **MUST** include:

- **product** — identity & delivery metadata (no money, no availability):
 - **id**: `string` — unique in payload.
 - **name**: `string`.
 - **type**: `"ticket" | "digital" | "physical"`.
 - **fulfillment?**: `{ methods: string[]; details?: Record<string, unknown> }`
 - **methods[]** **MUST** use server-defined enums; unknown values are **invalid** (validation error):
 - Recommended baseline: `"eticket"`, `"will_call"`, `"physical_mail"`, `"apple_pass"`, `"shipping"`.
 - The client uses these to display appropriate icons or labels (e.g., mobile phone icon for `"eticket"`, Apple Wallet logo for `"apple_pass"`, shipping truck for `"physical_mail"`).
 - **details** is vendor- or product-specific metadata; **display-only**.
 - May include shipping info, pickup instructions, redemption requirements, etc.
 - Client renders as supplementary text/tooltips; does not affect business logic.
 - Extra fields are limited to: `description?: string`, `subtitle?: string`, `category?: string`. Unknown extra fields are **invalid**.
 - **description**: longer product description for detail views or hovercards
 - These are **static product metadata**; for dynamic status text, use `state.messages[]`
- **variant** — differentiation placeholder (forward-compatible):
 - Optional and **MAY** be `{}` (most general admission tickets).
 - When present, **MAY** include:
 - **id?**: `string` — unique variant identifier (for multi-variant products)
 - **name?**: `string` — variant name (e.g., "Large", "Blue", "VIP Section A"); do not duplicate `product.name` unless it truly differs
 - **attributes?**: `Record<string, unknown>` — variant-specific data (size, color, seat info, time slot, etc.)
 - **MUST NOT** contain price or availability; those live in `commercial` and `state`.
 - **Common use cases**:
 - Physical goods: size/color variants (e.g., `{ attributes: { size: "L", color: "black" } }`)
 - Reserved seating: seat location (future; e.g., `{ attributes: { section: "A", row: "5", seat: "12" } }`)
 - Time-slotted tickets: time windows (e.g., `{ attributes: { slot: "10:00-11:00" } }`)
 - Currently: mostly empty `{}` for GA tickets; populated for physical merch
- **commercial** — authoritative clamp & price snapshot:
 - **price**: `{ amount: number; currency: { code: string; base: number; exponent: number }; scale: number }` (*Dinero v2 snapshot; server-computed*)
 - **feesIncluded**: `boolean` — **presentation hint** only (affects copy like "+ fees" vs "incl. fees").
 - **maxSelectable**: `number` — **single source of truth** for the quantity UI clamp (integer ≥ 0).
 - **Computed server-side** from: current `supply.remaining`, `limits.perOrder`, `limits.perUser`, any holds, fraud rules, etc.
 - When `0`, the item cannot be selected (sold out, locked, or unavailable for any reason).
 - **UI enforcement**: Quantity controls clamp to this value; never derive limits from other fields.
 - **Example**: If `supply.remaining=50`, `limits.perOrder=10`, `limits.perUser=6`, and the user has already bought 2 → server sends `maxSelectable=4`.
 - **limits?**: `{ perOrder?: number; perUser?: number }` — **informational only** for display copy (e.g., "Max 4 per order").
 - The client **MAY** use these in `clientCopy` templates but **MUST NOT** use them for quantity clamping.
 - **Example**: Show "Maximum 10 per order" hint text, but enforce only `maxSelectable=4` (the effective cap).

- **state** — the axes from §3 plus `messages[]` (already specified in §§3–5).
- **display** — view hints (see §4): `badges[]`, optional `badgeDetails`, optional `sectionId`, and optional `showLowRemaining: boolean`.
- **relations** (*optional*) — add-on dependencies (see §4.3):
 - `parentProductIds?: string[]` — IDs this item depends on
 - `matchBehavior?: "per_ticket" | "per_order"` — how add-on quantity relates to parent quantity
 - If absent or empty, the item has no dependencies (standalone product)

Field naming: contract fields are **camelCase**; machine **codes** (e.g., `messages[].code`) are **snake_case**.

6.2 Rationale (why the split looks like this)

- **Identity vs. money vs. truth:** `product` says *what it is*; `commercial` says *what you can buy now & how much*; `state` says *why*. This separation prevents accidental business logic drifting into the client.
- **Fulfillment is display-only:** Delivery method affects icons/badges/tooltips, not purchasability. A ticket with `["eticket", "apple_pass"]` shows mobile and Wallet icons; a physical item with `["physical_mail"]` shows a shipping icon. If multiple methods exist, the UI can list them. Special fulfillment requirements (e.g., age restrictions, ID requirements) are conveyed via `state.messages[]` or badge hovercards—never hardcoded.
- **One clamp to rule them all:** only `maxSelectable` controls the quantity UI. Everything else (remaining counts, per-order caps) are folded server-side into that number and/or surfaced as copy.
- **Variant flexibility:** The `variant` field is forward-compatible for future differentiation (seat info, time slots, sizes). Currently minimal or empty for most tickets; used more for physical goods with attributes.

6.3 Examples (tiny, valid JSON)

A) Simple on-sale ticket, mobile + Apple Wallet, clamp 6

```
{
  "product": {
    "id": "prod_ga",
    "name": "General Admission",
    "type": "ticket",
    "fulfillment": { "methods": ["eticket", "apple_pass"] }
  },
  "variant": {},
  "state": {
    "temporal": { "phase": "during", "reasons": [] },
    "supply": { "status": "available", "reasons": [] },
    "gating": {
      "required": false,
      "satisfied": true,
      "listingPolicy": "omit_until_unlocked",
      "reasons": []
    },
    "demand": { "kind": "none", "reasons": [] },
    "messages": []
  },
  "commercial": {
    "price": {
      "amount": 5000,
      "currency": { "code": "USD", "base": 10, "exponent": 2 },
      "scale": 2
    },
    "feesIncluded": false,
    "maxSelectable": 6,
    "limits": { "perOrder": 8, "perUser": 6 }
  }
}
```

```
"display": { "badges": ["Popular"], "showLowRemaining": false }
}
```

B) Physical merch (ships), order-level clamp 1

```
{
  "product": {
    "id": "tee_black",
    "name": "Event Tee (Black)",
    "type": "physical",
    "fulfillment": {
      "methods": ["physical_mail", "shipping"],
      "details": { "shipsFrom": "US-AL" }
    }
  },
  "variant": { "attributes": { "size": "L" } },
  "state": {
    "temporal": { "phase": "during", "reasons": [] },
    "supply": { "status": "available", "reasons": [] },
    "gating": {
      "required": false,
      "satisfied": true,
      "listingPolicy": "omit_until_unlocked",
      "reasons": []
    },
    "demand": { "kind": "none", "reasons": [] },
    "messages": []
  },
  "commercial": {
    "price": {
      "amount": 3000,
      "currency": { "code": "USD", "base": 10, "exponent": 2 },
      "scale": 2
    },
    "feesIncluded": false,
    "maxSelectable": 1,
    "limits": { "perOrder": 1 }
  },
  "display": { "badges": ["Merch"] }
}
```

C) Gated presale ticket (visible locked), price masked by UI, clamp=0

```
{
  "product": {
    "id": "prod_mem",
    "name": "Members Presale",
    "type": "ticket",
    "fulfillment": { "methods": ["eticket"] }
  },
  "variant": {},
  "state": {
    "temporal": { "phase": "during", "reasons": [] },
    "supply": { "status": "available", "reasons": [] },
    "gating": {
      "required": true,
      "satisfied": false,
      "listingPolicy": "visible_locked",
      "reasons": ["requires_code"]
    }
  },
}
```



```

    "demand": { "kind": "none", "reasons": [] },
    "messages": [
      {
        "code": "requires_code",
        "text": "Requires access code",
        "placement": "row.under_title",
        "priority": 80
      }
    ]
  },
  "commercial": {
    "price": {
      "amount": 8000,
      "currency": { "code": "USD", "base": 10, "exponent": 2 },
      "scale": 2
    },
    "feesIncluded": false,
    "maxSelectable": 0
  },
  "display": { "badges": ["Members"], "showLowRemaining": false }
}

```

D) Add-on with parent dependency (per-order parking pass)

```

{
  "product": {
    "id": "addon_parking",
    "name": "Parking Pass",
    "type": "physical",
    "fulfillment": { "methods": ["will_call"] }
  },
  "variant": {},
  "state": {
    "temporal": { "phase": "during", "reasons": [] },
    "supply": { "status": "available", "reasons": [] },
    "gating": {
      "required": false,
      "satisfied": true,
      "listingPolicy": "omit_until_unlocked",
      "reasons": []
    },
    "demand": { "kind": "none", "reasons": [] },
    "messages": []
  },
  "commercial": {
    "price": {
      "amount": 2000,
      "currency": { "code": "USD", "base": 10, "exponent": 2 },
      "scale": 2
    },
    "feesIncluded": false,
    "maxSelectable": 1,
    "limits": { "perOrder": 1 }
  },
  "display": {
    "badges": ["Add-on"],
    "sectionId": "add_ons"
  },
  "relations": {
    "parentProductIds": ["prod_ga", "prod_vip"],
    "matchBehavior": "per_order"
  }
}

```

```
}
}
```

Note: This parking pass requires a parent ticket (GA or VIP) and is limited to 1 per order regardless of ticket quantity.

6.4 Invariants & Guardrails (read before coding)

- **No price in `product` or `variant`.** Price lives only in `commercial.price`. *Reason:* prevents contradictory sources of truth.
- **Clamp is king.** The **only** value that constrains the quantity UI is `commercial.maxSelectable`. *Do not* clamp off `supply.remaining` or `limits.*`.
- **Fees hint only.** `feesIncluded` changes copy ("incl. fees" / "+ fees") but never math; all math is server-computed in `pricing`.
- **Price visibility policy.** Price is shown (`priceUI="shown"`) **only** when purchasable (except `priceUI="masked"` for locked rows). Showing price for non-purchasable rows (sold out, not on sale yet, or when `commercial.maxSelectable=0`) is **disallowed**. *Reason:* Displaying a price for an item you cannot buy creates psychological "tease" and "anchor" effects—users fixate on the unavailable price, leading to confusion ("Why show it if I can't buy it?") and potential frustration. By hiding price when not purchasable, we maintain clear, honest signals: if you see a price, you can act on it. Locked rows use masking (not hiding) to preserve the gating UX without leaking pricing to unauthorized users.
- **Fulfillment is never a gate.** Icons and notes only; it **MUST NOT** change purchasability.
 - **Special requirements** (age restrictions, ID requirements, redemption conditions) are conveyed via `state.messages[]` or badge hovercards.
 - **Example:** An alcohol voucher might have a message "Must present valid ID (21+)" or a badge "21+" with a hovercard explaining the requirement.
 - The panel's job is to inform the buyer **before** purchase; actual redemption (scanning QR, verifying ID) happens outside the panel scope.
- **Locked rows mask price.** When `gating.required && !satisfied`, the client masks price and hides quantity controls—even if `commercial.price` is present.
- **One currency per panel.** The server **MUST** keep currency consistent across `commercial.price` and `pricing.currency`. Clients **MUST NOT** mix.

6.5 Client do / don't (practical)

Do

- Render fulfillment **icons/tooltips** from `product.fulfillment.methods` (e.g., phone icon for "eticket", Wallet mark for "apple_pass").
- Use `display.showLowRemaining` to style urgency; interpolate counts via `state.messages[]` (templates from `context.copyTemplates`).
- Enable quantity UI only when `maxSelectable > 0` **and** the §3 purchasability boolean is true.
- Show "+ fees" / "incl. fees" based on `feesIncluded`.

Don't

- Don't compute totals, taxes, fees, or discounts in the client. Use `pricing` as sent.
- Don't gate UI on `limits.*` or `supply.remaining` directly. They are **copy** only.
- Don't leak gated prices. Mask price when locked.
- Don't add per-row payment-plan badges; surface plans only via a **panel** notice (see §5.3).

6.6 Edge cases & tests (acceptance checks)

- **Clamp beats counts** *Given* `supply.status="available"`, `supply.remaining=50`, `commercial.maxSelectable=0` *Expect* quantity UI hidden; row not selectable.
- **Sold out beats clamp** *Given* `supply.status="none"`, `commercial.maxSelectable=5` *Expect* quantity UI hidden; row shows sold-out message from `state.messages[]`.

- **Locked masks price** Given `gating.required=true`, `satisfied=false`, `listingPolicy="visible_locked"` Expect price masked; quantity hidden; show lock copy from `messages[]`.
- **Free items** Given `commercial.price.amount=0`, purchasability true, `maxSelectable>0` Expect quantity UI enabled; display "Free" via server message template; **no** client logic to infer labels.
- **Currency consistency** Given any item whose `commercial.price.currency.code` differs from `pricing.currency.code` Expect client validation error or block; the server is required to normalize currency per panel.
- **Unknown fulfillment method** Given `product.fulfillment.methods=["weird_future_channel"]` Expect validation error (unknown method).
- **Add-on without parent** Given an add-on with `relations.parentProductIds=["prod_vip"]` but no VIP ticket in the order Expect client may disable add-on selection or show validation message using `clientCopy`; server will reject order if submitted.
- **Add-on with per-ticket match** Given 3 GA tickets selected and a meal voucher with `matchBehavior="per_ticket"` Expect client allows up to 3 meal vouchers (matching parent quantity); server enforces same on submission.

6.7 Migration notes (v0.3 → v0.4)

- **Add-on is not a product type.** Keep `type ∈ {"ticket","digital","physical"}`. Add-on behavior is modeled via `relations` (§11).
- **Visibility → listing policy.** Replace `gating.visibilityPolicy` with `gating.listingPolicy ∈ {"omit_until_unlocked","visible_locked"}`.
- **Reason text channel unified.** Replace split `reasonTexts/microcopy` with `state.messages[]` (+ optional `context.copyTemplates`).
- **Price location unified.** Any legacy `variant.price/product.price` must move to `commercial.price`.

6.8 Quick reference tables

Product types


product.type	Meaning	Common fulfillment methods
ticket	Admission or time-bound entitlement	eticket, apple_pass, will_call
digital	Non-physical voucher/benefit	eticket (barcode), nfc (optional)
physical	Shipped or picked-up goods	physical_mail, shipping, will_call

Fulfillment methods (baseline mapping)

Method	UI hint (non-normative)	Notes
eticket	mobile ticket icon	QR or barcode delivered digitally
apple_pass	Apple Wallet icon	Add-to-Wallet available
will_call	badge "Will Call"	Pickup at venue
physical_mail	shipping truck icon	Ships to address
shipping	generic shipping badge/icon	Alias/companion to physical_mail

6.9 Developer checklist (quick)

- ☐ `product` has no price/availability; only identity + optional `fulfillment`.
- ☐ `variant` contains no money; omit or keep minimal (empty `{}` for GA tickets).
- ☐ `commercial.price` is a Dinero snapshot; `maxSelectable` present; `feesIncluded` set.
- ☐ Quantity UI clamps **only** to `maxSelectable` (never `limits.*` or `supply.remaining`).

- ☐ Price masked when locked; price hidden when not purchasable; quantity hidden when locked or not purchasable.
 - ☐ Copy comes from `state.messages []` / `context.copyTemplates` (no hardcoded strings).
 - ☐ Fulfillment icons/tooltips driven by `product.fulfillment.methods`; values **MUST** be from the enum (unknown methods are invalid).
 - ☐ Add-on dependencies respected via `relations.parentProductIds` and `matchBehavior`.
 - ☐ Currency consistency: all items' `commercial.price.currency.code` matches `pricing.currency.code`.
- .
- .
- 
- .
- .

7. Messages (Unified)

This section replaces the v0.3 split (`reasonTexts` + `microcopy`). Rows speak through **one channel only**: `state.messages []`. Panel banners live in **one channel only**: `context.panelNotices []` (see §5).

7.1 Normative (contract you can validate)

A. Single row-level channel

- Each item's **only** inline text channel is `state.messages []`.
- The client **MUST NOT** synthesize row text from machine codes (`reasons []`) or hardcoded strings.

B. Message object shape

Each entry in `state.messages []` **MUST** conform to:

```
{
  code: string, // machine code, snake_case (e.g., "sold_out",
  "requires_code")
  text?: string, // fully formatted UI string (already localized)
  params?: Record<string, unknown>, // values to interpolate if text omitted
  placement: "row.under_title" |
             "row.under_price" |
             "row.under_quantity" |
             "row.footer" |
             "row.cta_label", // REQUIRED slot; unrecognized placements MUST be
  ignored
  variant?: "neutral" | "info" | "warning" | "error", // styling hint only; default "info"
  priority?: number // order within the placement; higher renders
  first; default 0
}
```

Rules:

- **placement is required.** If omitted or invalid, the client **MUST** omit that message.
- If **text is present**, the client renders it verbatim (no templating).
- If **text is absent**, the client **MUST** attempt to resolve a template by `code` in `context.copyTemplates []` and interpolate with `params`. Unknown placeholders **MUST** resolve to empty string, not `{placeholder}`.
- If neither `text` nor a matching template is available, the client **MUST** omit the message.
- Messages within the **same placement** are sorted by **descending priority**, then **payload order**.

C. Relationship to axes

- Axis arrays `state.temporal.reasons[]`, `state.supply.reasons[]`, `state.gating.reasons[]`, `state.demand.reasons[]` are **machine evidence**, not UI.
- The server **SHOULD** emit one or more `messages[]` that correspond to the user-visible outcomes of those reasons.
- The client **MUST NOT** translate reason codes to strings.

D. Panel vs row speech

- Panel-level banners **MUST** come **only** from `context.panelNotices[]` (see §5). Row messages **MUST NOT** duplicate panel banners unless explicitly intended by the server.

E. Internationalization

- `messages[].text` and `copyTemplates[]` **MUST** be localized server-side before delivery. Clients **MUST NOT** perform translation.

7.2 Rationale (why this is clean and survivable)

- One channel per layer eliminates precedence wars, z-index fights, and translation drift.
- Codes stay machine-readable (`snake_case`), text stays human-readable (server-owned).
- Interpolation via server templates keeps logic centralized while letting the client render instantly.

7.3 Canonical examples (tiny, valid JSON)

A) Sold out (row status under quantity)

```
"state": {
  "supply": { "status": "none", "reasons": ["sold_out"] },
  "messages": [
    { "code": "sold_out", "text": "Sold Out", "placement": "row.under_quantity", "priority":
100 }
  ]
}
```

B) Low remaining with template interpolation

```
"context": {
  "copyTemplates": [
    { "key": "remaining_low", "template": "Only {count} left!" }
  ]
},
"state": {
  "supply": { "status": "available", "reasons": [] },
  "messages": [
    { "code": "remaining_low", "params": { "count": 3 }, "placement": "row.under_quantity",
"priority": 60 }
  ]
}
```

C) Gated (visible locked) message under title

```
"state": {
  "gating": { "required": true, "satisfied": false, "listingPolicy": "visible_locked",
"reasons": ["requires_code"] },
  "messages": [
```

```
    { "code": "requires_code", "text": "Requires access code", "placement":
      "row.under_title", "variant": "info", "priority": 80 }
    ]
  }
```

D) Before sale with formatted time (server pre-formats text)

```
"state": {
  "temporal": { "phase": "before", "reasons": ["outside_window"] },
  "messages": [
    { "code": "on_sale_at", "text": "On sale Fri 10:00 AM CT", "placement":
      "row.under_title", "variant": "info" }
  ]
}
```

7.4 Quick decision table (when to emit common messages)

Situation	Axis facts (examples)	Suggested <code>messages[]</code> entry	Placement	Variant
Not on sale yet	<code>temporal.phase="before"</code>	<code>{ code:"on_sale_at", text:"On sale Fri10 AM CT" }</code>	<code>row.under_title</code>	info
Sales ended	<code>temporal.phase="after", reasons:["sales_ended"]</code>	<code>{ code:"sales_ended", text:"Sales ended" }</code>	<code>row.under_title</code>	info
Sold out	<code>supply.status="none", reasons:["sold_out"]</code>	<code>{ code:"sold_out", text:"Sold Out" }</code>	<code>row.under_quantity</code>	info
Waitlist offered	<code>supply.status="none", demand.kind="waitlist"</code>	<code>{ code:"waitlist_available", text:"Join the waitlist" }</code>	<code>row.footer</code>	info
Low remaining urgency	<code>supply.status="available", remaining low (server decides)</code>	<code>{ code:"remaining_low", params:{count:N} }</code>	<code>row.under_quantity</code>	warning
Gated (visible locked)	<code>gating.required=true, satisfied=false, listingPolicy="visible_locked"</code>	<code>{ code:"requires_code", text:"Requires access code" }</code>	<code>row.under_title</code>	info
Members badge explanation	(badge via <code>display.badges</code>)	<code>{ code:"members_info", text:"Exclusive to members" }</code>	<code>row.footer</code>	neutral

Note: For **hidden** gated items (`omit_until_unlocked`) there is **no row** to message. Use a **panel** notice (`requires_code`) instead (see §5).

7.5 Invariants & guardrails (REMEMBER)

- **No `reasonTexts` map.** That construct is removed. All row text is `state.messages[]` (or template resolution).
- **Do not backfill.** If the server does not provide `text` or a matching template, the client **MUST** omit the message; it must not invent "Sold out".
- **No countdown math.** If a countdown is desired, the server emits time-boxed text and refreshes payloads; the client does not compute or tick.
- **No cross-placement spill.** The client **MUST NOT** move messages to new placements; it either renders at the declared placement or omits.

- **Variant is cosmetic.** It influences styling only; it never alters purchasability or CTAs.

7.6 Client do / don't (practical)

Do

- Sort messages **per placement** by **priority** desc, then payload order.
- Interpolate templates only when **text** is missing and a template with **key == code** exists.
- Use **params** exactly as provided (no client computation of derived params).
- Keep messages reactive: when a new payload arrives, replace the rendered set.

Don't

- Don't translate reason codes; don't display codes to users.
- Don't render messages with unknown **placement**.
- Don't deduplicate "similar" messages heuristically; render exactly what the server sent (within the placement/priority rules).
- Don't conflate row messages with panel banners.

7.7 Edge cases & tests (acceptance checks)

- **Template fallback works** *Given* `messages: [{ code: "remaining_low", params: { count: 2 }, placement: "row.under_quantity" }]` and `copyTemplates: [{ key: "remaining_low", template: "Only {count} left!" }]` *Expect* the row renders "Only 2 left!" under quantity.
- **Missing template is omitted** *Given* `messages: [{ code: "foo_bar", placement: "row.footer" }]` and no matching template and no **text** *Expect* no message is rendered.
- **Placement required** *Given* `messages: [{ code: "sold_out", text: "Sold Out" }]` (missing **placement**) *Expect* message omitted (client must not guess a location).
- **Priority ordering** *Given* two messages in the same placement with priorities 90 and 50 *Expect* the 90 message renders above 50; within equal priorities, payload order wins.
- **Hidden gated ⇒ no row messages** *Given* a gated item with `listingPolicy="omit_until_unlocked"` and `satisfied=false` *Expect* `no messages []` (item omitted); the hint to enter a code **must** be a panel notice (see §5).
- **Locked row masks price but still messages** *Given* `gating.required=true, satisfied=false, listingPolicy="visible_locked"` and `messages: [{ code: "requires_code", ... }]` *Expect* price masked (per §6) and the message displayed; quantity UI hidden.

7.8 Migration notes (v0.3 → v0.4)

- **Replace `reasonTexts` → `state.messages []`**
 - Before: axis `reasons []` + `reasonTexts` map → client text.
 - After: server emits `messages []` entries with **text** or a **code** resolved via `copyTemplates []`.
- **Replace `microcopy []` → `state.messages []`**
 - Keep the content; move to the unified structure with explicit **placement** and optional **severity/priority**.
- **No client dictionaries.** Any client-side string tables must be removed in favor of server-provided text/templates.
- **Codes stabilize; text flexes.** Keep your existing codes (`sold_out`, `requires_code`) but stop expecting the client to map them.

7.9 Developer checklist (fast audit)

- ☐ Every item that needs inline text has `state.messages []` (not `reasonTexts`, not legacy `microcopy`).

- ☐ Every message has a **valid placement**.
- ☐ If **text** is omitted, there is a **matching copyTemplates[key==code]** or the message is intentionally omitted.
- ☐ When interpolating templates, any unknown **{placeholder}** resolves to **""** (empty string).
- ☐ Messages render in **priority order** within their placement.
- ☐ No client code translates machine codes; no hardcoded "Sold Out", "Requires access code", etc.

Up next, we can lock **§8 Rendering Composition** (truth tables for row presentation, purchasability, CTA resolution) or **§8 Gating & Unlock Flow** (pre-unlock vs post-unlock payloads and zero-leak rules), depending on what you want to ship first.

8. Rendering Composition (*Derived Atoms Only*)

What this section does: turns server facts (§§3–6) into concrete UI: row presentation, purchasability, quantity/price visibility, and CTA. **Discipline:** derive; don't decide. **No banners, no strings, no math** unless provided by payload (see §§4–5 & §13).

8.1 Normative — Derived flags (pure functions over the contract)

The client **MUST** derive, per `items[]` element:

- **Row presentation** `presentation: "normal" | "locked"`
 - **"locked"** iff `state.gating.required === true AND state.gating.satisfied === false AND state.gating.listingPolicy === "visible_locked"`.
 - Otherwise **"normal"**.
 - Items with `listingPolicy="omit_until_unlocked"` **never** arrive in `items[]` (server omission; see §3.3).
- **Purchasable boolean** `isPurchasable: boolean` `isPurchasable = (state.temporal.phase === "during") && (state.supply.status === "available") && (!state.gating.required || state.gating.satisfied) && (commercial.maxSelectable > 0)`
- **Quantity UI** `quantityUI: "hidden" | "select" | "stepper"`
 - **Hidden** unless `presentation === "normal" && isPurchasable && commercial.maxSelectable > 0`.
 - If shown:
 - **"select"** when `commercial.maxSelectable === 1` (single-tap "Add" or a 1-step selector).
 - **"stepper"** when `commercial.maxSelectable > 1`.
- **Price UI** `priceUI: "hidden" | "masked" | "shown"`
 - **"masked"** when `presentation === "locked"`.
 - **"shown"** iff `presentation === "normal" AND isPurchasable === true`.
 - Otherwise **"hidden"**.
 - **Rationale:** This strict policy prevents psychological "tease/anchor" effects. Showing a price for something you can't buy creates confusion and frustration ("Why display it if I can't select it?"). Hiding price until purchasable maintains honest, actionable signals. Locked rows use masking (not hiding) to preserve gating UX without leaking price to unauthorized users. See §6 for full policy discussion.
- **CTA** `cta.kind: "quantity" | "waitlist" | "notify" | "none"` with `cta.enabled: boolean`

- **Gate precedence:** if `presentation === "locked"`, `cta.kind="none"`.
- Else, evaluate in order:
 1. If `isPurchasable` → `cta.kind="quantity"` and `cta.enabled = (commercial.maxSelectable > 0)`.
 2. Else if `state.supply.status === "none"` and `state.demand.kind === "waitlist"` → `cta.kind="waitlist"` (enabled).
 3. Else if `state.temporal.phase === "before"` and `state.demand.kind === "notify_me"` → `cta.kind="notify"` (enabled).
 4. Else → `cta.kind="none"`.

- **CTA label text**

- The **button label** (when textual) **MUST** come from the payload:
 - Prefer a row message with `placement: "row.cta_label"` and `text` (or `code` + `copyTemplates`, see §5.4).
 - The client **MUST NOT** hardcode "Join Waitlist" / "Notify Me" / "Add" strings.
- Quantity controls MAY be icon-only; if text is used, it must also come from payload copy.

No "Request" CTA: approval/requests are **out of scope** for v0.4 (see §13).

8.1.a Row decision cheatsheet (developer aid)

```
function getRowPresentation(state: ItemState): "normal" | "locked" {
  const g = state.gating;
  if (g.required && !g.satisfied && g.listingPolicy === "visible_locked")
    return "locked";
  return "normal"; // Items with omit_until_unlocked never arrive in items[]
}

function isPurchasable(state: ItemState, maxSelectable: number): boolean {
  return (
    state.temporal.phase === "during" &&
    state.supply.status === "available" &&
    (!state.gating.required || state.gating.satisfied) &&
    maxSelectable > 0
  );
}

type CTA = {
  kind: "quantity" | "waitlist" | "notify" | "none";
  enabled?: boolean;
};

function getRowCTA(state: ItemState, maxSelectable: number): CTA {
  if (getRowPresentation(state) === "locked") return { kind: "none" };
  if (isPurchasable(state, maxSelectable))
    return { kind: "quantity", enabled: maxSelectable > 0 };
  if (state.supply.status === "none" && state.demand.kind === "waitlist")
    return { kind: "waitlist", enabled: true };
  if (state.temporal.phase === "before" && state.demand.kind === "notify_me")
    return { kind: "notify", enabled: true };
  return { kind: "none" };
}
```

Use payload-provided strings for labels (see §5). This snippet mirrors the truth tables above; it is illustrative only.

8.2 Decision tables (mechanical mapping)

A) Presentation

Condition	presentation
<code>gating.required && !gating.satisfied && listingPolicy="visible_locked"</code>	locked
Otherwise	normal
<i>(Items omitted by server—omit_until_unlocked—do not appear and thus have no presentation.)</i>	

B) Purchasable

temporal.phase	supply.status	Gate satisfied?	maxSelectable	isPurchasable
<code>"during"</code>	<code>"available"</code>	<code>(!required \\ \\ satisfied)</code>	<code>> 0</code>	<code>true</code>
Any other combo	Any other combo	Any	Any	<code>false</code>

C) CTA resolution (in order; first match wins)

Condition	cta.kind	Notes
<code>presentation === "locked"</code>	<code>none</code>	Gate precedence; see §3.5
<code>isPurchasable</code>	<code>quantity</code>	Quantity control shown if <code>commercial.maxSelectable > 0</code>
<code>supply.status === "none" AND demand.kind === "waitlist"</code>	<code>waitlist</code>	Label from <code>messages[]</code> or <code>copyTemplates</code> ; action handler is app-level
<code>temporal.phase === "before" AND demand.kind === "notify_me"</code>	<code>notify</code>	Label from payload
Otherwise	<code>none</code>	Purely informational row

D) Quantity & Price visibility

presentation	isPurchasable	maxSelectable	quantityUI	priceUI
<code>locked</code>	—	—	<code>hidden</code>	<code>masked</code>
<code>normal</code>	<code>false</code>	<code>any</code>	<code>hidden</code>	<code>hidden</code>
<code>normal</code>	<code>true</code>	<code>1</code>	<code>select</code>	<code>shown</code>
<code>normal</code>	<code>true</code>	<code>> 1</code>	<code>stepper</code>	<code>shown</code>

Remember: `maxSelectable` is **authoritative**. Never recompute min/max from counts/limits (see §13).

8.3 Interaction & data refresh (authoritative loop)

- **All interactions** (quantity changes, unlock attempts, demand CTAs) **MUST** call the server and rely on a refreshed payload. The client **MUST** re-derive presentation from the new data; no local toggling of truth.
- **Access code:** submit → server validates → refreshed `items[]` (locked → unlocked or omitted → included), `gatingSummary` updated. **Client MUST NOT** locally flip `gating.satisfied`.
- **Quantity change:** client sends proposed selection → server recomputes clamps and pricing → client renders updated `commercial.maxSelectable` + `pricing`. **Client MUST NOT** compute totals/fees/discounts.

8.4 Rollups (panel-level, no copy)

The client **MAY** derive rollups **only** for layout/controls, never to show banners:

- `selectionValid` — computed from `context.orderRules` (min types/tickets) and current selection; gates the bottom “Continue” button (out of this contract’s copy scope; string comes from app).

- `allVisibleSoldOut` — `items.every(i => i.state.supply.status === "none")`. Used **only** with `effectivePrefs.showTypeListWhenSoldOut === false` to collapse the list. **MUST NOT** generate a "Sold out" banner; show one only if `panelNotices[]` contains it.
- `anyLockedVisible` — `items.some(i => presentation === "locked")`. Could be used to surface an **app-level** unlock affordance (UI only). **Never** invent text.

8.5 Examples (compact, valid JSON + expected renders)

A) Waitlist CTA

```
{
  "state": {
    "temporal": { "phase": "during", "reasons": [] },
    "supply": { "status": "none", "reasons": ["sold_out"] },
    "gating": {
      "required": false,
      "satisfied": true,
      "listingPolicy": "omit_until_unlocked",
      "reasons": []
    },
  },
  "demand": { "kind": "waitlist", "reasons": ["waitlist_available"] },
  "messages": [
    {
      "code": "sold_out",
      "text": "Sold Out",
      "placement": "row.under_quantity"
    },
    {
      "code": "waitlist_cta",
      "text": "Join Waitlist",
      "placement": "row.cta_label"
    }
  ]
},
  "commercial": {
    "price": {
      "amount": 12000,
      "currency": { "code": "USD", "base": 10, "exponent": 2 },
      "scale": 2
    },
    "feesIncluded": false,
    "maxSelectable": 0
  }
}
```

- **Derived:** `presentation="normal"`, `isPurchasable=false`, `quantityUI=hidden`, `priceUI=hidden`, `cta.kind="waitlist"` with label **"Join Waitlist"** from the row message.

B) Notify-me CTA (before window)

```
{
  "state": {
    "temporal": { "phase": "before", "reasons": ["outside_window"] },
    "supply": { "status": "available", "reasons": [] },
    "gating": {
      "required": false,
      "satisfied": true,
      "listingPolicy": "omit_until_unlocked",
      "reasons": []
    },
  },
  "commercial": {
    "price": {
      "amount": 12000,
      "currency": { "code": "USD", "base": 10, "exponent": 2 },
      "scale": 2
    },
    "feesIncluded": false,
    "maxSelectable": 0
  }
}
```

```

    },
    "demand": { "kind": "notify_me", "reasons": [] },
    "messages": [
      {
        "code": "outside_window",
        "text": "On sale Friday 10:00 AM CT",
        "placement": "row.under_title"
      },
      {
        "code": "notify_cta",
        "text": "Notify Me",
        "placement": "row.cta_label"
      }
    ]
  },
  "commercial": {
    "price": {
      "amount": 5000,
      "currency": { "code": "USD", "base": 10, "exponent": 2 },
      "scale": 2
    },
    "feesIncluded": false,
    "maxSelectable": 0
  }
}

```

- **Derived:** `presentation="normal"`, `isPurchasable=false`, `cta.kind="notify"` with label **"Notify Me"**.

C) Visible locked (price masked)

```

{
  "state": {
    "temporal": { "phase": "during", "reasons": [] },
    "supply": { "status": "available", "reasons": [] },
    "gating": {
      "required": true,
      "satisfied": false,
      "listingPolicy": "visible_locked",
      "reasons": ["requires_code"]
    },
  },
  "demand": { "kind": "none", "reasons": [] },
  "messages": [
    {
      "code": "requires_code",
      "text": "Requires access code",
      "placement": "row.under_title"
    }
  ]
},
"commercial": {
  "price": {
    "amount": 9000,
    "currency": { "code": "USD", "base": 10, "exponent": 2 },
    "scale": 2
  },
  "feesIncluded": false,
  "maxSelectable": 0
}
}

```

- **Derived:** `presentation="locked"`, `priceUI="masked"`, `quantityUI=hidden`, `cta="none"`.

8.6 Pseudocode (reference; keep it boring)

```
function derivePresentation(item) {
  const g = item.state.gating;
  return g.required && !g.satisfied && g.listingPolicy === "visible_locked"
    ? "locked"
    : "normal";
}

function derivePurchasable(item) {
  const s = item.state;
  const max = item.commercial.maxSelectable ?? 0;
  return (
    s.temporal.phase === "during" &&
    s.supply.status === "available" &&
    (!s.gating.required || s.gating.satisfied) &&
    max > 0
  );
}

function deriveQuantityUI(item, pres, purch) {
  if (pres !== "normal" || !purch) return "hidden";
  const max = item.commercial.maxSelectable ?? 0;
  if (max <= 0) return "hidden";
  return max === 1 ? "select" : "stepper";
}

function derivePriceUI(pres, purch) {
  if (pres === "locked") return "masked";
  return purch ? "shown" : "hidden";
}

function deriveCTA(item, pres, purch) {
  if (pres === "locked") return { kind: "none", enabled: false };
  if (purch)
    return { kind: "quantity", enabled: item.commercial.maxSelectable > 0 };
  const s = item.state;
  if (s.supply.status === "none" && s.demand.kind === "waitlist")
    return { kind: "waitlist", enabled: true };
  if (s.temporal.phase === "before" && s.demand.kind === "notify_me")
    return { kind: "notify", enabled: true };
  return { kind: "none", enabled: false };
}
```

Strings: Any visible label for these controls must be supplied in `state.messages[]` (e.g., `placement: "row.cta_label"`) or resolvable via `copyTemplates` (§5.4). No hardcoded UI text.

8.7 Edge cases & tests (acceptance)

- **Gate precedence over demand** Given `gating.required=true && !satisfied && listingPolicy="visible_locked"` and `demand.kind="waitlist"` Expect `presentation="locked"`, `cta="none"`, price masked.
- **Clamp beats counts** Given `temporal.phase="during"`, `supply.status="available"`, gating satisfied or not required, and `commercial.maxSelectable=0` Expect `isPurchasable=false`, `quantityUI="hidden"`, `priceUI="hidden"`, `cta.kind="quantity"` not selected (no quantity CTA).
- **Unknown supply** Given `temporal.phase="during"` and `supply.status="unknown"` Expect `isPurchasable=false`, `quantityUI="hidden"`, `priceUI="hidden"`. If the server wants copy, it supplies a message

(e.g., "Availability updating...").

- **Notify vs. waitlist priority** Given `temporal.phase="before"`, `demand.kind="notify_me"`, and later a refresh with `supply.status="none"` and `demand.kind="waitlist"` Expect CTA moves from `notify` to `waitlist` after refresh (no client heuristics).
- **No auto banners** Given `allVisibleSoldOut === true` and `context.panelNotices=[]` Expect no banner is invented; list may collapse only if `effectivePrefs.showTypeListWhenSoldOut=false`.
- **CTA labels from payload** Given `cta.kind="waitlist"` and no `row.cta_label` message or template Expect the control renders icon-only or as an affordance with no text; client **MUST NOT** inject a default string.

8.8 Client checklist (quick)

- ☐ Compute `presentation`, `isPurchasable`, `quantityUI`, `priceUI`, `cta` **only** from server fields.
- ☐ Do **not** invent banners or strings. Pull CTA labels via `messages[]/copyTemplates`.
- ☐ Respect `commercial.maxSelectable` as the **only** clamp.
- ☐ Mask price on locked rows; hide price when not purchasable.
- ☐ If `maxSelectable=0`, the row is not purchasable; price is hidden and quantity is hidden.
- ☐ When interpolating templates, any unknown `{placeholder}` resolves to `""` (empty string).
- ☐ Collapse sold-out lists only per `effectivePrefs.showTypeListWhenSoldOut`.
- ☐ On any interaction, call server → replace payload → re-derive.

9. Gating & Unlock (No Leakage)

Purpose: Define how access-controlled items are represented, hidden, revealed, and rendered—without leaking SKU identity, price, or counts before authorization. This section is **normative** and composes with §§3.3, 5.3/5.3a, 7, 8, and 13.

9.1 Normative

A. Authoritative fields & sendability

- Each item’s gating lives in `state.gating: { required: boolean, satisfied: boolean, listingPolicy: "omit_until_unlocked"|"visible_locked", reasons[], requirements?[] }`. See §3.3 for the full axis shape.
- **Default sendability:** If `required=true` and `satisfied=false` and `listingPolicy` is **absent**, treat it as `"omit_until_unlocked"`.
- **Omit policy ("omit_until_unlocked"):**
 - When unsatisfied, the item **MUST NOT** appear in `items[]`.
 - The **only** allowed hint is `context.gatingSummary.hasHiddenGatedItems: boolean`. No placeholders, no counts, no names.
- **Visible-locked policy ("visible_locked"):**
 - When unsatisfied, the item **MUST** be present in `items[]` but rendered as **locked**.
 - While locked, price **MUST** be **masked**, quantity UI **MUST** be **hidden**, and row CTA **MUST** be **none** (see §8 decision tables).
- **Unlock transition:** Upon successful unlock, the server **MUST**:

- Include previously omitted items in `items[]` or flip visible-locked rows to `gating.satisfied=true`.
- Recompute `commercial.maxSelectable` and `pricing` as needed.
- Update `context.gatingSummary.hasHiddenGatedItems` accordingly (see §9.4).

B. GatingSummary (panel-level hints)

- `context.gatingSummary` **MUST** be present **iff** gating exists for the event.
- It **MUST** include:
 - `hasHiddenGatedItems: boolean` — **true** **iff** there exists at least one **omitted** gated item that currently has purchasable stock (per §8 purchasability) or could become available during the current session (server decision). *Do not set true for items that are omitted but permanently unavailable.*
- It **MAY** include:
 - `hasAccessCode: boolean` — feature presence hint only.
- Clients **MUST NOT** infer anything beyond this boolean (no SKU counts, no names, no ranges).

C. Unlock UX derivation (panel-level, not notices)

- The access-code UI (the `AccessCodeCTA`) is **derived**, not configured:
 - It **MUST** appear when **either**:
 - `context.gatingSummary.hasHiddenGatedItems === true`, or
 - Any visible item is locked: `gating.required && !gating.satisfied && listingPolicy="visible_locked"`. (*Matches §5.3a.*)
- Panel banners about codes (e.g., instructional "Enter access code...") **MUST** come from `context.panelNotices[]` (optional). The `AccessCodeCTA` itself is **not** a notice (see §5.3/§5.3a).

D. No leakage & precedence

- Clients **MUST NOT**:
 - Render row placeholders for omitted items.
 - Display or log codes, tokens, or any derived inference about hidden SKUs.
 - Surface demand CTAs (waitlist/notify) for an item while it is gated and unsatisfied. **Gating precedence applies.** (See §3.5 and §8.)
- Locked rows (`visible_locked`) **MUST** mask price and hide quantity controls regardless of `commercial.price` presence (§6).

Visual treatment of locked rows (implementation guidance):

- Locked rows typically appear **greyed-out** with a **lock icon** (visual indicator).
- Price may be replaced with placeholder text like "Locked" or "—" (not just blank).
- The row should clearly communicate its locked state through both visual styling and the message from `state.messages[]`.

Unlock confirmation pattern:

- When a gated item unlocks but has `supply.status="none"`, the item **MUST** still appear (as a disabled/sold-out row) to confirm the code worked.
- This prevents user confusion ("Did my code work?") by showing the item exists even when unavailable.
- The confirmation differs by `listingPolicy`: omitted items appear for the first time; `visible_locked` items transition from locked to unlocked-but-unavailable.

E. Validation & error handling

- Access-code validation, rate limiting, and unlock token issuance happen **server-side**. The client **MUST NOT** validate or rate-limit locally.
- On invalid/expired code, the server returns a payload state that conveys errors via:
 - A panel-level notice (e.g., `{ code:"code_invalid", variant:"error", text:"Invalid access code" }`), **or**
 - A row message for a **still-locked** row (e.g., `{ code:"code_invalid", placement:"row.under_title", variant:"error" }`). The client renders **only** what the payload says (no local strings), per §7.

F. Requirements metadata (optional, for copy only)

- `gating.requirements[]` **MAY** include structured facts (e.g., `{ kind:"unlock_code", satisfied:false, validWindow, limit }`).
- Clients **MUST** treat `requirements[]` as **explanatory** metadata only (copy and tooling). They **MUST NOT** derive purchasability or sendability from it. `gating.satisfied` remains authoritative.

G. State replacement

- After any unlock attempt, the client **MUST** replace local derived state from the **new** payload (no local flips, no predictions). See §8.3 and §13.

H. Unlock flow (user journey)

Understanding the complete unlock sequence helps implementers handle edge cases correctly:

Pre-unlock state (omit_until_unlocked):

- User loads panel → server sends **empty or partial** `items[]` (public items only, if any)
- `context.gatingSummary.hasHiddenGatedItems=true` (the hint)
- `context.panelNotices[]` includes unlock prompt: `{ code:"requires_code", text:"Enter access code to view tickets" }`
- Client displays AccessCodeCTA prominently (input + "Apply Code" button)
- User sees either: (a) no items if all are gated, or (b) public items marked sold out if those exist

Unlock attempt:

- User enters code → client POSTs to server `/panel/unlock` (or similar endpoint)
- Server validates code server-side (checks signature, expiry, usage limits, etc.)

Post-unlock (success):

- Server responds with updated payload:
 - Previously omitted items now appear in `items[]`
 - Those items have `gating.satisfied=true`
 - `context.gatingSummary.hasHiddenGatedItems` updates (may stay `true` if partial unlock)
 - Panel notice may change or be removed
- Client replaces state → atoms re-derive → React re-renders
- User sees: newly visible items with `presentation="normal"` (if available) or sold-out status (if `supply.status="none"`)

Post-unlock (error):

- Server responds with error payload or notice:
 - `context.panelNotices[]` includes `{ code:"code_invalid", variant:"error", text:"Invalid access code. Please try again." }`
 - No items are unlocked; state remains unchanged
- Client displays error banner (red, high priority)
- User can retry with correct code

Critical edge case: Public sold out + hidden gated:

- **Scenario:** All visible items have `supply.status="none"` but `gatingSummary.hasHiddenGatedItems=true`
- **Client behavior:** Do **NOT** show "Event Sold Out" final state; show "Enter access code" prompt instead
- **Rationale:** Prevents misleading users into thinking nothing is available when gated inventory exists
- **After unlock:** If gated items are also sold out, **then** show "Event Sold Out" (but user got confirmation their code worked)

9.2 Rationale

- **Zero-leak default:** Scrapers love disabled rows. `omit_until_unlocked` eliminates name/price leakage by not sending the row at all.
- **Two explicit modes:** Marketing sometimes needs a tease. `visible_locked` is the opt-in tease, with strict masking rules.
- **One unlock surface:** Keeping the AccessCodeCTA panel-level avoids per-row code UX complexity and aligns with `gatingSummary`.
- **Server as oracle:** The server owns the unlock lifecycle, stock truth, and clamp. The client merely re-renders on new facts.

9.3 Decision tables

A) Sendability & presentation

required	satisfied	listingPolicy	In items[]?	Row presentation	Price UI	Quantity UI	Row CTA
false	—	—	✓	normal	per \$8	per \$8	per \$8
true	false	omit_until_unlocked	✗	—	—	—	—
true	false	visible_locked	✓	locked	masked	hidden	none
true	true	(either)	✓	normal	per \$8	per \$8	per \$8

B) When to show the AccessCodeCTA (panel-level)

Condition	Show AccessCodeCTA?
<code>context.gatingSummary.hasHiddenGatedItems === true</code>	✓
Any visible item locked (<code>required && !satisfied && listingPolicy="visible_locked"</code>)	✓
Neither of the above	✗

Instructional banners about codes are optional notices (`context.panelNotices[]`), separate from the AccessCodeCTA.

C) Demand precedence (no leakage)

Locked state? (<code>required && !satisfied</code>)	demand.kind	Row CTA
true	"waitlist"/"notify_me"	none
false	"waitlist"	join_waitlist (per \$8)
false	"notify_me"	notify (per \$8)

9.4 Server obligations (for clarity)

- **hasHiddenGatedItems truthiness:** Set `true` when **any omitted** gated SKU is meaningfully unlockable (e.g., has stock or may open during the current sale), `false` otherwise. Do not thrash this flag for transient, non-purchasable states.
- **After unlock:**
 - Include newly unlocked items (previously omitted), or flip `satisfied=true` on formerly locked items.
 - Recompute `commercial.maxSelectable` and all `pricing`.

- Adjust `hasHiddenGatedItems` to reflect remaining hidden stock (it may stay `true` if the code only unlocked a subset).
 - **Error states:** Convey invalid/expired/limit-exhausted outcomes via messages/notices; never rely on client-invented copy.
-

9.5 Examples (tiny, canonical)

A) Hidden until unlock (default)

```
// Payload (pre-unlock): the gated item is omitted; only a hint + optional notice
"context": {
  "gatingSummary": { "hasHiddenGatedItems": true },
  "panelNotices": [
    { "code": "requires_code", "variant": "info", "text": "Enter access code to view tickets", "priority": 90 }
  ]
},
"items": [
  // ... public items only
]
```

B) Visible locked (tease)

```
{
  "product": {
    "id": "prod_members",
    "name": "Members Presale",
    "type": "ticket"
  },
  "state": {
    "temporal": { "phase": "during", "reasons": [] },
    "supply": { "status": "available", "reasons": [] },
    "gating": {
      "required": true,
      "satisfied": false,
      "listingPolicy": "visible_locked",
      "reasons": ["requires_code"]
    },
    "demand": { "kind": "none", "reasons": [] },
    "messages": [
      {
        "code": "requires_code",
        "text": "Requires access code",
        "placement": "row.under_title",
        "variant": "info",
        "priority": 80
      }
    ]
  },
  "commercial": {
    "price": {
      "amount": 9000,
      "currency": { "code": "USD", "base": 10, "exponent": 2 },
      "scale": 2
    },
    "feesIncluded": false,
    "maxSelectable": 0
  },
  "display": { "badges": ["Members"] }
}
```

C) After successful unlock

```
// Same item after server validation
"state": {
  "temporal": { "phase": "during", "reasons": [] },
  "supply": { "status": "available", "reasons": [] },
  "gating": { "required": true, "satisfied": true, "listingPolicy": "visible_locked",
"reasons": [] },
  "demand": { "kind": "none", "reasons": [] },
  "messages": []
},
"commercial": { "maxSelectable": 2 }
```

D) Invalid code (panel-level)

```
"context": {
  "gatingSummary": { "hasHiddenGatedItems": true },
  "panelNotices": [
    { "code": "code_invalid", "variant": "error", "text": "Invalid access code. Please try
again.", "priority": 100 }
  ]
}
```

E) Requirements metadata (explanatory only)

```
"gating": {
  "required": true,
  "satisfied": false,
  "listingPolicy": "visible_locked",
  "requirements": [{
    "kind": "unlock_code",
    "satisfied": false,
    "validWindow": { "startsAt": "2025-10-22T00:00:00Z", "endsAt": "2025-10-25T23:59:59Z" },
    "limit": { "maxUses": 100, "usesRemaining": 23 }
  }],
  "reasons": ["requires_code"]
}
```

F) Public sold out + hidden gated (no "sold out" dead-end)

```
"context": {
  "gatingSummary": { "hasHiddenGatedItems": true },
  "panelNotices": [
    { "code": "requires_code", "variant": "info", "text": "Enter access code to view
tickets", "priority": 90 }
  ]
},
"items": [
  // public items with supply.status="none"
  {
    "product": { "id": "prod_ga", "name": "General Admission", "type": "ticket" },
    "state": {
      "supply": { "status": "none", "reasons": ["sold_out"] },

```

```
    "messages": [
      { "code": "sold_out", "text": "Sold Out", "placement": "row.under_quantity",
        "priority": 100 }
    ],
    "commercial": { "maxSelectable": 0 }
  }
]
// Panel SHOULD NOT collapse to final sold-out state; show AccessCodeCTA instead.
// User sees: sold-out public items + prominent "Enter access code" prompt (not "Event Sold Out" finale).
```

G) Reason code usage guidance

Gating-related codes and their typical presentation:

- **requires_code** (axis reason + panel notice + row message)
 - As axis reason: `gating.reasons: ["requires_code"]`
 - As panel notice: Instructional banner "Enter access code to view tickets" (info variant)
 - As row message: Small text "Requires access code" under locked item title (info variant)
 - Visual: Lock icon, greyed-out row styling
- **code_invalid** (panel notice, error state)
 - Shown after failed unlock attempt
 - Panel notice: "Invalid access code. Please try again." (error variant, red banner)
 - High priority (displays prominently)
 - Client does NOT generate this; server includes it in error response payload
- **unlocked** (row message, optional celebration)
 - May appear after successful unlock: "Unlocked with your code" (neutral variant)
 - Server may omit this if the unlocked state is obvious from newly visible items
 - Purpose: explicit confirmation feedback

Visual treatment by code:

Code	Icon	Styling	Typical Placement	Variant
requires_code	Lock	Grey/muted	Row: under title; Panel: top	info
code_invalid	Alert	Red border	Panel: top banner	error
unlocked	Check	Normal	Row: under title (optional)	neutral

9.6 Tests (acceptance checks)

- **Omit enforcement** *Given* `required=true, satisfied=false, listingPolicy="omit_until_unlocked"` *Expect* item **absent** from `items[]`; `context.gatingSummary.hasHiddenGatedItems=true` **iff** omitted stock is meaningfully available.
- **Locked rendering** *Given* `required=true, satisfied=false, listingPolicy="visible_locked"` *Expect* row `presentation="locked"`, **price masked, quantity hidden, CTA none**; render any messages per §7.
- **Unlock transition** *When* server validates code → next payload has either newly included items or flipped `satisfied=true` *Expect* client to **replace** state and re-derive (§8). No local flips.
- **Gating precedence over demand** *Given* locked row and `demand.kind="waitlist"` *Expect* **no** waitlist CTA until `satisfied=true`. (Row CTA remains none.)

- **AccessCodeCTA derivation** Given `hasHiddenGatedItems=true` or any visible locked row Expect AccessCodeCTA present (panel-level). Given neither condition Expect no AccessCodeCTA.
- **Error surfacing** Given invalid code Expect error displayed **only** via payload text (panel notice or row message); no client-invented copy.
- **No leakage** Given omitted items Expect no SKU placeholders, names, or prices in UI or logs; only the boolean hint in `gatingSummary`.
- **Price masking** Given `presentation="locked"` Expect price UI **masked** even if `commercial.price` is present.
- **Partial unlock** Given code unlocks a subset of gated SKUs Expect newly unlocked items included; `hasHiddenGatedItems` may remain `true` if others remain omitted.
- **Public sold out + hidden gated (no "sold out" dead-end)** Given all visible items have `supply.status="none"` AND `gatingSummary.hasHiddenGatedItems=true` Expect client **MUST NOT** display final "Event Sold Out" state; **MUST** show AccessCodeCTA and unlock prompt notice instead. *Rationale* Prevents misleading users when gated inventory still exists.
- **Unlocked but sold out (confirmation feedback)** Given user successfully unlocks code but unlocked item has `supply.status="none"` Expect item appears as disabled/sold-out row (not omitted) to confirm code validation succeeded. *User benefit* Clear feedback: "Your code worked, but this ticket sold out."

9.7 Developer checklist (quick)

Core contract compliance:

- ☐ Enforce sendability: omit vs visible-locked exactly as specified by `listingPolicy`.
- ☐ Show AccessCodeCTA when `hasHiddenGatedItems` is true or any visible row is locked.
- ☐ Mask price + hide quantity on locked rows; no row CTA while locked.
- ☐ Do not render demand CTAs (waitlist/notify) for locked rows.
- ☐ Never invent copy; render `messages[]` / `panelNotices[]` only (templates via `copyTemplates`).
- ☐ After unlock attempt, re-render from **new** payload; do not toggle `satisfied` locally.
- ☐ Do not log codes/tokens (see §13 Security guardrails).

Visual implementation:

- ☐ Locked rows: greyed-out styling + lock icon + price placeholder ("Locked" or "—").
- ☐ Reason code display: `requires_code` shows lock icon; `code_invalid` shows red error banner.
- ☐ Unlock confirmation: unlocked items that are sold out still appear (disabled) to confirm code worked.

Edge case handling:

- ☐ Public sold out + `hasHiddenGatedItems=true`: show unlock prompt, NOT "Event Sold Out" finale.
- ☐ Post-unlock sold out: render item as disabled row with sold-out message (confirmation feedback).
- ☐ Invalid code: display server-provided error notice prominently; allow retry.
- ☐ Partial unlock: keep AccessCodeCTA visible if `hasHiddenGatedItems` remains `true`.

9.8 Consistency audit (cross-section)

- **§3.3 (Gating axis):** Listing policy, reasons, and `requirements[]` semantics are unchanged and respected here. ✓
- **§5.3/5.3a (Prefs & CTAs):** AccessCodeCTA derivation and panel-notice separation match the rules above. ✓
- **§7 (Messages):** All user copy comes from `messages[]`/`panelNotices[]`/`copyTemplates`; no `reasonTexts` usage. ✓
- **§8 (Rendering):** Presentation, price masking, CTA resolution (including precedence) align with the decision tables here. ✓
- **§13 (Guardrails):** No schedule/availability/clamp math; no leakage; no local unlock; no per-row payment-plan badges. ✓

With these rules, gated flows stay secure by architecture: hidden items don't exist client-side until the server says otherwise, and visible locks never leak price or enable alternate CTAs prematurely.

10. Relations & Add-ons (*selection vs ownership*; *matchBehavior*)

What this section does: defines how a row can **depend on** other rows (parents), and how the client presents and validates those dependencies **without** inventing business rules. This is **not** a new axis; it's metadata used to constrain selection and explain behavior.

Common use cases: Add-ons like parking passes, meal vouchers, or merchandise that should only be purchasable alongside a main ticket. For example, a parking pass might be limited to one per order regardless of ticket quantity (*per_order*), while a meal voucher might be available for each ticket purchased (*per_ticket*).

Key principle: The server is the final gatekeeper. It validates all parent-child relationships on submission, so even if a user bypassed the UI, the server's *maxSelectable* and order validation would catch invalid selections.

10.1 Normative (contract you can validate)

A. Shape

- An item **MAY** include *relations*:

```
relations?: {
  parentProductIds?: string[];           // IDs this item depends on
  matchBehavior?: "per_ticket" | "per_order";
}
```

- If *relations.parentProductIds* is **absent or empty** \Rightarrow the item is **standalone** (not an add-on).
- If *relations.parentProductIds* is **present and non-empty** \Rightarrow the item is an **add-on** to those **parent** products (IDs must match *items[].product.id* present or potentially present in this panel).

B. Semantics (ownership & selection)

- Add-on definition:** Any item with *parentProductIds[]* is an **add-on**. "Add-on" is a **relationship**, not a product type. *product.type* stays *"ticket" | "digital" | "physical"*.
- Parent set:** The **parent set** for an add-on is the multiset sum of selected quantities across **all** *parentProductIds* that are currently visible/unlocked.
 - Let *parentSelectedCount* = \sum *selection[parentId]* across the listed IDs (client selection state).
- matchBehavior meaning:**
 - "per_ticket"* \Rightarrow An add-on can be selected **at most** one-for-one with **the current** *parentSelectedCount*.
 - "per_order"* \Rightarrow An add-on is limited **at the order level** regardless of parent quantity (typically one per order or a small cap).
- Practical examples:**
 - "per_ticket"*: A "Fast Pass" add-on for each ticket. If a user selects 3 VIP tickets, the client should allow up to 3 Fast Passes to match (enforced via server-updated *maxSelectable*).
 - "per_order"*: A "Parking Pass" limited to one per order. No matter how many GA or VIP tickets are selected, only one Parking Pass can be added. The UI would disable adding a second once *maxSelectable* is reached.

- **Default:** If `parentProductIds []` exists and `matchBehavior` is **omitted**, clients **MUST** treat it as `"per_order"` for compatibility.

C. Server responsibilities

- **Authoritative clamp:** `commercial.maxSelectable` remains the **authoritative cap** for every item (see §§4, 6, 13). The server **MUST** recompute `maxSelectable` for add-ons on every payload refresh to reflect:
 - stock/holds/limits **and**
 - the current selection of their parents (per `matchBehavior`).
- **Parent-absent state:** When `parentSelectedCount === 0`, the server **SHOULD** send `maxSelectable=0` for add-ons; on parent selection changes, it **MUST** update `maxSelectable` accordingly in the next payload.
- **Gating zero-leak:** If **all** parents for an add-on are omitted due to gating (`omit_until_unlocked`) or are otherwise not sendable, the add-on **MUST** also be omitted (or sent with `maxSelectable=0` and its own gating), so that parent existence is not leaked.
- **Multi-parent:** When multiple parents are listed, the server's clamp **MUST** reflect the **sum** of all selected parents for `"per_ticket"`, or the order cap for `"per_order"` (still $\leq \text{maxSelectable}$).

D. Client responsibilities

- **Never invent caps:** The client **MUST NOT** compute or enforce numeric quantity caps besides `commercial.maxSelectable`. *Corollary:* The client **MUST** initiate a payload refresh whenever parent selection changes so the server can update dependent `maxSelectable` values.
- **Visibility & purchasability:** Add-ons follow the **same** axes rules as any item (temporal, supply, gating, demand). If `maxSelectable=0` or the derived `isPurchasable` (§8) is false, quantity UI is hidden.
- **Parent presence affordance:** When an add-on has `parentProductIds []` and `maxSelectable=0`, the client **MAY** render explanatory copy using server-provided strings (e.g., `clientCopy.addon_requires_parent`) but **MUST NOT** inject local prose.
- **Reduce on refresh:** If a payload refresh lowers `maxSelectable` below the user's current add-on selection, the client **MUST** clamp the selection down to the new `maxSelectable` (see §8 and selection family) and reflect it immediately.
- **No leakage:** Clients **MUST NOT** infer or surface any information about parents that are omitted due to gating. No placeholders, counts, or prices.
- **UI presentation flexibility:** The contract doesn't mandate how add-ons are displayed. Some UIs might nest add-ons under parent tickets; others list them separately with explanatory text (e.g., "Requires a GA ticket"). What matters is that the dependency is clear to the user through server-provided copy and badges.

E. Interactions with other fields

- **Limits & caps:** `limits.perOrder/limits.perUser` remain informational; `maxSelectable` already reflects them. Clients **MUST NOT** enforce `limits.*` directly.
- **Demand:** Add-ons **may** use `demand.kind` (`waitlist`, `notify_me`) like any item. Gating precedence (§3.5) still applies.
- **Sectioning:** The server **SHOULD** place add-ons under a distinct section via `display.sectionId` (e.g., `"add_ons"`), but clients **MUST** render them wherever they are assigned.

10.2 Rationale (why this shape works)

- Keeps **one clamp** (`maxSelectable`) as the only numeric authority while allowing **relationship semantics** (per-ticket/per-order) to be expressed server-side and reflected on each refresh.
- Avoids client-side math races (parent changes while stock updates) and **prevents leakage** when parents are gated/omitted.
- Scales from trivial (one parking pass per order) to complex (many parents, dynamic holds) without changing client code.
- **Security:** The server validates everything on submission. If a user somehow bypassed the UI and tried to add 5 parking passes when `maxSelectable=1`, the server's order validation would reject it. The client UI is a convenience, not a security boundary.

10.3 Examples (tiny, canonical)

The following examples demonstrate how `matchBehavior` works in practice. Notice how `maxSelectable` starts at 0 when no parent is selected, then updates via server refresh when parents are added.

Quick reference box — per-ticket vs per-order

```
// Per-ticket add-on: matches parent quantity (e.g., Fast Pass)
{
  "product": { "id": "add_fastpass", "name": "Fast Pass", "type": "digital" },
  "relations": { "parentProductIds": ["prod_ga", "prod_vip"], "matchBehavior": "per_ticket"
},
  "commercial": { "maxSelectable": 0 }
}

// Per-order add-on: one per order regardless of parent quantity (e.g., Parking)
{
  "product": { "id": "add_parking", "name": "Parking Pass", "type": "physical" },
  "relations": { "parentProductIds": ["prod_ga", "prod_vip"], "matchBehavior": "per_order"
},
  "commercial": { "limits": { "perOrder": 1 }, "maxSelectable": 0 }
}
```

Server recomputes `maxSelectable` on refresh based on current parent selection and stock; client never invents caps.

A) Per-ticket meal voucher (initial: no parent selected)

```
{
  "product": { "id": "addon_meal", "name": "Meal Voucher", "type": "digital" },
  "relations": {
    "parentProductIds": ["prod_ga", "prod_vip"],
    "matchBehavior": "per_ticket"
  },
  "state": {
    "temporal": { "phase": "during", "reasons": [] },
    "supply": { "status": "available", "reasons": [] },
    "gating": {
      "required": false,
      "satisfied": true,
      "listingPolicy": "omit_until_unlocked",
      "reasons": []
    },
    "demand": { "kind": "none", "reasons": [] },
    "messages": [
      {
        "code": "addon_requires_parent",
        "text": "Add at least one ticket to select this add-on",
        "placement": "row.under_quantity",
        "variant": "neutral",
        "priority": 40
      }
    ]
  },
  "commercial": {
    "price": {
      "amount": 1500,
      "currency": { "code": "USD", "base": 10, "exponent": 2 },
      "scale": 2
    },
    "feesIncluded": false,
    "maxSelectable": 0
  }
}
```



```

    },
    "display": { "badges": ["Add-on"], "sectionId": "add_ons" }
  }

```

After the buyer selects **3** GA tickets and the client refreshes the payload:

```

"commercial": {
  "price": { "amount": 1500, "currency": { "code": "USD", "base": 10, "exponent": 2 },
    "scale": 2 },
  "feesIncluded": false,
  "maxSelectable": 3
}

```

B) Per-order parking pass (cap 1), parent required

This example shows a typical per-order add-on. The server enforces `limits.perOrder=1`, and `maxSelectable` updates from 0 to 1 once any parent ticket is selected (regardless of parent quantity).

```

{
  "product": {
    "id": "addon_parking",
    "name": "Parking Pass",
    "type": "physical"
  },
  "relations": {
    "parentProductIds": ["prod_ga", "prod_vip"],
    "matchBehavior": "per_order"
  },
  "state": {
    "temporal": { "phase": "during", "reasons": [] },
    "supply": { "status": "available", "reasons": [] },
    "gating": {
      "required": false,
      "satisfied": true,
      "listingPolicy": "omit_until_unlocked",
      "reasons": []
    },
    "demand": { "kind": "none", "reasons": [] },
    "messages": []
  },
  "commercial": {
    "price": {
      "amount": 2000,
      "currency": { "code": "USD", "base": 10, "exponent": 2 },
      "scale": 2
    },
    "feesIncluded": false,
    "limits": { "perOrder": 1 },
    "maxSelectable": 0 // before any parent selected
  },
  "display": { "badges": ["Add-on"], "sectionId": "add_ons" }
}

```

After selecting any parent tickets (≥ 1), refresh yields `maxSelectable: 1`.

C) Hidden parent \Rightarrow add-on omitted (zero-leak)

```
// Parent product is gated and omitted with listingPolicy="omit_until_unlocked".
// The add-on referencing only that parent is also omitted.
// Context hints gating, but does not leak what the add-on is.
"context": {
  "gatingSummary": { "hasHiddenGatedItems": true },
  "panelNotices": [
    { "code": "requires_code", "text": "Enter access code to view all available options",
"variant": "info", "priority": 90 }
  ]
}
```

10.4 How the UI derives behavior (mechanical)

These rules use **only** server facts + current selection, and never compute money or availability.

- **Quantity UI shows** only when (§8):
 - `presentation === "normal"` and
 - `isPurchasable === true` and
 - `commercial.maxSelectable > 0`.
- **Add-on enablement loop (client duties):**
 1. Parent selection changes → **POST** selection → server recalculates clamps/pricing.
 2. Client receives refreshed payload → re-derives UI.
 3. Any add-on whose `maxSelectable` increased/decreased updates its UI accordingly.
- **"Per ticket" experience:** Buyers **experience** a one-for-one cap because the server reflects parent totals in `maxSelectable` on each refresh. The client **does not** compute a local numeric cap.
- **"Per order" experience:** The server typically returns `maxSelectable=1` once any parent is selected; the client shows a single-select affordance.

10.5 Edge cases & tests (acceptance checks)

- **Parent absent ⇒ add-on disabled** *Given* `relations.parentProductIds=["prod_ga"]`, `parentSelectedCount=0` *Expect* server sends `maxSelectable=0`; quantity UI hidden; any row message comes from payload (no client prose).
- **Parent added ⇒ add-on enabled (per_ticket)** *Given* buyer selects 3 GA; server refresh sets `maxSelectable=3` *Expect* quantity UI `stepper`, cap 3; price shown when row is purchasable (per §8).
- **Parent reduced ⇒ add-on clamped down** *Given* add-on selection=3, then parent selection drops to 1; server refresh sets `maxSelectable=1` *Expect* client clamps selection to 1 immediately upon applying refresh.
- **Per-order cap respected** *Given* `"per_order"`, `limits.perOrder=1`, `parentSelectedCount=4`; server sends `maxSelectable=1` *Expect* quantity UI single-select; attempts to exceed are prevented by `maxSelectable`.
- **Hidden parent (omit) ⇒ add-on omitted** *Given* the only parent for an add-on is omitted due to `omit_until_unlocked` *Expect* the add-on is also omitted (or `maxSelectable=0` with its own gating); no leakage of name/price.
- **Locked add-on masks price** *Given* add-on has `gating.required=true && !satisfied` and `listingPolicy="visible_locked"` *Expect* `presentation="locked"`, price masked, quantity hidden, CTA none (per §§3.3, 6, 8).
- **Multi-parent sum (per_ticket)** *Given* `parentProductIds=["prod_ga","prod_vip"]`, selection GA=2, VIP=1 → `parentSelectedCount=3` *Expect* server returns `maxSelectable=3`; client displays cap 3.
- **No client caps** *Given* any scenario above *Expect* client never computes `min(parentSelectedCount, X)` as a hard cap; it only applies the latest server `maxSelectable`.

10.6 Developer checklist (quick)

- ☐ Treat any `relations.parentProductIds[]` item as an **add-on**; do **not** change `product.type`.
- ☐ Never compute numeric caps locally; always use `commercial.maxSelectable`.
- ☐ On **any parent selection change**, trigger a server refresh; re-derive UI from the new payload.
- ☐ If a refresh lowers `maxSelectable`, clamp the current selection down to that value.
- ☐ Do not leak gated parents: if parents are omitted, the dependent add-on must not reveal them (trust server omission).
- ☐ Place add-ons in an "Add-ons" section via `display.sectionId` when provided; otherwise render where assigned.
- ☐ Use only payload copy (e.g., `state.messages[]`, `clientCopy`) to explain dependencies; no local strings.

Author note: This section deliberately preserves the spec’s single-clamp discipline (**maxSelectable is king**) while giving the server all levers to express parent-child constraints. The client’s job is to **refresh often** and **render faithfully**.

11. Pricing Footer (*server math; inclusions flags*)

Normative

- **Authoritative math**
 - **pricing** is **always present** and is **100% server-computed**. The client **MUST NOT** perform any money arithmetic (no sums, no prorations, no taxes/fees math).
 - All monetary values in **pricing** **MUST** be **Dinero.js v2 snapshots**: `{ amount: number, currency: { code: string, base: number, exponent: number }, scale: number }`.
 - **Single currency per panel**: `pricing.currency.code` **MUST** match every `items[].commercial.price.currency.code`. Mixed currency panels are **invalid**.
- **Shape (until the pricing contract fully stabilizes)**
 - The server **SHOULD** send the **line-item form** and **MAY** also send simpler fields. Unknown fields are **invalid**.

```
"pricing": {
  "currency": { "code": "USD", "base": 10, "exponent": 2 },
  "mode": "reserve", // or "final" (optional; see below)
  "lineItems": [
    { "code": "TICKETS", "label": "Tickets", "amount": { ...Dinero... } },
    { "code": "FEES", "label": "Fees", "amount": { ...Dinero... } },
    { "code": "TAX", "label": "Tax", "amount": { ...Dinero... } },
    { "code": "DISCOUNT", "label": "Discount", "amount": { ...Dinero... } }, //
    negative allowed
    { "code": "TOTAL", "label": "Total", "amount": { ...Dinero... } }
  ]
}
```

- Clients **MUST** render **lineItems** **in the order provided**. Clients **MUST NOT** reorder, insert, or compute derived rows.
- **Visual exception:** Clients **MAY** apply visual styling to distinguish TOTAL (e.g., bold, separator line above) without reordering.
- **Edge case:** If **TOTAL** appears mid-array (server bug), still render in provided order; do not auto-move to end.
- **Negative amounts are allowed** (e.g., discounts). Render signed values exactly as provided.

- **Mode (status of the math)**

- `pricing.mode?: "reserve" | "final"` (optional; provisional):
 - `"reserve"` → interim numbers (e.g., estimated taxes/fees).
 - `"final"` → checkout-locked totals.
- Styling or “estimated” disclaimers are **copy** concerns; the server provides the right `label` text when needed. The client **MUST NOT** infer “estimated”.

- **Inclusions flags (what's included where)**

- **Per-row:** `items[].commercial.feesIncluded: boolean`
 - **Affects copy only** next to the **row price** (e.g., “incl. fees” vs “+ fees”). It **does not** change math.
- **Footer breakdown:** Inclusion/exclusion is communicated by the **presence and values of `lineItems`**. Examples:
 - If fees are broken out, the server sends a non-zero `FEES` line.
 - If taxes are included in ticket prices, the server **omits** `TAX` or sends it as `0`.
- Clients **MUST** take inclusions/exclusions **only** from what the server sends; never guess based on flags or numbers elsewhere.

- **Updates & lifecycle**

- Any change in selection, unlock, discount, or rule **MUST** cause the client to request/receive a refreshed payload. The footer then **replaces** itself with the new `pricing`.
- With **no selection**, `pricing` still exists. The server **MAY** send an empty breakdown (`lineItems: []`) or a single `TOTAL` of 0. The client renders **exactly** what is present (no local totals).

- **When to request pricing refresh (implementation triggers)**

- **MUST refresh** on:
 - User changes any item quantity (debounce 300ms recommended)
 - User enters/applies discount code
 - User unlocks gated items (code validation success)
 - Server pushes updated payload (WebSocket/SSE)
- **MUST NOT** refresh on:
 - Pure UI interactions (expanding sections, hovering badges)
 - Client-side validation errors (max quantity warnings)
 - Rendering or scrolling events
- While awaiting refresh, keep displaying **previous pricing state**; do not show loading spinners in the footer itself (use top-level loading indicators if needed).

- **Accessibility & formatting**

- Clients **MUST** format Dinero snapshots for display (locale, currency symbol) but **MUST NOT** alter numeric values.
- The `label` field is the source of truth for line names (“Fees”, “Estimated tax”, etc.). Clients **MUST NOT** substitute their own strings.

- **Dinero formatting (implementation guidance)**

- The client **MAY** format Dinero snapshots using one of two approaches:

Approach A: Using Dinero.js utils

```
import { dinero, toDecimal } from "dinero.js";
```

```
const price = dinero(snapshot); // reconstruct from snapshot
const formatted = toDecimal(price); // "150.00"
```

Approach B: Direct from snapshot (no Dinero import required)

```
function formatDineroSnapshot(snapshot: DineroSnapshot): string {
  const { amount, currency, scale } = snapshot;
  const divisor = Math.pow(currency.base, scale);
  const value = amount / divisor;
  return new Intl.NumberFormat("en-US", {
    style: "currency",
    currency: currency.code,
  }).format(value);
}
```

- Both approaches are valid; choose based on bundle size vs. simplicity tradeoffs.
- The client **MUST NOT** perform arithmetic on snapshot values beyond display formatting.
- **Interplay with rows (§6 & §8)**
 - **Row price visibility** is governed by §6/§8 (shown only when purchasable; masked when locked). The footer is **independent** of row price visibility.
 - `commercial.maxSelectable` and all selection logic **MUST NOT** be inferred from `pricing`.
- **Line-item code registry (for semantics, not strings)**
 - Recommended codes the client recognizes **without** hardcoding display text:
 - `"TICKETS", "FEES", "TAX", "DISCOUNT", "TOTAL"`
 - Unknown codes **MUST** be rendered as provided (label + amount); clients ignore semantics.

Rationale

- Keeping **all** money math on the server eliminates client/server drift, race conditions, and rounding bugs.
 - Treating inclusions as **what the server chooses to show** (breakout vs. folded into "Tickets") avoids duplicate logic and disagreements.
 - Using Dinero snapshots across the wire ensures consistent precision and currency handling with zero floating-point surprises.
-

State transitions (loading & errors)

- **During pricing refresh:**
 - Keep displaying the **last valid pricing** state.
 - Apply subtle "updating" indicator (e.g., 50% opacity overlay or small spinner icon) if refresh takes >500ms.
 - **MUST NOT** clear the footer to empty state during refresh (prevents jarring layout shifts).
- **On pricing fetch error:**
 - Keep displaying last valid pricing with error indicator (e.g., warning icon + "Prices may be outdated").
 - Retry automatically after 3s; show "Retry" button if 3 attempts fail.
 - **MUST NOT** block checkout if pricing is stale <30s (server will revalidate).
- **On currency mismatch (validation error):**
 - Display error state immediately: "Configuration error. Please contact support."
 - Log error with payload excerpt for debugging; **MUST NOT** attempt to render broken pricing.

Visual rendering patterns (implementation guidance)

• Empty state (no selection):

[No pricing footer displayed]

Rationale: Showing "\$0.00" before selection implies something is in cart; showing nothing is clearer.

• Standard breakdown:

Tickets	\$150.00
Fees	\$ 18.00
Tax	\$ 12.00
<hr/>	
Total	\$180.00

Use visual hierarchy: regular weight for line items, bold for TOTAL.

• With discount (negative amount):

Tickets	\$150.00
Fees	\$ 18.00
Promo applied	-\$ 10.00
<hr/>	
Total	\$158.00

Render negative amounts with minus sign; use distinct color (not red, as red implies error—use theme accent or green for savings).

• All-inclusive (single line):

Total	\$ 50.00
-------	----------

When server sends only TOTAL, render just that line (no artificial breakdown).

Examples (compact, canonical)

A) Minimal, no selection yet (empty breakdown)

```
"pricing": {
  "currency": { "code": "USD", "base": 10, "exponent": 2 },
  "mode": "reserve",
  "lineItems": []
}
```

Render nothing in the footer (no sums). Do **not** invent a \$0 total.

B) Tickets + fees + tax + discount + total (reserve)

```

"pricing": {
  "currency": { "code": "USD", "base": 10, "exponent": 2 },
  "mode": "reserve",
  "lineItems": [
    {
      "code": "TICKETS",
      "label": "Tickets",
      "amount": { "amount": 15000, "currency": { "code": "USD", "base": 10, "exponent": 2 },
"scale": 2 }
    },
    {
      "code": "FEES",
      "label": "Fees",
      "amount": { "amount": 1800, "currency": { "code": "USD", "base": 10, "exponent": 2 },
"scale": 2 }
    },
    {
      "code": "TAX",
      "label": "Estimated tax",
      "amount": { "amount": 1200, "currency": { "code": "USD", "base": 10, "exponent": 2 },
"scale": 2 }
    },
    {
      "code": "DISCOUNT",
      "label": "Promo applied",
      "amount": { "amount": -500, "currency": { "code": "USD", "base": 10, "exponent": 2 },
"scale": 2 }
    },
    {
      "code": "TOTAL",
      "label": "Total",
      "amount": { "amount": 17500, "currency": { "code": "USD", "base": 10, "exponent": 2 },
"scale": 2 }
    }
  ]
}

```

Render labels as provided; display signed values; no recomputation.

C) All-in ticket pricing (fees folded), tax omitted (final)

```

"pricing": {
  "currency": { "code": "USD", "base": 10, "exponent": 2 },
  "mode": "final",
  "lineItems": [
    {
      "code": "TICKETS",
      "label": "Tickets (incl. fees)",
      "amount": { "amount": 5000, "currency": { "code": "USD", "base": 10, "exponent": 2 },
"scale": 2 }
    },
    {
      "code": "TOTAL",
      "label": "Total",
      "amount": { "amount": 5000, "currency": { "code": "USD", "base": 10, "exponent": 2 },
"scale": 2 }
    }
  ]
}

```

*Fees are included in tickets; server chose not to show a separate **FEES** row.*

Tests (acceptance checks)

- **Presence & shape**

- Given a payload without **pricing** → Expect client validation error (section 4 requires it).
- Given **pricing.lineItems** present → Expect render **exactly** those rows, in provided order.

- **Currency consistency**

- Given any **items[].commercial.price.currency.code** ≠ **pricing.currency.code** → Expect client validation error or hard block (single currency per panel).

- **Currency mismatch (error state)**

- Given **items[0].commercial.price.currency.code="USD"** and **items[1].commercial.price.currency.code="EUR"**
- Expect client **MUST** reject payload during validation; display error state: "Configuration error: Mixed currencies detected"
- *Implementation:* Validate currency consistency immediately upon payload receipt, before any rendering.
- *Rationale:* Mixed currency is a server configuration bug, not a runtime state; fail fast.

- **No local math**

- Given **lineItems** omit **TOTAL** → Expect no computed total; render only present rows.
- Given negative **DISCOUNT.amount** → Expect show signed value; do not alter other rows.

- **Mode handling**

- Given **pricing.mode="reserve"** with **TAX** labeled "Estimated tax" → Expect render that label verbatim; do not infer or add "estimated" elsewhere.
- Given **pricing.mode="final"** → Expect render labels verbatim; client does not change copy.

- **Inclusions**

- Given rows with **commercial.feesIncluded=false** and a non-zero **FEES** line → Expect per-row copy uses the server's provided strings (e.g., "+ fees" via templates) and the footer shows the **FEES** row; no client inference.
- Given a panel where taxes are included in **TICKETS** (no **TAX** row) → Expect no tax row rendered; do not add one.

- **Zero/empty state**

- Given **lineItems: []** (no selection) → Expect no totals rendered; do not synthesize **\$0.00**.
- Given a single **TOTAL** of 0 → Expect render "Total \$0.00" with provided label; no extra rows.

- **Robustness**

- Given an unknown line item { **code:"ADJUSTMENT", label:"Adjustment", amount: {...}** } → Expect render it as provided; do not drop or re-label.
 - Given selection change → Expect client requests/uses new **pricing**; do not animate by recomputing locally.
-

Developer checklist (implementation audit)

Money handling:

- ☐ All Dinero snapshots formatted for display only (no arithmetic on **amount** fields)
- ☐ Currency consistency validated on every payload receipt
- ☐ Negative amounts (discounts) rendered with minus sign and distinct styling
- ☐ No rounding or precision changes applied to snapshot values

Rendering:

- ☐ `lineItems` rendered in exact payload order (no reordering logic)
- ☐ `label` text used verbatim (no substitutions like "Subtotal" → "Items")
- ☐ Empty `lineItems[]` renders nothing (no synthetic "\$0.00" line)
- ☐ Visual hierarchy applied (TOTAL distinguished via styling, not reordering)

State management:

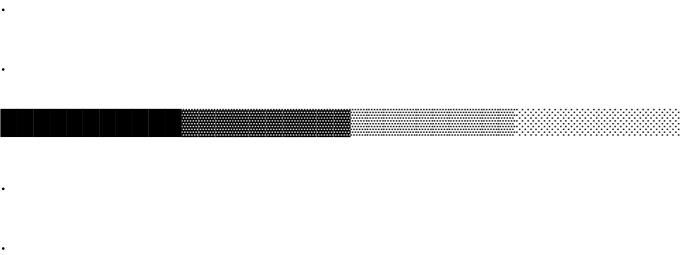
- ☐ Previous pricing displayed during refresh (no empty state flash)
- ☐ Pricing refresh triggered only on quantity/selection changes, not UI events
- ☐ Debounce on quantity changes (300ms recommended) to avoid excessive server calls

Error handling:

- ☐ Currency mismatch causes validation failure (payload rejected)
- ☐ Stale pricing (fetch error) handled gracefully with retry logic
- ☐ Unknown line item codes are invalid (strict enum); update server before introducing new codes

Testing coverage:

- ☐ Fixture: Empty lineItems (no selection)
- ☐ Fixture: Standard breakdown (tickets + fees + tax + total)
- ☐ Fixture: With discount (negative amount)
- ☐ Fixture: All-inclusive (single TOTAL line)
- ☐ Fixture: Unknown line item code (e.g., "ADJUSTMENT")
- ☐ Fixture: Currency mismatch (validation error)



12. Reason Codes Registry (*machine codes; copy via messages/templates*)

Contract intent: Codes are **machine facts**. They never render by themselves. User-facing text is delivered via `items[].state.messages[]` (row) or `context.panelNotices[]` (panel), optionally using `context.copyTemplates[]`. See §§2, 5, 7.

Normative

- **Code style**
 - Codes **MUST** be `snake_case` ASCII tokens (e.g., `sold_out`, `requires_code`).
 - Codes **MUST NOT** contain tense, punctuation, or spaces.
 - Field names remain `camelCase`; only machine codes are `snake_case`. (§2)
- **Where codes appear**
 - **Axis evidence:** `state.temporal.reasons[]`, `state.supply.reasons[]`, `state.gating.reasons[]`, `state.demand.reasons[]`. These arrays carry machine facts; they **MUST NOT** be rendered as strings directly. (§§3, 7)
 - **Row messages:** `state.messages[].code` may **reference** a code when `text` is omitted and a `copyTemplates[key==code]` exists. (§7.1)
 - **Panel banners:** `context.panelNotices[].code` identifies the notice kind; the banner text comes from `text` or a template keyed by `code`. (§5.4)
 - **CTA labels:** if a row CTA needs text, provide a `state.messages[]` entry with `placement: "row.cta_label"` and `code` or `text`. (§8.1, §7)

- **What codes do not do**
 - Codes **MUST NOT** drive business decisions; business logic lives on the server and is expressed via axis fields and clamps. (§1, §13)
 - Clients **MUST NOT** translate codes locally. If no `text` and no matching template exist, omit the string. (§7.1)
 - Clients **MUST NOT** invent panel banners from codes present only in item axes. Panel banners come **only** from `context.panelNotices[]`. (§5)
- **Extending the registry**
 - New codes **MUST** be added to the shared schema and deployed atomically with server and client.
 - Prefer short, specific codes (one meaning each). Avoid synonyms (`event_sold_out` **not** `event_fully_booked`).
- **Scope**
 - This registry covers **reason/message/notice** codes. **Pricing line item codes** like `"TICKETS"`, `"FEES"`, `"TOTAL"` are **not** reason codes and are out of scope for this section. (§4.4)

Canonical code sets (v0.4-g)

The lists below reflect codes **already used** in this spec’s examples and recommended authoring patterns. You can ship more, but keep semantics consistent with the axis.

A) Temporal (axis: `state.temporal.reasons[]` and/or row messages)

Code	Meaning (machine fact)	Typical surface
<code>outside_window</code>	Not on sale yet; phase is <code>"before"</code>	Row message under title (e.g., on-sale info)
<code>sales_ended</code>	Sale window closed; phase is <code>"after"</code>	Row message under title
<code>sales_end_soon</code>	Sale ending soon (server-decided threshold)	Row message (warning) or panel notice (info)
<code>on_sale_at*</code>	Preformatted on-sale stamp (message-only code)	Row message under title

- `on_sale_at` is a **message code**, not an axis reason; server supplies `text` (“On sale Fri 10:00 AM CT”) directly. (§7.3)

B) Supply (axis: `state.supply.reasons[]` and/or row messages)

Code	Meaning (machine fact)	Typical surface
<code>sold_out</code>	No remaining stock; <code>supply.status="none"</code>	Row message under quantity
<code>remaining_low*</code>	Low stock urgency (server decided)	Row message under quantity; use <code>params.count</code>

- `remaining_low` is commonly used as a **message code** resolved via `copyTemplates` with `{count}`; it does **not** change `supply.status` (which remains `"available"`). Use `display.showLowRemaining=true` for styling. (§2, §5.4)

C) Gating (axis: `state.gating.reasons[]` and panel notice)

Code	Meaning (machine fact)	Typical surface
<code>requires_code</code>	Access gate present and not satisfied	Locked row message (<code>visible_locked</code>) or panel notice
<code>invalid_code*</code>	Last unlock attempt failed validation (server decision)	Panel notice (error)
<code>unlocked*</code>	Gate satisfied (for celebratory copy if desired)	Optional row message under title

- `invalid_code/unlocked` are **message/notice codes**; there’s no requirement to add them to axis `reasons[]`. Access validation remains server-side. (§3.3, §5)

D) Demand (axis: `state.demand.reasons[]` and CTA messaging)

Code	Meaning (machine fact)	Typical surface
<code>waitlist_available</code>	Waitlist is enabled for this item	Row CTA label message (e.g., "Join Waitlist")
<code>notify_available</code>	Notify-me is enabled (pre-sale phase)	Row CTA label message (e.g., "Notify Me")

Gating precedence: if `gating.required && !satisfied`, demand CTAs **MUST NOT** surface until unlocked. (§3.5, §8.7)

E) Panel-level notices (context: `panelNotices[].code`)

Code	Meaning	Notes & surface
<code>requires_code</code>	Event has gated inventory users can unlock	Info banner; pairs with AccessCode input UI
<code>event_sold_out</code>	All visible sellable inventory is gone	Info banner; may appear with/without waitlist
<code>payment_plan_available</code>	Payment plans exist at checkout	Info banner; never per-row badge (§5.3)
<code>sales_end_soon</code>	Event-wide urgency on sale end	Optional banner (server chooses)

If public items are sold out **and** `gatingSummary.hasHiddenGatedItems=true`, prefer `requires_code` over an "Event Sold Out" banner to avoid misleading users. (§3.5, §5)

F) Client-triggered validation (context: `clientCopy` keys)

Key	Purpose (client action → uses server copy)
<code>selection_min_reached</code>	Pressed checkout without required selection
<code>selection_max_types</code>	Tried to select > allowed ticket types
<code>quantity_min_reached</code>	Tried to set quantity below minimum for a type
<code>quantity_max_reached</code>	Tried to exceed <code>maxSelectable</code> /per-order guidance

These are **not axis reasons**. They are templates the client triggers with params (e.g., `{max}`, `{min}`) on invalid actions. (§5.5)

G) CTA label helpers (row message, `placement`: `"row.cta_label"`)

Code	Typical text (server-provided)	When used (derived; §8)
<code>waitlist_cta</code>	"Join Waitlist"	<code>supply.status="none"</code> + demand= <code>waitlist</code>
<code>notify_cta</code>	"Notify Me"	<code>temporal.phase="before"</code> + demand= <code>notify_me</code>

Remember: **the client never hardcodes** CTA strings; provide these via messages/templates. (§8.1)

Authoring guidance (server-side)

- **One cause, one code.** Put the cause exactly once in its axis `reasons[]`. Use `messages[]` or `panelNotices[]` to speak to the user. (§3, §7)
- **Pick canonical names.** Use these codes as written:
 - Temporal: `outside_window`, `sales_ended`, `sales_end_soon`
 - Supply: `sold_out` (+ message code `remaining_low`)
 - Gating: `requires_code`, `invalid_code` (notice), `unlocked` (message)
 - Demand: `waitlist_available`, `notify_available`
 - Panel: `event_sold_out`, `requires_code`, `payment_plan_available`, `sales_end_soon`

- **Don't overload codes.** `sold_out` means "no stock" only; it does not imply waitlist—set `demand.kind="waitlist"` separately. (§3.4)
- **Message-only codes.** It's fine to emit message codes with no matching axis reason (e.g., `on_sale_at`, `remaining_low`), provided you supply `text` or a template. (§7.4)
- **Priorities & placement.** For stacked messages, set `priority` to control order and choose a specific `placement` slot. (§7.1)
- **Localization.** Resolve `messages[].text`/templates server-side per locale; clients don't translate. (§7.1, §5.4)

Examples (compact & canonical)

1) Sold out with waitlist (row CTA label via message code):

```
"state": {
  "temporal": { "phase": "during", "reasons": [] },
  "supply": { "status": "none", "reasons": ["sold_out"] },
  "gating": { "required": false, "satisfied": true, "listingPolicy":
"omit_until_unlocked", "reasons": [] },
  "demand": { "kind": "waitlist", "reasons": ["waitlist_available"] },
  "messages": [
    { "code": "sold_out", "text": "Sold Out", "placement": "row.under_quantity",
"priority": 100 },
    { "code": "waitlist_cta", "text": "Join Waitlist", "placement": "row.cta_label",
"priority": 80 }
  ]
}
```

2) Before sale (preformatted time) + notify-me:

```
"state": {
  "temporal": { "phase": "before", "reasons": ["outside_window"] },
  "supply": { "status": "available", "reasons": [] },
  "gating": { "required": false, "satisfied": true, "listingPolicy":
"omit_until_unlocked", "reasons": [] },
  "demand": { "kind": "notify_me", "reasons": ["notify_available"] },
  "messages": [
    { "code": "on_sale_at", "text": "On sale Fri 10:00 AM CT", "placement":
"row.under_title" },
    { "code": "notify_cta", "text": "Notify Me", "placement": "row.cta_label"
  ]
}
```

3) Visible locked (gated) + panel requires-code banner:

```
// Row
"state": {
  "gating": { "required": true, "satisfied": false, "listingPolicy": "visible_locked",
"reasons": ["requires_code"] },
  "messages": [
    { "code": "requires_code", "text": "Requires access code", "placement":
"row.under_title", "priority": 80 }
  ],
  "temporal": { "phase": "during", "reasons": [] },
  "supply": { "status": "available", "reasons": [] },
  "demand": { "kind": "none", "reasons": [] }
},
```

```
// Panel
"context": {
  "gatingSummary": { "hasHiddenGatedItems": false },
  "panelNotices": [
    { "code": "requires_code", "variant": "info", "text": "Enter access code to view tickets", "priority": 90 }
  ]
}
```

4) Low remaining message via template (no axis reason required):

```
"context": {
  "copyTemplates": [ { "key": "remaining_low", "template": "Only {count} left!" } ]
},
"state": {
  "supply": { "status": "available", "reasons": [], "remaining": 3 },
  "messages": [
    { "code": "remaining_low", "params": { "count": 3 }, "placement": "row.under_quantity",
"priority": 60 }
  ]
}
```

5) Panel sold-out vs hidden gated (choose banner wisely):

```
// Case A: truly sold out with no hidden gated
"context": {
  "gatingSummary": { "hasHiddenGatedItems": false },
  "panelNotices": [ { "code": "event_sold_out", "variant": "info", "text": "Event sold out", "priority": 100 } ]
}
// Case B: public sold out but hidden gated exists – do NOT show event_sold_out; invite code instead
"context": {
  "gatingSummary": { "hasHiddenGatedItems": true },
  "panelNotices": [ { "code": "requires_code", "variant": "info", "text": "Enter access code to view tickets", "priority": 100 } ]
}
```

Tests (acceptance checks)

- **Code style**
 - Given any `reasons []` or `messages [].code` with non-`snake_case`, validation **MUST** fail.
- **No direct rendering of codes**
 - Given an item with `state.supply.reasons=["sold_out"]` and **no** matching message/template, the UI **MUST NOT** show `"sold_out"`; it shows **no** string for that reason. (§7.1)
- **Template resolution**
 - Given `messages:[{ code:"remaining_low", params:{count:2}, placement:"row.under_quantity" }]` and `copyTemplates:[{key:"remaining_low",template:"Only {count} left!"}]`, render "Only 2 left!". (§7.7)
- **Panel vs row separation**
 - Given all rows `sold_out` and `panelNotices=[]`, the client **MUST NOT** invent an "Event Sold Out" banner. (§5)

- Given `panelNotices=[{ code:"payment_plan_available", ...}]`, a banner **MUST** render; per-row payment-plan badges **MUST NOT** appear. (§5.3)

- **Gating precedence**

- Given `gating.required=true && !satisfied` (visible locked) and `demand.kind="waitlist"`, the row CTA **MUST** be `none` (no waitlist button) until unlocked; price masked. (§3.5, §8.7)

- **Hidden gated hint**

- Given omitted gated items with stock, `context.gatingSummary.hasHiddenGatedItems` **MUST** be `true`; client **MUST NOT** infer more than that boolean. (§3.3)

- **Unknown codes safe-ignore**

- Given an unrecognized `messages[].code` with `text` supplied → render `text`.
- Given unrecognized `panelNotices[].code` with `text` supplied → render the notice.
- Given unrecognized `code` with **no** `text` and **no** template → omit silently.

Quick reference (cheat sheet)

- **Temporal:** `outside_window`, `sales_ended`, `sales_end_soon`, `(message) on_sale_at`
- **Supply:** `sold_out`, `(message) remaining_low`
- **Gating:** `requires_code`, `(notice) invalid_code`, `(message) unlocked`
- **Demand:** `waitlist_available`, `notify_available`
- **Panel:** `event_sold_out`, `requires_code`, `payment_plan_available`, `sales_end_soon`
- **Client validation (templates):** `selection_min_reached`, `selection_max_types`, `quantity_min_reached`, `quantity_max_reached`
- **CTA labels (row message placement):** `waitlist_cta`, `notify_cta`

Memory hook: **Axes explain; messages speak; notices announce.** Codes tag the facts; copy carries the words.

13. Invariants & Guardrails (what the client MUST NOT do)

Why these exist: The panel's power comes from being a **pure view** of server state. These guardrails prevent common pitfalls that lead to drift, bugs, security issues, and user confusion. By keeping all business logic server-side, we gain: unified logic (no client/server contradictions), dynamic adaptability (same UI handles all scenarios), improved user clarity (server knows exact reasons), and security (client cannot bypass rules).

Hard rules (non-negotiable)

- **No schedule math.** Do **not** compute countdowns or sale windows. Use `temporal.phase` and any server-supplied text.
 - **Why:** Timezones, DST transitions, and schedule pauses make client-side time math brittle. Sales can be paused mid-window for operational reasons. The server is authoritative.
- **No availability math.** Do **not** infer "sold out" or "low stock" from numbers. Use `supply.status` and server messages.
 - **Why:** Counts can be stale; seat maps impose adjacency constraints; holds exist. A single `status` decision from the server avoids race conditions and incorrect displays.
- **No price math.** Do **not** compute totals, fees, taxes, discounts, or installment effects. Render `pricing` verbatim.
 - **Why:** Tax rules, promotional discounts, and installment schedules have complex business logic. Client-computed prices create a "gray area" where displayed totals contradict the actual charge.

- **Money architecture:** All monetary values are **Dinero.js V2 snapshots**. Nothing happens to money outside Dinero utils. The client receives Dinero snapshots, **MAY** format for display using Dinero utils or directly from snapshot values, but **MUST NOT** perform arithmetic. All calculations happen server-side.
- **No clamp math.** Do **not** derive selection limits from counts or limits. Enforce **only** `commercial.maxSelectable`.
 - **Why:** Multiple constraints (per-order, per-user, remaining, fraud rules) combine server-side. A single clamp prevents contradictory UI behavior and respects all policies.
- **No gate logic.** Do **not** validate access codes, apply rate limits, or reveal omitted SKUs. Respect `gating.satisfied` and `listingPolicy`.
- **No hidden-SKU inference.** Do **not** guess names/counts/prices for omitted items. The **only** hint is `gatingSummary.hasHiddenGatedItems`.
- **No ad-hoc banners or row text.** Render **only** `context.panelNotices[]` and `state.messages[]` (or `copyTemplates/clientCopy` when specified).
- **No approval/request flows.** There is **no** approval concept in this contract; do not invent a "Request" CTA.
- **No per-row payment-plan badges.** Payment plans surface via a **panel** notice, not per item.
- **No legacy fields.** The schema is authoritative; if a field isn't defined here, it doesn't exist.

What the client MAY derive (presentation only)

The client **derives UI state** from server facts using pure functions (atoms). These derivations are **re-run** when new payloads arrive, using the new facts. The client never predicts or back-calculates server state.

Allowed derivations:

- **Row presentation:** `normal` | `locked` from server facts (locked = present + `gating.required` && `!satisfied`).
 - Items with `listingPolicy="omit_until_unlocked"` are not sent; they have no presentation state.
- **Purchasable boolean:** `temporal.phase="during"` AND `supply.status="available"` AND (gate satisfied or not required).
- **CTA selection:** `purchase` / `join_waitlist` / `notify_me` / `none` from **server fields only** (`demand.kind`, `supply.status`, `temporal.phase`, `gating`).
- **Selection validity:** Enable/disable bottom CTA against `context.orderRules` (min types/tickets), without computing business policy.

Key principle: These are **pure transformations** of server data into UI state. Atoms compose server facts; they do not compute business decisions.

Security guardrails

Threat model: Because all business logic lives server-side, the system is robust against client-side tampering. Users cannot unlock hidden tickets, bypass purchase limits, or discover secret SKUs except by calling the server, which validates everything. The client is a "dumb terminal" that respects server state—**security by architecture**, not by obfuscation.

- **Access codes & tokens**
 - Do **not** log access codes or gating tokens in analytics, errors, or URLs.
 - **Why:** Prevents code leakage in logs, error reports, or shared links.
 - Do **not** persist gating tokens beyond server TTL or across accounts.
 - **Why:** Tokens are single-session credentials; reuse across accounts or after expiry bypasses server validation.
 - All unlock attempts go to the server; client performs no retries beyond normal UX.
 - **Why:** Server enforces rate limiting and tracks brute-force attempts; client retries would bypass these protections.
- **Data exposure**
 - Do **not** display machine codes directly; show payload-supplied text only.
 - **Why:** Machine codes (`sold_out`, `requires_code`) are internal; exposing them in UI creates support burden and leaks implementation details.
 - Do **not** render price for locked rows (`visible_locked`); price is masked.
- **Caching & staleness**

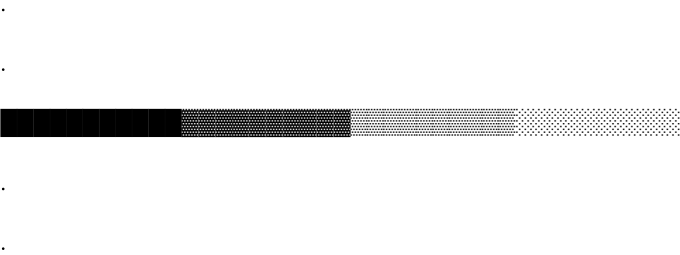
- Do **not** predict or back-calculate business state; always re-derive presentation from new server facts.
 - **Why:** The client derives UI state (presentation, CTA, purchasability) from server facts via atoms. When a new payload arrives, atoms re-run their derivation functions using the **new facts**. The client never attempts to predict what the next server state will be (e.g., flipping `temporal.phase` based on local clock) or back-calculate business decisions (e.g., recomputing `maxSelectable` from `remaining`).
 - **Pattern:** New payload → atoms re-derive → React re-renders. No local state contradicts server state.
- Do **not** cache omitted items or speculate their presence.
 - **Why:** Omitted items (`listingPolicy="omit_until_unlocked"`) are a zero-leak security feature. Caching defeated items reveals their existence.

Allowed vs. forbidden (quick matrix)

Action	Allowed?	Source of truth
Show "Sold Out" on a row	✓	<code>supply.status="none"</code> + messages
Show "Enter access code" banner	✓	<code>context.panelNotices[]</code>
Enable "Join Waitlist" CTA	✓	<code>demand.kind="waitlist"</code>
Mask price on a locked row	✓	<code>gating.required && !satisfied</code>
Compute countdown to sale start	✗	Use server-provided messaging
Derive <code>maxSelectable</code> from <code>remaining/limits</code>	✗	Use <code>commercial.maxSelectable</code>
Add a custom "Event Sold Out" banner	✗	Only <code>panelNotices[]</code>
Display per-row "Payment Plan" badge	✗	Use panel notice

Next steps (tight, build-ready)

- Lock enums and shapes in the schemas: `supply`, `gating.listingPolicy`, `demand`, `state.messages[]`, `panelNotices[]`, `gatingSummary`, `orderRules`.
- Add Storybook fixtures to exercise guardrails: available, sold out + waitlist, `visible_locked`, `omit_until_unlocked`, public-sold-out + hidden-gated.
- Draft **Appendix: Terminology & Migration Notes** with the full forbidden→preferred mapping (for authors only), keeping Section 2 purely normative.



14. Zod/TS Schemas (*single source of truth*)

Purpose: Define runtime-validated, type-safe schemas for the entire Product Panel contract using Zod 4. These schemas serve as the **single source of truth** for TypeScript types, runtime validation, form handling (TanStack Form), API responses (TanStack Query), and atom state (Jotai).

Architecture Context: TanStack Start & Validation Strategy

Deployment Model: DayOf uses **TanStack Start with server functions**, which fundamentally shapes our validation approach.

Key Architectural Facts:

1. **Atomic Deployments:** Server and client code deploy together as a single Vercel project. There is **no version skew**—server and client are always the same version.

- 2. **Isomorphic Validation:** The same Zod schemas validate data on both:
 - **Server-side:** During SSR, loader execution, and server function calls
 - **Client-side:** During hydration, subsequent fetches, and form submissions
- 3. **SSR/Hydration Consistency:** The critical risk is not forward compatibility (no old clients exist), but **server-rendered HTML mismatching client hydration**. Strict validation catches these bugs immediately.
- 4. **Greenfield Architecture:** No legacy clients, no long-lived mobile apps with independent release schedules, no cached data surviving deployments.

Why This Matters for Schemas:

Traditional API design assumes **version skew**: old clients hitting new servers, requiring forward-compatible schemas that ignore unknown fields. With TanStack Start server functions:

- **No old clients exist** → Unknown fields are bugs, not future features
- **SSR + hydration use same schemas** → Strict validation ensures consistency
- **Atomic deploys** → Adding enum values just requires one coordinated deploy
- **Server functions** → No separate API versioning; server + client update together

Validation Philosophy:

Traditional REST API	TanStack Start (Our Approach)
<code>.passthrough()</code> (ignore unknown fields)	<code>.strict()</code> (reject unknown fields)
Optional with <code>.default()</code> everywhere	Required fields fail fast
Extensible enums (<code>z.string()</code>)	Strict enums (<code>z.enum()</code>)
Forward compatibility priority	Bug detection priority
Goal: Old clients don't break	Goal: Server/client stay in sync

Concrete Example:

```
// ❌ Traditional REST API approach (not needed for us)
const OrderRulesSchema = z
  .object({
    types: z.enum(["single", "multiple"]).default("multiple"), // Hide server bugs
  })
  .passthrough(); // Ignore unknown fields

// ✅ TanStack Start approach (what we do)
const OrderRulesSchema = z
  .object({
    types: z.enum(["single", "multiple"]), // Required; fail if missing
  })
  .strict(); // Reject unknown fields immediately
```

Benefits of Strict Validation:

- 1. **Catches Server Bugs:** Missing required fields fail validation, not silently at render time
- 2. **SSR/Hydration Safety:** Schema mismatches surface immediately, preventing hydration errors
- 3. **Type Safety:** No optional-everywhere types that hide bugs
- 4. **Clear Contracts:** Required fields document server obligations explicitly
- 5. **Fail Fast:** Validation errors at the boundary, not deep in component trees

When We Would Need Forward Compatibility:

- Long-lived mobile apps (we're web-only for panel)
- Cached API responses surviving deployments (we don't cache across deploys)

- Multiple frontends with independent release schedules (our frontends deploy atomically)

We have none of these, so strict validation is the correct choice.

Schema Decisions Explained:

- **.strict()** everywhere: Unknown fields = validation errors = bugs caught early
- **No .default() on business fields**: Server must send explicit values; hiding missing data is worse than failing
- **Required fields**: `orderRules.types` is required, not `optional().default("multiple")`
- **Strict enums**: `z.enum(["single", "multiple"])` not `z.string()` (coordinated deploys are fine)
- **Remove copy refinements**: Template existence checked at render time, not schema time (separation of concerns)

14.1 Normative — Schema Architecture

A. Single source principle

- All TypeScript types for the Product Panel contract **MUST** be derived from Zod schemas via `z.infer<typeof Schema>`.
- Manual TypeScript types that duplicate schema structure are **forbidden**.
- Schemas **MUST** validate server payloads at the boundary (TanStack Query response).
- Derived client state (atoms) **MUST** use types inferred from these schemas.

B. Validation boundaries

- **Server → Client (required)**: TanStack Query **MUST** validate all API responses with the root `PanelDataSchema`.
- **Client → Server (required)**: Form submissions (access codes, quantity changes) **MUST** validate with appropriate schemas.
- **Internal derivations (type-only)**: Atom transformations use inferred types but do not re-validate (server data is already validated).

C. Schema composition rules

- Use Zod's `.strict()` on the root `PanelDataSchema` to reject unknown top-level keys (SSR/hydration consistency; atomic deploys).
- Prefer `.strict()` for nested object schemas to catch shape drift.
- Use `.default()` for **optional fields with server-defined defaults** only; client **MUST NOT** invent defaults for business fields.
- Use `.readonly()` for **immutable reference data** (e.g., `context.copyTemplates`).
- Chain `.brand()` for **nominal types** where primitive aliasing would be unsafe (e.g., `ProductId`, `DineroSnapshot`).

D. Zod 4 leveraged features

- **Codecs**: Use for bi-directional validation (parse incoming, serialize outgoing).
- **Discriminated unions**: Axis-specific schemas (e.g., `supply.status` discriminates supply state).
- **Transforms**: Currency formatting, Dinero reconstruction (display-only; never for business logic).
- **Refinements**: Cross-field validation (e.g., `maxSelectable` consistency with `supply.status`).
- **Error maps**: Custom error messages for validation failures (user-friendly, not technical).

14.2 Core Contract Schemas

These schemas mirror the contract structure from §§3–11. Order matches payload shape.

```
import { z } from "zod";

// =====
// Primitives & Branded Types
// =====

/** Machine code (snake_case reason/message/notice identifiers) */
const MachineCodeSchema = z
```

```

    .string()
    .regex(/^([a-z][a-z0-9_])*/$, "Machine codes must be snake_case")
    .brand("MachineCode");

/** Product ID (unique per panel payload) */
const ProductIdSchema = z.string().min(1).brand("ProductId");

/** Section ID (references sections[]) */
const SectionIdSchema = z.string().min(1).brand("SectionId");

// =====
// Dinero Snapshots ($4.4, $6, $11)
// =====

/** Dinero.js V2 currency object */
const DineroCurrencySchema = z.object({
  code: z.string().length(3), // ISO 4217 (e.g., "USD", "EUR")
  base: z.number().int().positive(),
  exponent: z.number().int().nonnegative(),
});

/** Dinero.js V2 snapshot (transport format for all money) */
const DineroSnapshotSchema = z
  .object({
    amount: z.number().int(), // minor units (cents)
    currency: DineroCurrencySchema,
    scale: z.number().int().nonnegative(),
  })
  .strict()
  .brand("DineroSnapshot");

type DineroSnapshot = z.infer<typeof DineroSnapshotSchema>;

// =====
// Context Schemas ($4.1, $5)
// =====

const OrderRulesSchema = z
  .object({
    types: z.enum(["single", "multiple"]),
    typesPerOrder: z.enum(["single", "multiple"]),
    ticketsPerType: z.enum(["single", "multiple"]),
    minSelectedTypes: z.number().int().nonnegative(),
    minTicketsPerSelectedType: z.number().int().nonnegative(),
  })
  .strict();

const GatingSummarySchema = z
  .object({
    hasHiddenGatedItems: z.boolean(), // Required; server decides explicitly
    hasAccessCode: z.boolean().optional(),
  })
  .strict();

const NoticeActionSchema = z
  .object({
    label: z.string().min(1),
    kind: z.enum(["link", "drawer"]),
    target: z.string().optional(),
  })
  .strict();

const NoticeSchema = z
  .object({
    code: MachineCodeSchema,

```

```

    scope: z.enum(["panel", "item"]).default("panel"),
    variant: z.enum(["neutral", "info", "warning", "error"]).default("info"),
    title: z.string().optional(),
    text: z.string().optional(),
    params: z.record(z.unknown()).optional(),
    action: NoticeActionSchema.optional(),
    priority: z.number().default(0),
    expiresAt: z.string().datetime().optional(),
  })
  .strict();
// Note: No .refine() check for "text or params required"
// Rationale: Template existence cannot be validated at schema time (coupling).
// Enforcement: Render layer omits notices/messages with neither text nor matching template.

const CopyTemplateSchema = z
  .object({
    key: MachineCodeSchema,
    template: z.string().min(1),
    locale: z.string().optional(),
  })
  .strict()
  .readonly();

const ClientCopySchema = z
  .object({
    selection_min_reached: z.string().optional(),
    selection_max_types: z.string().optional(),
    quantity_min_reached: z.string().optional(),
    quantity_max_reached: z.string().optional(),
    addon_requires_parent: z.string().optional(),
    panel_cta_continue: z.string().optional(),
    panel_cta_waitlist: z.string().optional(),
    panel_cta_disabled: z.string().optional(),
  })
  .strict();

const TooltipSchema = z
  .object({
    id: z.string().min(1),
    text: z.string().min(1),
  })
  .strict()
  .readonly();

const HoverCardSchema = z
  .object({
    id: z.string().min(1),
    title: z.string().optional(),
    body: z.string().min(1),
    action: NoticeActionSchema.optional(),
  })
  .strict()
  .readonly();

const EffectivePrefsSchema = z
  .object({
    showTypeListWhenSoldOut: z.boolean(),
    displayPaymentPlanAvailable: z.boolean(),
    displayRemainingThreshold: z.number().int().positive().optional(),
  })
  .strict();

const ContextSchema = z
  .object({
    orderRules: OrderRulesSchema, // REQUIRED per §4
  })

```

```

    gatingSummary: GatingSummarySchema.optional(),
    panelNotices: z.array(NoticeSchema).default([]),
    effectivePrefs: EffectivePrefsSchema,
    copyTemplates: z.array(CopyTemplateSchema).optional(),
    clientCopy: ClientCopySchema.optional(),
    tooltips: z.array(TooltipSchema).optional(),
    hovercards: z.array(HoverCardSchema).optional(),
  })
  .strict();

// =====
// Sections (§4.2)
// =====

const SectionSchema = z
  .object({
    id: SectionIdSchema,
    label: z.string().min(1),
    order: z.number().int().positive(),
    labelOverride: z.string().nullable().optional(),
  })
  .strict();

// =====
// State Axes (§3)
// =====

const TemporalSchema = z
  .object({
    phase: z.enum(["before", "during", "after"]),
    reasons: z.array(MachineCodeSchema),
    currentWindow: z
      .object({
        startsAt: z.string().datetime(),
        endsAt: z.string().datetime(),
      })
      .optional(),
    nextWindow: z
      .object({
        startsAt: z.string().datetime(),
        endsAt: z.string().datetime(),
      })
      .optional(),
  })
  .strict();

const SupplySchema = z
  .object({
    status: z.enum(["available", "none", "unknown"]),
    remaining: z.number().int().nonnegative().optional(),
    reasons: z.array(MachineCodeSchema),
  })
  .strict();

const GatingRequirementSchema = z
  .object({
    kind: z.string(), // e.g., "unlock_code", "membership"
    satisfied: z.boolean(),
    validWindow: z
      .object({
        startsAt: z.string().datetime(),
        endsAt: z.string().datetime(),
      })
      .optional(),
    limit: z

```

```

        .object({
          maxUses: z.number().int().positive().optional(),
          usesRemaining: z.number().int().nonnegative().optional(),
        })
        .optional(),
      })
      .strict();

const GatingSchema = z
  .object({
    required: z.boolean(),
    satisfied: z.boolean(),
    listingPolicy: z.enum(["omit_until_unlocked", "visible_locked"]),
    reasons: z.array(MachineCodeSchema),
    requirements: z.array(GatingRequirementSchema).optional(),
  })
  .strict();

const DemandSchema = z
  .object({
    kind: z.enum(["none", "waitlist", "notify_me"]),
    reasons: z.array(MachineCodeSchema),
  })
  .strict();

const MessageSchema = z
  .object({
    code: MachineCodeSchema,
    text: z.string().optional(),
    params: z.record(z.unknown()).optional(),
    placement: z.enum([
      "row.under_title",
      "row.under_price",
      "row.under_quantity",
      "row.footer",
      "row.cta_label",
    ]),
    variant: z.enum(["neutral", "info", "warning", "error"]).default("info"),
    priority: z.number().default(0),
  })
  .strict();
// Note: No .refine() check for "text or params required"
// Rationale: Template existence cannot be validated at schema time (coupling).
// Enforcement: Render layer omits messages with neither text nor matching template (§5.4, §7.1).

const StateSchema = z
  .object({
    temporal: TemporalSchema,
    supply: SupplySchema,
    gating: GatingSchema,
    demand: DemandSchema,
    messages: z.array(MessageSchema), // Required; server must send (even if empty array)
  })
  .strict();

// =====
// Product & Item Structure (§6)
// =====

const FulfillmentSchema = z
  .object({
    methods: z.array(
      z.enum([
        "eticket",

```

```

        "apple_pass",
        "will_call",
        "physical_mail",
        "shipping",
        "nfc",
    ])
  ),
  details: z.record(z.unknown()).optional(),
})
.strict();

const ProductSchema = z
  .object({
    id: ProductIdSchema,
    name: z.string().min(1),
    type: z.enum(["ticket", "digital", "physical"]).default("ticket"),
    fulfillment: FulfillmentSchema.optional(),
    description: z.string().optional(),
    subtitle: z.string().optional(),
    category: z.string().optional(),
  })
  .strict();

const VariantSchema = z
  .object({
    id: z.string().optional(),
    name: z.string().optional(),
    attributes: z.record(z.unknown()).optional(),
  })
  .strict();

const CommercialSchema = z
  .object({
    price: DineroSnapshotSchema,
    feesIncluded: z.boolean().default(false),
    maxSelectable: z.number().int().nonnegative(),
    limits: z
      .object({
        perOrder: z.number().int().positive().optional(),
        perUser: z.number().int().positive().optional(),
      })
      .strict()
      .optional(),
  })
  .strict()
  .refine(
    (commercial) => {
      // maxSelectable should be 0 when item is not selectable
      // This is a soft validation; server controls the authoritative value
      return true;
    },
    { message: "maxSelectable must reflect current selectability" }
  );

const RelationsSchema = z
  .object({
    parentProductIds: z.array(ProductIdSchema).optional(),
    matchBehavior: z.enum(["per_ticket", "per_order"]).optional(),
  })
  .strict();

const BadgeDetailRefSchema = z
  .object({
    kind: z.enum(["tooltip", "hovercard"]),
    ref: z.string().min(1),
  })

```

```

    })
    .strict();

const DisplaySchema = z
  .object({
    badges: z.array(z.string()),
    badgeDetails: z.record(BadgeDetailRefSchema).optional(),
    sectionId: SectionIdSchema.optional(),
    showLowRemaining: z.boolean(),
  })
  .strict();

const PanelItemSchema = z
  .object({
    product: ProductSchema,
    variant: VariantSchema.optional(),
    state: StateSchema,
    commercial: CommercialSchema,
    relations: RelationsSchema.optional(),
    display: DisplaySchema,
  })
  .strict()
  .refine(
    (item) => {
      // Gating invariant: omit_until_unlocked items should not be sent when unsatisfied
      // This validation catches server bugs
      if (
        item.state.gating.required &&
        !item.state.gating.satisfied &&
        item.state.gating.listingPolicy === "omit_until_unlocked"
      ) {
        return false; // Should not be in payload
      }
      return true;
    },
    {
      message:
        "Items with unsatisfied omit_until_unlocked gating must not be sent",
    }
  );

// =====
// Pricing (§11)
// =====

const PricingLineItemSchema = z
  .object({
    code: z.enum(["TICKETS", "FEES", "TAX", "DISCOUNT", "TOTAL"]),
    label: z.string().min(1),
    amount: DineroSnapshotSchema,
  })
  .strict();

const PricingSchema = z
  .object({
    currency: DineroCurrencySchema,
    mode: z.enum(["reserve", "final"]).optional(),
    lineItems: z.array(PricingLineItemSchema).default([]),
  })
  .strict()
  .refine(
    (pricing) => {
      // All line items must use the same currency
      return pricing.lineItems.every(
        (item) => item.amount.currency.code === pricing.currency.code
      );
    }
  );

```



```

    );
  },
  { message: "All line items must use the same currency as pricing.currency" }
);

// =====
// Root Contract ($4)
// =====

const PanelDataSchema = z
  .object({
    context: ContextSchema,
    sections: z.array(SectionSchema).min(1),
    items: z.array(PanelItemSchema).default([]),
    pricing: PricingSchema,
  })
  .strict() // reject unknown top-level keys for SSR/hydration consistency
  .refine(
    (data) => {
      // Currency consistency: all items must match pricing currency
      const pricingCurrency = data.pricing.currency.code;
      return data.items.every(
        (item) => item.commercial.price.currency.code === pricingCurrency
      );
    },
    { message: "All items must use the same currency as pricing" }
  )
  .refine(
    (data) => {
      // Product IDs must be unique
      const ids = data.items.map((item) => item.product.id);
      return new Set(ids).size === ids.length;
    },
    { message: "Product IDs must be unique within the panel" }
  );

// =====
// Type Exports (inferred from schemas)
// =====

export type MachineCode = z.infer<typeof MachineCodeSchema>;
export type ProductId = z.infer<typeof ProductIdSchema>;
export type SectionId = z.infer<typeof SectionIdSchema>;
export type DineroCurrency = z.infer<typeof DineroCurrencySchema>;
export type DineroSnapshot = z.infer<typeof DineroSnapshotSchema>;

export type OrderRules = z.infer<typeof OrderRulesSchema>;
export type GatingSummary = z.infer<typeof GatingSummarySchema>;
export type Notice = z.infer<typeof NoticeSchema>;
export type NoticeAction = z.infer<typeof NoticeActionSchema>;
export type CopyTemplate = z.infer<typeof CopyTemplateSchema>;
export type ClientCopy = z.infer<typeof ClientCopySchema>;
export type Tooltip = z.infer<typeof TooltipSchema>;
export type HoverCard = z.infer<typeof HoverCardSchema>;
export type EffectivePrefs = z.infer<typeof EffectivePrefsSchema>;
export type Context = z.infer<typeof ContextSchema>;

export type Section = z.infer<typeof SectionSchema>;

export type Temporal = z.infer<typeof TemporalSchema>;
export type Supply = z.infer<typeof SupplySchema>;
export type Gating = z.infer<typeof GatingSchema>;
export type GatingRequirement = z.infer<typeof GatingRequirementSchema>;
export type Demand = z.infer<typeof DemandSchema>;
export type Message = z.infer<typeof MessageSchema>;

```

```

export type State = z.infer<typeof StateSchema>;

export type Fulfillment = z.infer<typeof FulfillmentSchema>;
export type Product = z.infer<typeof ProductSchema>;
export type Variant = z.infer<typeof VariantSchema>;
export type Commercial = z.infer<typeof CommercialSchema>;
export type Relations = z.infer<typeof RelationsSchema>;
export type BadgeDetailRef = z.infer<typeof BadgeDetailRefSchema>;
export type Display = z.infer<typeof DisplaySchema>;
export type PanelItem = z.infer<typeof PanelItemSchema>;

export type PricingLineItem = z.infer<typeof PricingLineItemSchema>;
export type Pricing = z.infer<typeof PricingSchema>;

export type PanelData = z.infer<typeof PanelDataSchema>;

// Schema exports for runtime validation
export {
  MachineCodeSchema,
  ProductIdSchema,
  SectionIdSchema,
  DineroCurrencySchema,
  DineroSnapshotSchema,
  OrderRulesSchema,
  GatingSummarySchema,
  NoticeSchema,
  NoticeActionSchema,
  CopyTemplateSchema,
  ClientCopySchema,
  TooltipSchema,
  HoverCardSchema,
  EffectivePrefsSchema,
  ContextSchema,
  SectionSchema,
  TemporalSchema,
  SupplySchema,
  GatingSchema,
  GatingRequirementSchema,
  DemandSchema,
  MessageSchema,
  StateSchema,
  FulfillmentSchema,
  ProductSchema,
  VariantSchema,
  CommercialSchema,
  RelationsSchema,
  BadgeDetailRefSchema,
  DisplaySchema,
  PanelItemSchema,
  PricingLineItemSchema,
  PricingSchema,
  PanelDataSchema,
};

```

14.3 TanStack Query Integration

Purpose: Validate all server responses at the API boundary and provide type-safe query hooks.

```

import { useQuery, useMutation } from "@tanstack/react-query";
import { PanelDataSchema, type PanelData } from "../schemas";

// =====

```

```

// Query Keys (type-safe, hierarchical)
// =====

export const panelKeys = {
  all: ["panel"] as const,
  event: (eventId: string) => [...panelKeys.all, "event", eventId] as const,
  withSelection: (eventId: string, selection: Record<string, number>) =>
    [...panelKeys.event(eventId), "selection", selection] as const,
};

// =====
// API Client (validates with Zod)
// =====

async function fetchPanelData(eventId: string): Promise<PanelData> {
  const response = await fetch(`/api/events/${eventId}/panel`);

  if (!response.ok) {
    throw new Error(`Failed to fetch panel: ${response.statusText}`);
  }

  const rawData = await response.json();

  // Runtime validation with Zod
  const parseResult = PanelDataSchema.safeParse(rawData);

  if (!parseResult.success) {
    console.error("Panel validation failed:", parseResult.error.format());
    throw new Error("Invalid panel data received from server");
  }

  return parseResult.data;
}

async function updatePanelWithSelection(
  eventId: string,
  selection: Record<ProductId, number>
): Promise<PanelData> {
  const response = await fetch(`/api/events/${eventId}/panel/selection`, {
    method: "POST",
    headers: { "Content-Type": "application/json" },
    body: JSON.stringify({ selection }),
  });

  if (!response.ok) {
    throw new Error(`Failed to update selection: ${response.statusText}`);
  }

  const rawData = await response.json();
  const parseResult = PanelDataSchema.safeParse(rawData);

  if (!parseResult.success) {
    throw new Error("Invalid panel data after selection update");
  }

  return parseResult.data;
}

// =====
// Query Hooks (type-safe, validated)
// =====

export function usePanelData(eventId: string) {
  return useQuery({
    queryKey: panelKeys.event(eventId),
  });
}

```

```

    queryFn: () => fetchPanelData(eventId),
    staleTime: 30_000, // 30s
    refetchOnWindowFocus: true,
    retry: 3,
  });
}

export function useUpdateSelection(eventId: string) {
  return useMutation({
    mutationFn: (selection: Record<ProductId, number>) =>
      updatePanelWithSelection(eventId, selection),
    // Optimistic updates would go here
  });
}

```

14.4 Jotai Integration

Purpose: Type-safe atoms for derived state and selection management.

```

import { atom, type PrimitiveAtom } from "jotai";
import { atomFamily } from "jotai/utils";
import { atomWithQuery } from "jotai-tanstack-query";
import type { PanelData, PanelItem, ProductId } from "../schemas";
import { panelKeys, fetchPanelData } from "../api";

// =====
// Query Atom (TanStack Query + Jotai)
// =====

export const panelDataQueryAtom = (eventId: string) =>
  atomWithQuery<PanelData>(() => ({
    queryKey: panelKeys.event(eventId),
    queryFn: () => fetchPanelData(eventId),
  }));

// =====
// Selection Atoms (per-item quantity)
// =====

/** Per-product selection state (quantity) */
export const selectionFamily = atomFamily((productId: ProductId) =>
  atom(0, (get, set, newQty: number) => {
    // Clamp to maxSelectable (authoritative)
    const panel = get(panelDataQueryAtom); // Assumes eventId in scope
    const item = panel.items.find((it) => it.product.id === productId);
    const max = item?.commercial.maxSelectable ?? 0;
    const clamped = Math.max(0, Math.min(newQty, max));
    set(selectionFamily(productId), clamped);
  })
);

// =====
// Derived State Atoms (§8 rendering composition)
// =====

/** Row presentation: normal | locked */
export const rowPresentationAtom = (item: PanelItem) =>
  atom((get) => {
    const { gating } = item.state;
    return gating.required &&
      !gating.satisfied &&
      gating.listingPolicy === "visible_locked"
  })

```

```

    ? ("locked" as const)
    : ("normal" as const);
});

/** Purchasable boolean (§8.1) */
export const isPurchasableAtom = (item: PanelItem) =>
atom((get) => {
  const { temporal, supply, gating } = item.state;
  return (
    temporal.phase === "during" &&
    supply.status === "available" &&
    (!gating.required || gating.satisfied)
  );
});

/** CTA kind (§8.1) */
export const ctaKindAtom = (item: PanelItem) =>
atom((get) => {
  const presentation = get(rowPresentationAtom(item));
  if (presentation === "locked") return "none" as const;

  const isPurchasable = get(isPurchasableAtom(item));
  if (isPurchasable) {
    return item.commercial.maxSelectable > 0
      ? ("quantity" as const)
      : ("none" as const);
  }

  const { supply, demand, temporal } = item.state;
  if (supply.status === "none" && demand.kind === "waitlist") {
    return "waitlist" as const;
  }
  if (temporal.phase === "before" && demand.kind === "notify_me") {
    return "notify" as const;
  }

  return "none" as const;
});

/** Quantity UI mode (§8.1) */
export const quantityUIAtom = (item: PanelItem) =>
atom((get) => {
  const presentation = get(rowPresentationAtom(item));
  const isPurchasable = get(isPurchasableAtom(item));
  const { maxSelectable } = item.commercial;

  if (presentation !== "normal" || !isPurchasable || maxSelectable <= 0) {
    return "hidden" as const;
  }

  return maxSelectable === 1 ? ("select" as const) : ("stepper" as const);
});

/** Price UI mode (§8.1) */
export const priceUIAtom = (item: PanelItem) =>
atom((get) => {
  const presentation = get(rowPresentationAtom(item));
  if (presentation === "locked") return "masked" as const;

  const isPurchasable = get(isPurchasableAtom(item));
  return isPurchasable ? ("shown" as const) : ("hidden" as const);
});

// =====
// Panel-level rollups

```

```
// =====

/** All visible items are sold out (no hidden gated considered) */
export const allVisibleSoldOutAtom = atom((get) => {
  const panel = get(panelDataQueryAtom);
  return panel.items.every((item) => item.state.supply.status === "none");
});

/** Any visible item is locked (requires showing access code CTA) */
export const anyLockedVisibleAtom = atom((get) => {
  const panel = get(panelDataQueryAtom);
  return panel.items.some(
    (item) =>
      item.state.gating.required &&
      !item.state.gating.satisfied &&
      item.state.gating.listingPolicy === "visible_locked"
  );
});

/** Selection is valid per orderRules (§5.3a) */
export const selectionValidAtom = atom((get) => {
  const panel = get(panelDataQueryAtom);
  const { orderRules } = panel.context; // REQUIRED per §4

  // Count selected types
  const selectedTypes = panel.items.filter(
    (item) => get(selectionFamily(item.product.id)) > 0
  );

  // Min types check
  if (selectedTypes.length < orderRules.minSelectedTypes) {
    return false;
  }

  // Min tickets per type check
  for (const item of selectedTypes) {
    const qty = get(selectionFamily(item.product.id));
    if (qty < orderRules.minTicketsPerSelectedType) {
      return false;
    }
  }

  return true;
});
```

14.5 TanStack Form Integration

Purpose: Type-safe form validation for access codes and user inputs.

```
import { useForm } from "@tanstack/react-form";
import { zodValidator } from "@tanstack/zod-form-adapter";
import { z } from "zod";

// =====
// Access Code Form
// =====

const AccessCodeFormSchema = z.object({
  code: z
    .string()
    .min(1, "Access code is required")
    .max(100, "Access code too long")
```

```

    .regex(/^[A-Za-z0-9-_$]+$/, "Invalid code format"),
  });

  type AccessCodeForm = z.infer<typeof AccessCodeFormSchema>;

  export function useAccessCodeForm(onSubmit: (code: string) => Promise<void>) {
    return useForm<AccessCodeForm, typeof zodValidator>({
      defaultValues: {
        code: "",
      },
      validatorAdapter: zodValidator,
      validators: {
        onSubmit: AccessCodeFormSchema,
      },
      onSubmit: async ({ value }) => {
        await onSubmit(value.code);
      },
    });
  }

  // Usage in component:
  // const form = useAccessCodeForm(async (code) => {
  //   await unlockWithCode(code);
  // });
  //
  // <form.Field name="code" validators={{ onChange: z.string().min(1) }}>
  //   {(field) => <input {...field.getInputProps()} />}
  // </form.Field>

  // =====
  // Quantity Selection Validation (client-side)
  // =====

  const QuantityInputSchema = z
    .object({
      quantity: z.number().int().nonnegative(),
      maxSelectable: z.number().int().nonnegative(),
    })
    .refine(
      (data) => data.quantity <= data.maxSelectable,
      "Quantity exceeds maximum allowed"
    );

  export function validateQuantityInput(
    quantity: number,
    maxSelectable: number
  ): boolean {
    const result = QuantityInputSchema.safeParse({ quantity, maxSelectable });
    return result.success;
  }

```

14.6 Validation Patterns & Error Handling

```

// =====
// Custom Error Map (user-friendly messages)
// =====

import { z } from "zod";

const customErrorMap: z.ZodErrorMap = (issue, ctx) => {
  if (issue.code === z.ZodIssueCode.invalid_type) {
    if (issue.expected === "string") {

```

```

    return { message: "Expected text" };
  }
  if (issue.expected === "number") {
    return { message: "Expected a number" };
  }
}

if (issue.code === z.ZodIssueCode.invalid_enum_value) {
  return {
    message: `Must be one of: ${issue.options.join(", ")}`,
  };
}

if (issue.code === z.ZodIssueCode.too_small) {
  if (issue.type === "string") {
    return { message: `Must be at least ${issue.minimum} characters` };
  }
  if (issue.type === "number") {
    return { message: `Must be at least ${issue.minimum}` };
  }
}

// Fallback to default message
return { message: ctx.defaultError };
};

z.setErrorMap(customErrorMap);

// =====
// Validation Helpers
// =====

/** Parse with detailed error logging (development) */
export function parseWithLogging<T>({
  schema: z.ZodSchema<T>,
  data: unknown,
  context: string
}): T {
  const result = schema.safeParse(data);

  if (!result.success) {
    console.group(`[Validation Error] ${context}`);
    console.error("Raw data:", data);
    console.error("Errors:", result.error.format());
    console.groupEnd();
    throw new Error(`Invalid ${context}: ${result.error.message}`);
  }

  return result.data;
}

/** Safe parse with fallback */
export function parseWithFallback<T>({
  schema: z.ZodSchema<T>,
  data: unknown,
  fallback: T
}): T {
  const result = schema.safeParse(data);
  return result.success ? result.data : fallback;
}

```


A) Complete type flow (server → atoms → components)

```
// 1. Server response validated by TanStack Query
const { data: panel, isLoading, error } = usePanelData(eventId);
//   ^? PanelData (inferred from PanelDataSchema)

// 2. Atoms derive presentation from validated data
const rowStates = panel.items.map((item) => ({
  key: item.product.id,
  presentation: rowPresentationAtom(item),
  isPurchasable: isPurchasableAtom(item),
  cta: ctaKindAtom(item),
  quantityUI: quantityUIAtom(item),
  priceUI: priceUIAtom(item),
}));

// 3. Component consumes typed state
function ProductRow({ item }: { item: PanelItem }) {
  const [quantity, setQuantity] = useAtom(selectionFamily(item.product.id));
  const presentation = useAtomValue(rowPresentationAtom(item));
  const cta = useAtomValue(ctaKindAtom(item));
  //   ^? "quantity" | "waitlist" | "notify" | "none"

  if (presentation === "locked") {
    return <LockedRow item={item} />;
  }

  return <NormalRow item={item} cta={cta} quantity={quantity} />;
}
```

B) Dinero formatting with type safety

```
import { dinero, toDecimal } from "dinero.js";
import type { DineroSnapshot } from "./schemas";

function formatPrice(snapshot: DineroSnapshot): string {
  const price = dinero(snapshot);
  return toDecimal(price, ({ value, currency }) =>
    new Intl.NumberFormat("en-US", {
      style: "currency",
      currency: currency.code,
    }).format(Number(value))
  );
}

// Usage (type-safe):
const priceText = formatPrice(item.commercial.price);
//   ^? string
```

C) Access code form with validation

```
function AccessCodeInput() {
  const form = useAccessCodeForm(async (code) => {
    await unlockGatedItems(code);
  });

  return (
    <form
```

```

    onSubmit={e => {
      e.preventDefault();
      form.handleSubmit();
    }}
  >
  <form.Field
    name="code"
    validators={{
      onChange: z.string().min(1, "Enter a code"),
    }}
  >
    {(field) => (
      <div>
        <input
          {...field.getInputProps()}
          placeholder="Access code"
          aria-label="Access code"
        />
        {field.state.meta.errors.length > 0 && (
          <span role="alert">{field.state.meta.errors[0]}</span>
        )}
      </div>
    )}
  </form.Field>
  <button type="submit" disabled={!form.state.canSubmit}>
    Apply Code
  </button>
</form>
);
}

```

14.8 Rationale

Why single source schemas?

- **Type safety:** One schema → one type. No drift between runtime validation and compile-time types.
- **Fail-fast:** Server bugs (schema changes, new fields) are caught at the API boundary, not deep in component trees.
- **Documentation:** Schemas are self-documenting; they show exactly what fields exist and their constraints.

Why validate at the boundary?

- **Trust but verify:** Server is authoritative, but network/proxies can corrupt. Validation ensures we're rendering valid data.
- **Debug aid:** Validation errors pinpoint exactly which field failed and why (better than `undefined is not a function`).
- **Security:** Prevents XSS/injection if server is compromised or returns unexpected data.

Why Zod 4?

- **Codecs:** Bi-directional parsing (incoming JSON → TypeScript, TypeScript → outgoing JSON).
- **Performance:** Faster parsing than v3; critical for high-frequency updates (selection changes).
- **Better errors:** More readable error messages for debugging and user feedback.
- **Ecosystem:** First-class support in TanStack Form, TanStack Query adapters.

Why brand types?

- **Nominal typing:** Prevents mixing `ProductId` with plain `string` (compile-time safety).
- **Intent clarity:** `ProductId` is clearer than `string` in function signatures.
- **Refactor safety:** If we change ID format (e.g., UUIDs → NanoIDs), only schema changes; usage sites are protected.

14.9 Tests (schema validation)

```

import { describe, it, expect } from "vitest";
import { PanelDataSchema, PanelItemSchema } from "../schemas";

// Test helper: minimal valid context (required fields)
const validContext = {
  orderRules: {
    types: "multiple" as const,
    typesPerOrder: "multiple" as const,
    ticketsPerType: "multiple" as const,
    minSelectedTypes: 0,
    minTicketsPerSelectedType: 0,
  },
  panelNotices: [],
  effectivePrefs: {
    showTypeListWhenSoldOut: true,
    displayPaymentPlanAvailable: false,
  },
};

describe("PanelDataSchema", () => {
  it("validates minimal valid payload", () => {
    const minimal = {
      context: validContext,
      sections: [{ id: "main", label: "Tickets", order: 1 }],
      items: [],
      pricing: {
        currency: { code: "USD", base: 10, exponent: 2 },
        lineItems: [],
      },
    };

    const result = PanelDataSchema.safeParse(minimal);
    expect(result.success).toBe(true);
  });

  it("rejects unknown top-level fields", () => {
    const withUnknown = {
      context: validContext,
      sections: [{ id: "main", label: "Tickets", order: 1 }],
      items: [],
      pricing: {
        currency: { code: "USD", base: 10, exponent: 2 },
        lineItems: [],
      },
      unknownField: "should be rejected",
    };

    const result = PanelDataSchema.safeParse(withUnknown);
    expect(result.success).toBe(false);
  });

  it("rejects payload with currency mismatch", () => {
    const invalid = {
      context: validContext,
      sections: [{ id: "main", label: "Tickets", order: 1 }],
      items: [
        {
          product: { id: "prod1", name: "GA", type: "ticket" },
          state: {
            temporal: { phase: "during", reasons: [] },
            supply: { status: "available", reasons: [] },
            gating: {
              required: false,
              satisfied: true,
            },
          },
        },
      ],
    };

    const result = PanelDataSchema.safeParse(invalid);
    expect(result.success).toBe(false);
  });
});

```

```

        listingPolicy: "omit_until_unlocked",
        reasons: [],
      },
      demand: { kind: "none", reasons: [] },
      messages: [],
    },
    commercial: {
      price: {
        amount: 5000,
        currency: { code: "EUR", base: 10, exponent: 2 }, // Mismatch
        scale: 2,
      },
      feesIncluded: false,
      maxSelectable: 1,
    },
    display: { badges: [], showLowRemaining: false },
  },
],
pricing: {
  currency: { code: "USD", base: 10, exponent: 2 },
  lineItems: [],
},
};

const result = PanelDataSchema.safeParse(invalid);
expect(result.success).toBe(false);
expect(result.error?.issues[0].message).toContain("currency");
});

it("rejects item with duplicate product IDs", () => {
  const invalid = {
    context: validContext,
    sections: [{ id: "main", label: "Tickets", order: 1 }],
    items: [
      {
        product: {
          id: "prod1",
          name: "GA",
          type: "ticket",
        } /* ... minimal required fields omitted for brevity */,
      },
      {
        product: {
          id: "prod1",
          name: "VIP",
          type: "ticket",
        } /* ... minimal required fields omitted for brevity */,
      },
    ],
  },
  pricing: {
    currency: { code: "USD", base: 10, exponent: 2 },
    lineItems: [],
  },
};

const result = PanelDataSchema.safeParse(invalid);
expect(result.success).toBe(false);
expect(result.error?.issues[0].message).toContain("unique");
});
});

describe("PanelItemSchema", () => {
  it("rejects omit_until_unlocked item that should be omitted", () => {
    const invalid = {
      product: { id: "secret", name: "Secret", type: "ticket" },

```

```

state: {
  temporal: { phase: "during", reasons: [] },
  supply: { status: "available", reasons: [] },
  gating: {
    required: true,
    satisfied: false, // Not satisfied
    listingPolicy: "omit_until_unlocked", // Should be omitted
    reasons: [],
  },
  demand: { kind: "none", reasons: [] },
  messages: [],
},
commercial: {
  price: {
    amount: 9000,
    currency: { code: "USD", base: 10, exponent: 2 },
    scale: 2,
  },
  feesIncluded: false,
  maxSelectable: 0,
},
display: { badges: [], showLowRemaining: false },
};

const result = PanelItemSchema.safeParse(invalid);
expect(result.success).toBe(false);
expect(result.error?.issues[0].message).toContain("omit_until_unlocked");
});
});

describe("MachineCodeSchema", () => {
  it("accepts valid snake_case codes", () => {
    expect(MachineCodeSchema.safeParse("sold_out").success).toBe(true);
    expect(MachineCodeSchema.safeParse("requires_code").success).toBe(true);
    expect(MachineCodeSchema.safeParse("remaining_low").success).toBe(true);
  });

  it("rejects non-snake_case codes", () => {
    expect(MachineCodeSchema.safeParse("SoldOut").success).toBe(false);
    expect(MachineCodeSchema.safeParse("sold-out").success).toBe(false);
    expect(MachineCodeSchema.safeParse("Sold Out").success).toBe(false);
  });
});

```

14.10 Invariants & Guardrails

Schema discipline

- **Root schema strict:** Use `.strict()` on `PanelDataSchema` so clients reject unknown top-level keys (SSR/hydration consistency).
- **Nested strictness:** Use `.strict()` for nested object schemas to catch unknown sub-fields.
- **MUST** derive TypeScript types via `z.infer<>` (no manual types).
- **MUST** validate all server responses with root schema (`PanelDataSchema`).
- **MUST NOT** use Zod transformers for business logic (display formatting only).
- **MUST NOT** create separate "API types" and "UI types" (single source).

Integration rules

- **TanStack Query:** Parse response with `PanelDataSchema.safeParse()` before returning.
- **TanStack Form:** Use `zodValidator` adapter; validate on submit (not `onChange` for performance).
- **Jotai:** Atom types inferred from schemas; atoms do not re-validate (already validated at boundary).

Error handling

- Development: Log full validation errors with `error.format()`.
- Production: Show generic error to user; log details to error tracking.
- Never expose raw Zod errors to end users (use custom error map).

14.11 Developer Checklist

Schema authoring:

- ☐ Root `PanelDataSchema` uses `.strict()` (reject unknown top-level keys)
- ☐ Nested object schemas use `.strict()` (catch unknown sub-fields)
- ☐ Optional fields with server defaults use `.default()` (no client-invented defaults)
- ☐ Enums use `z.enum()` with exhaustive values (matches spec §§3–12)
- ☐ Money fields use `DineroSnapshotSchema` (never plain numbers)
- ☐ Machine codes validated with `MachineCodeSchema` (snake_case regex)

Type safety:

- ☐ All types exported via `z.infer<typeof Schema>`
- ☐ No manual TypeScript interfaces that duplicate schemas
- ☐ Branded types used for IDs (`ProductId`, `SectionId`)
- ☐ Function signatures use inferred types, not `any` or `unknown`

Validation boundaries:

- ☐ TanStack Query validates responses with `.safeParse()` before returning
- ☐ Form submissions validate with TanStack Form + `zodValidator`
- ☐ Atoms use inferred types but do not re-validate (trust boundary validation)
- ☐ Custom error map provides user-friendly messages (not technical Zod errors)

Integration:

- ☐ Query keys typed and hierarchical (`panelKeys.*`)
- ☐ Atom families use branded types for parameters (`ProductId` not `string`)
- ☐ Derived atoms reference schema types (e.g., `PanelItem`, not ad-hoc interfaces)
- ☐ Form hooks use schema-derived types for `onSubmit` callbacks

Testing:

- ☐ Valid minimal payload passes validation
- ☐ Unknown fields rejected (`.strict()` enforcement)
- ☐ Currency mismatch rejected (cross-field validation)
- ☐ Duplicate product IDs rejected
- ☐ Gating invariants validated (`omit_until_unlocked` items must be absent)
- ☐ Machine code format validated (snake_case)

14.12 Migration Notes (schema evolution)

Atomic Deployment Model: With TanStack Start, server + client always deploy together. This simplifies schema evolution significantly.

Adding new optional fields

```
// Old schema
const OldSchema = z
  .object({
    name: z.string(),
  })
```

```

    .strict();

// New schema (add optional field)
const NewSchema = z
  .object({
    name: z.string(),
    description: z.string().optional(), // New optional field
  })
  .strict();

```

Deploy: Update schema → deploy server + client together → done. Both sides know about the new field immediately; no coordination complexity.

Changing field types (atomic deploy)

```

// Changing field type
const Updated = z.object({
  price: DineroSnapshotSchema, // Changed from z.number()
  // ✅ Deploy server + client together; no version skew
});

// Adding enum values
const SupplySchema = z.object({
  status: z.enum(["available", "none", "unknown", "pending"]), // Added "pending"
  // ✅ Update schema + deploy atomically; strict validation catches issues
});

```

Key Insight: With atomic deploys, "breaking changes" just means "update the schema and deploy." No multi-version support needed.

Schema versioning (future)

```

// Version discriminator for major contract changes
const PanelDataV1Schema = PanelDataSchema.extend({
  version: z.literal("1.0"),
});

const PanelDataV2Schema = z.object({
  version: z.literal("2.0"),
  // ... new structure
});

const VersionedPanelDataSchema = z.discriminatedUnion("version", [
  PanelDataV1Schema,
  PanelDataV2Schema,
]);

```

.

.



.

.

Appendix — Authoring & Migration Notes (non-normative)

This appendix is **guidance for authors and implementers**. It explains naming choices, “forbidden → preferred” mappings, and practical “before/after” examples. The body of the spec remains the only normative source.

A. Purpose & Scope

- Keep authors aligned on **names**, **axes**, and **message channels**.
- Prevent re-introducing past mistakes (e.g., **inventory**, ad-hoc banners, dual messaging systems).
- Offer concrete **before/after** payload examples you can copy-paste into fixtures.
- Capture **security rationale** (zero-leak gating) in one place.

B. Axis & Field Naming — Migration Cheatsheet

Use this when editing schemas, payloads, and fixtures. Items marked **removed** no longer exist in the contract; don’t mention them outside this appendix.

Area	Previously used (do not ship)	Ship now (canonical)	Notes
Availability axis	availability.* , stock , inventory	supply.status , supply.reasons , supply.remaining?	"inventory/stock" are forbidden terms.
Demand axis	demandCapture	demand.kind ∈ none \ waitlist \ notify_me	Single noun, standard CTAs.
Gating visibility	visibilityPolicy: "visible" \ "hidden"	listingPolicy: "omit_until_unlocked" \ "visible_locked"	"Sendability" is explicit.
Hidden gating hint	Row placeholders, counts of hidden SKUs	context.gatingSummary.hasHiddenGatedItems: boolean	Boolean hint only; no leakage.
Row text channels	reasonTexts , microcopy split	state.messages[] (+ optional copyTemplates)	One display channel per row.
Panel banners	dynamicNotices , client-invented banners	context.panelNotices[]	One banner channel, server-authored.
Admin/approval	admin axis, approvalRequired	Removed	No approval/request flow in contract.
Payment plan surf.	Per-row "Payment Plan" badges	Panel notice payment_plan_available	Order-level concept.
Code style	camelCase , free text	snake_case reason codes + payload copy for UI strings	Machine stable + localizable.

C. “Before → After” JSON Patterns

C1. Availability → Supply (and single source for “Sold Out”)

Before (don’t use):

```
{
  "availability": { "status": "soldOut" },
  "reasonTexts": { "sold_out": "Sold Out" }
}
```


After (ship this):

```
{
  "supply": { "status": "none", "reasons": ["sold_out"] },
  "state": {
    "messages": [
      {
        "code": "sold_out",
        "text": "Sold Out",
        "placement": "row.under_quantity",
        "priority": 100
      }
    ]
  }
}
```

C2. Gating zero-leak default (`omit_until_unlocked`)**Before (leaky placeholders):**

```
{
  "product": { "name": "VIP Secret" },
  "gating": { "required": true, "satisfied": false },
  "price": 15000
  // present but disabled; leaks name/price
}
```

After (no leakage pre-unlock):

```
// Item is omitted entirely from items[]
"context": {
  "gatingSummary": { "hasHiddenGatedItems": true },
  "panelNotices": [
    { "code": "requires_code", "variant": "info", "text": "Enter access code to view tickets", "priority": 90 }
  ]
}
```

Post-unlock, the item appears:

```
{
  "product": { "id": "prod_secret", "name": "VIP Secret", "type": "ticket" },
  "state": {
    "gating": {
      "required": true,
      "satisfied": true,
      "listingPolicy": "omit_until_unlocked"
    },
    "supply": { "status": "available" }
  },
  "stateMessages": [
    {
      "code": "unlocked",
      "text": "Unlocked with your code",
      "placement": "row.under_title"
    }
  ]
}
```

```
  ],  
  "commercial": { "maxSelectable": 2 }  
}
```

C3. Panel banners — single channel

Before (client-invented “Sold Out” banner):

```
// nothing in context, client decides to show banner when all rows sold out
```

After (server decides, client renders):

```
"context": {  
  "panelNotices": [  
    { "code": "event_sold_out", "variant": "info", "text": "Event sold out", "priority": 100  
    }  
  ]  
}
```

C4. Unified per-row messages (no **reasonTexts** at row level)

Before (dual channels):

```
"reasonTexts": { "remaining_low": "Only 3 left!" },  
"microcopy": [{ "code": "remaining_low" }]
```

After (one channel):

```
"state": {  
  "messages": [  
    {  
      "code": "remaining_low",  
      "text": "Only 3 left!",  
      "placement": "row.under_quantity",  
      "variant": "info",  
      "priority": 60  
    }  
  ]  
}
```

C5. Demand CTAs

Before:

```
"demandCapture": { "kind": "waitlist" }
```

After:

```
"demand": { "kind": "waitlist" }
```

CTA mapping is mechanical: `supply.status="none" + demand.kind="waitlist"` → “Join Waitlist”.

C6. Payment plans (panel-level, not per row)

Before:

```
"display": { "badges": ["PaymentPlanAvailable"] }
```

After:

```
"context": {
  "panelNotices": [
    { "code": "payment_plan_available", "variant": "info", "text": "Payment plans available
at checkout", "priority": 50 }
  ]
}
```

D. Security Rationale — Zero-Leak Gating

- **Threat model:** Scrapers and curious users harvest SKU names/prices via hidden or disabled rows.
 - **Mitigation:** `omit_until_unlocked` keeps gated SKUs out of `items[]` until server-validated unlock.
 - **Hinting:** `gatingSummary.hasHiddenGatedItems` is the only allowed signal; it’s **boolean**, not counts.
 - **Unlock flow:** client submits code → server validates + returns a short-lived token → subsequent panel load includes unlocked items.
 - **Client obligations:** never log codes/tokens; never cache unlocked payloads across accounts; never infer or display price for locked items.
-

E. Authoring Heuristics (keep payloads crisp)

- **Keep axes orthogonal.** Causes go into axis `reasons[]`; user text goes into `state.messages[]`.
 - **Prefer explicit over clever.** If you want a banner, send `panelNotices[]`. If you want inline urgency, send a message with `placement`.
 - **No duplicates.** The same concept must not appear in more than one axis or channel.
 - **Short JSON, many states.** One tiny example per state beats one giant all-states example.
 - **Priority first.** Higher priority messages and notices should be rendered first; set `priority` in payload.
-

F. Do / Don’t Cookbook

Do

- Use `supply.status="none" + state.messages[]` → “Sold Out”.
- Use `panelNotices[]` for “Event sold out”, “Payment plans available”, “Enter access code”.
- Use `gating.listingPolicy="omit_until_unlocked"` by default.
- Use `commercial.maxSelectable` as the **only** UI clamp.
- Use `copyTemplates` for templated strings; let server control phrasing.

Don’t

- Don’t hardcode UI text (ever).
- Don’t compute totals/countdowns/limits client-side.
- Don’t ship gated rows just to show a lock if you need zero-leak.

- Don't invent banners or CTAs not present in payload.
- Don't use `inventory`, `stock`, or `availability` as field names.

G. Reason Code Registry (authoring reference)

Codes are machine tokens; UI text comes from `state.messages[]` or `panelNotices[]`.

- **Temporal:** `outside_window`, `sales_ended`
- **Supply:** `sold_out`, `remaining_low`
- **Gating:** `requires_code`, `code_invalid`, `code_verified`
- **Demand:** `waitlist_available`, `notify_available`
- **Panel-level:** `event_sold_out`, `payment_plan_available`, `requires_code`

Keep codes **short and specific**. Use `snake_case`. Avoid tense in codes; tense belongs in copy.

H. Message Placement Slots (recommended)

Use a small set of placements so UI composition is predictable:

Placement	Intended spot
<code>row.under_title</code>	Small text just below the row title
<code>row.under_price</code>	Inline with or under price
<code>row.under_quantity</code>	Under the quantity control / CTA area
<code>row.footer</code>	At bottom of the row
<code>row.cta_label</code>	CTA button label text for the row

Note: Panel-level banners use `context.panelNotices[]`, not row message placements.

I. Fixture Set (starter pack)

Create and snapshot these six fixtures in Storybook:

1. **Available:** `supply.available`, purchasable.
2. **Sold out + waitlist:** `supply.none`, `demand.waitlist`.
3. **Visible locked:** `gating.required && !satisfied`, `listingPolicy="visible_locked"`; price masked.
4. **Omit until unlock:** gated item omitted; `gatingSummary.hasHiddenGatedItems=true`; access-code banner.
5. **Public sold out + hidden gated:** all visible `none`, summary `true` (banner invites code).
6. **Payment plan:** `panelNotices[payment_plan_available]`.

J. FAQ (for team alignment)

- Q: Why not compute countdowns client-side?** A: Timezones, daylight saving edges, and sales pauses make it brittle. Server owns truth; client renders text.
- Q: Can we show price for a locked row?** A: Not for secure flows. If marketing wants a tease, use `visible_locked` with masked price and an explanatory message.
- Q: Can we surface "total remaining across event"?** A: Only if the server sends it (e.g., via a panel notice or an explicit rollout field). The client must not sum.
- Q: Where do "max N per order" messages live?** A: Client-reactive microcopy using `context.clientCopy` (strings from server), triggered by `commercial.maxSelectable`.

K. Copy & Templates — Quick Patterns

- **Template style:** "Only {count} left!", "Max {max} per order.", "Sales end {date_local}."
- **Interpolate server-supplied params** only; don't compute `date_local` or counts in the client.
- **Severity:** `neutral` | `info` | `warn` | `error` for styling; keep content terse.

L. Schema Cliff Notes (enums & shapes)

```
// Axes
supply.status: "available" | "none" | "unknown"
gating: {
  required: boolean
  satisfied: boolean
  listingPolicy: "omit_until_unlocked" | "visible_locked"
  reasons: string[]
}
demand.kind: "none" | "waitlist" | "notify_me"

// Messaging
state.messages[]: { code: string; text?: string; params?: {}; placement: string; variant?:
"neutral" | "info" | "warning" | "error"; priority?: number }

// Banners
context.panelNotices[]: { code: string; text?: string; params?: {}; variant?: "neutral" |
"info" | "warning" | "error"; priority?: number }

// Hints
context.gatingSummary: { hasHiddenGatedItems: boolean }
commercial.maxSelectable: number
```

M. Lint the Payload (authoring checklist)

- ☐ No `inventory/stock/availability` fields.
- ☐ Every inline string is in `state.messages[]` (or properly templated); no duplicated strings.
- ☐ Gated items omitted when `listingPolicy="omit_until_unlocked"` && `!satisfied`.
- ☐ If **all** visible items are `supply.none` and **no** hidden gated items, include `event_sold_out` panel notice (or intentionally none).
- ☐ No approval/request fields or CTAs.
- ☐ `commercial.maxSelectable` present and consistent with purchasability.

N. Rationale Snapshots (why these choices stick)

- **One name per axis.** "Supply" ends the "availability/inventory" bikeshed and keeps code grep-able.
- **One row text channel.** `state.messages[]` prevents precedence fights and duplicate copy.
- **Zero-leak default.** Security trumps tease. If you want tease, opt-in with `visible_locked`.
- **Server as oracle.** If the server can't express it cleanly, the client shouldn't guess it.

O. Future Hooks (safe extensions)

- `demand.kind="standby_queue"` (virtual line) — CTA and copy remain data-driven.
- `messages[].placement="row.badge_tooltip"` — progressive disclosure slots without new channels.
- `panelNotices[].expiresAt` — server-controlled timeboxing of banners.

P. Migration Notes — ViewModel → State (for authoring hygiene)

We are greenfield going forward. These notes exist only to prevent legacy terms from reappearing in code or docs. The body of the spec uses the new names exclusively.

Authoritative renames (do not reintroduce old names)

```
- RowViewModel → RowState
- SectionViewModel → SectionState
- PanelViewModel → PanelState

- panelViewModelAtom → panelStateAtom
- rowViewModelAtom → rowStateAtom

- mapItemToRowVM → deriveRowState
- mapItemToRowViewModel → deriveRowState
```

Conceptual language shift

Old (MVVM)	New (React/Jotai)
view model	derived state
mapping to view model	deriving state from contract
view model atom	state atom
panel view model	panel state
row view model	row state

Example updates (illustrative)

```
- const rowVMs = items.map((it) => mapItemToRowVM(it, payload));
+ const rowStates = items.map((it) => deriveRowState(it, payload));

- export type RowViewModel = {
+ export type RowState = {
  key: string;
  presentation: RowPresentation;
  // ...
}
```

Rationale (short)

- Aligns with React vocabulary and Jotai’s atom model.
- Removes MVVM two-way binding baggage; our flow is one-directional.
- Keeps boundaries crisp: server contract → client-derived state → components.

Close-out

This appendix will grow with real fixtures and gotchas from implementation. When in doubt, simplify: one axis, one banner channel, one row message channel, and a boolean hint for hidden gated items.

Q. ReUI Filters + shadcn Field + TanStack Form + Jotai + Query

Purpose

Non-normative integration pattern showing how to pair ReUI Filters with shadcn Field for accessible layout, Jotai for state (no prop drilling), TanStack Form for optional form modeling, TanStack Query for data fetching, and TanStack Start (router) for URL/search-param sync — while respecting Product Panel guardrails (§§3–13).

Scope and fit

- Use for list/data filtering UIs around the panel (e.g., browsing events, add-ons directories), not to override server truths in the panel contract.
- Server remains the source of truth; filters shape the query, not the business decisions.

Constraints (align with spec)

- Copy: no client-invented strings for the panel; filter UI labels come from component config, not panel payload.
- Security: never leak gated info; don't log sensitive tokens; sanitize/validate filter payload on server.
- State: Jotai is the single source of truth for filter state; sync into TanStack Form only if needed.
- Query: include filters structurally in the Query key (stable/serializable) to drive invalidation/refetch.
- URL: optional sync to router search params; avoid encoding secrets.
- a11y: wrap filter UI in shadcn Field primitives for label/description/error consistency.

Blueprint

Install components

```
bunx shadcn@latest add @reui/filters
bunx shadcn@latest add field
```

Define filter atom and field config (typed)

- `filtersAtom = atom<Filter[]>([])` (use ReUI public types)
- `fields: FilterFieldConfig[] | FilterFieldGroup[]` from the page context

Accessible wrapper with shadcn Field

```
<Field>
  <FieldLabel>Filters</FieldLabel>
  <FieldDescription>Refine results</FieldDescription>
  <Filters fields={fields} filters={filters} onChange={setFilters} />
  <FieldError />
</Field>
```

Optional TanStack Form sync

- If filters belong to a form, bind via a `form.Field name="filters"` and pass `value/onChange` to `<Filters>`, or mirror Jotai → form using `form.setFieldValue('filters', filters)`.

TanStack Query integration

- Query key: `['dataset', { filters }]` (stable, JSON-serializable)
- Server endpoint validates a whitelist of fields/operators and values with Zod; never interpolate raw strings into SQL.

URL/search-param sync (optional)

- Encode filters as a compact, stable string; use TanStack Start router search params. Keep the Jotai atom and router in two-way sync with debounced updates.

Testing

- Unit: when `filtersAtom` changes, the Query key changes and refetches.
- a11y: Field renders label/description/error; no positive tabIndex; valid roles.
- Security: ensure server rejects unknown fields/operators; snapshot tests for payload validation.

Example (minimal, typed)

```
// atoms
type AppFilter = {
  id: string;
  field: string;
  operator: string;
  values: unknown[];
};
const filtersAtom = atom<AppFilter[]>([]);

function FiltersField({ fields }: { fields: any[] }) {
  const [filters, setFilters] = useAtom(filtersAtom);
  return (
    <Field>
      <FieldLabel>Filters</FieldLabel>
      <Filters fields={fields} filters={filters} onChange={setFilters} />
      <FieldError />
    </Field>
  );
}

// query
const { data } = useQuery({
  queryKey: ["dataset", { filters: useAtomValue(filtersAtom) }],
  queryFn: () => api.listItems({ filters: get(filtersAtom) }),
});
```

Do / Don't (quick)

- Do: keep Jotai as source; validate on server; include filters in queryKey; wrap in Field; sync URL when useful.
- Don't: hardcode panel strings; leak gated info; mutate queryKey types; store secrets in URL.

References

- ReUI Filters: reui.io/docs/filters
- shadcn Field: ui.shadcn.com/docs/components/field
- shadcn Field video: [YouTube](#)