

TMF622 Product Ordering Management API:

Overview and Role in TM Forum Open APIs

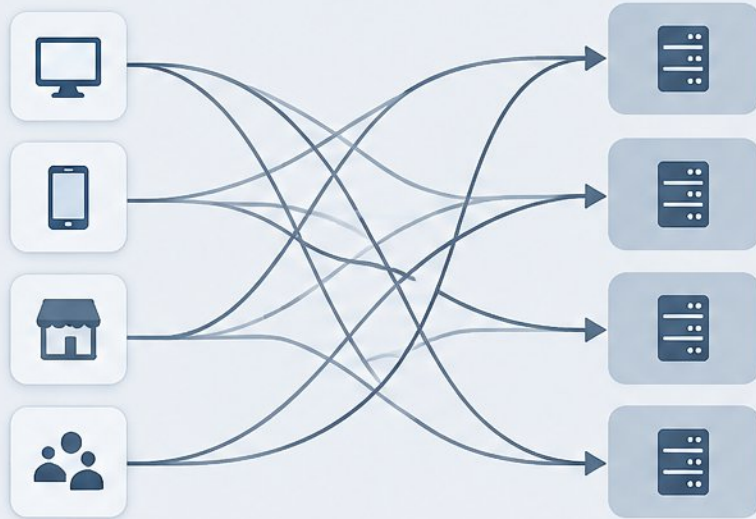
- TMF622 standardizes how product orders are placed, tracked, and managed
- Orders flow between customer and provider, or between partner providers
- Built from catalog offerings defined in TMF620, carrying pricing and options
- Stable version v5.0 (08 May 2026) moved to OpenAPI 3.0
- v5 introduced requestedInitialState for explicit order submission control
- Key dependencies: TMF620 Catalog, TMF637 Inventory, TMF641 Service Ordering
- Adopters: CSPs modernizing OSS/BSS, OMS/COM vendors, wholesale integration teams



Why Standardized Product Ordering Matters

Bespoke Ordering Interfaces

Complex, fragmented, slower to onboard



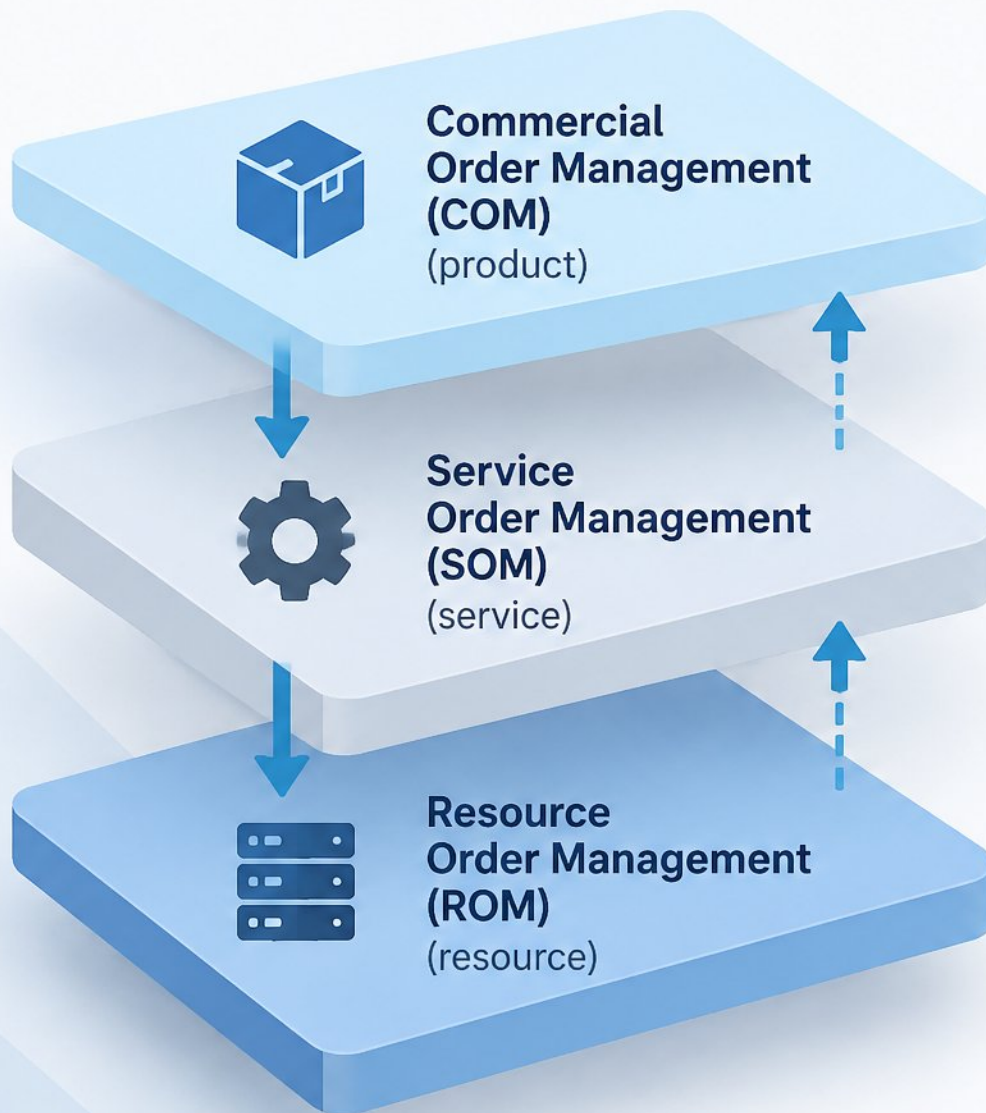
One Canonical Product Order Model

Normalize every channel into one order manager



- Bespoke ordering interfaces drive high integration cost and slow partner onboarding
- Order semantics drift across channels, causing inconsistent behavior
- TM Forum Open API program: 100+ REST, event-driven, and domain-specific APIs
- Shared guidelines, specifications, and conformance test kits for members
- Business drivers: faster time-to-market and multi-partner orchestration
- 1M+ Open API downloads as of early 2025 signal broad ecosystem adoption
- One canonical product order model normalizes every channel into one order manager

Product, Service, and Resource Ordering: Where TMF622 Fits

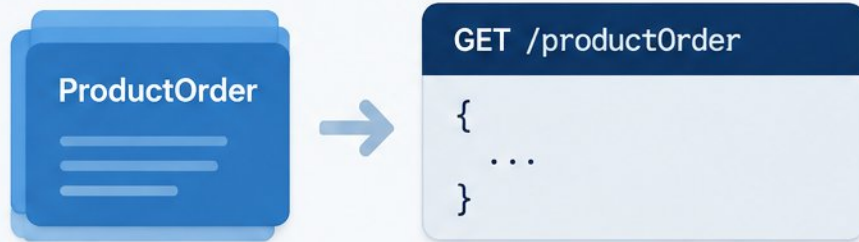


- Three order layers: COM (product), SOM (service), ROM (resource)
- TMF622 handles commercial product offerings; TMF641 targets Customer-Facing Services; TMF652 covers network resources
- Orders decompose top-down by catalog rules; completion rolls up bottom-up
- No 1:1 mapping: one 200 Mbps item may become 3 service and 7+ resource items
- Each layer owns its own order API, catalog, and inventory
- B2B2C chains product orders between providers, each owning only its commercial view

Core Resources: ProductOrder, ProductOrderItem, and References

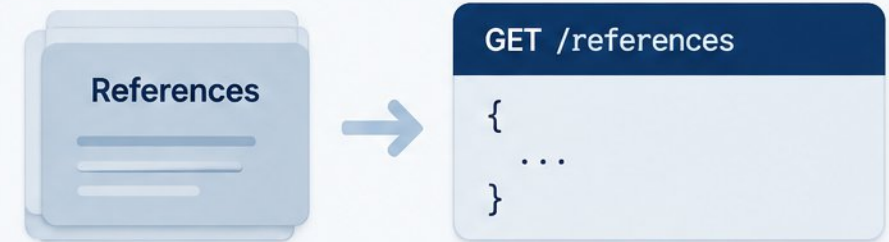
ProductOrder

- ProductOrder: id, href, externalId, category, priority (0 highest–4 lowest), state, related dates



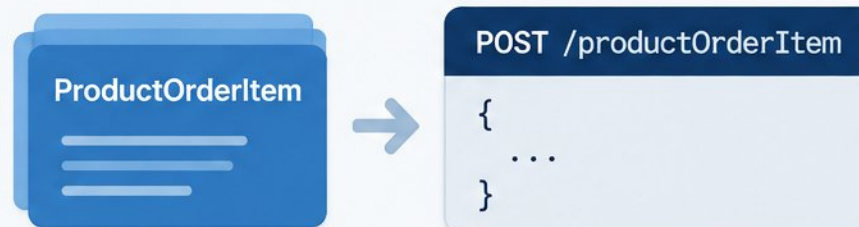
Context references

- Context references: billingAccount, relatedParty, channel, agreement, quote, note, orderRelationship



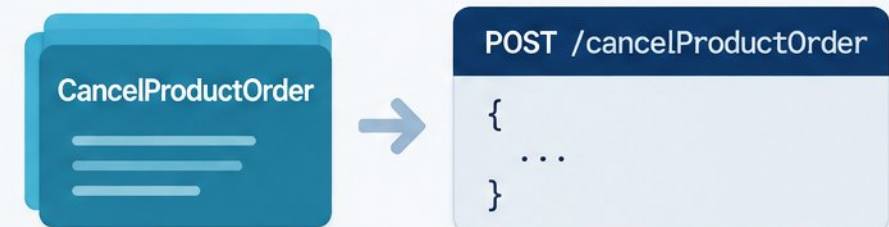
ProductOrderItem

- ProductOrderItem requires id (01, 02...) and action: add, modify, delete, noChange
- Optional item fields: quantity, state, productOffering, product, itemPrice, itemTerm, appointment

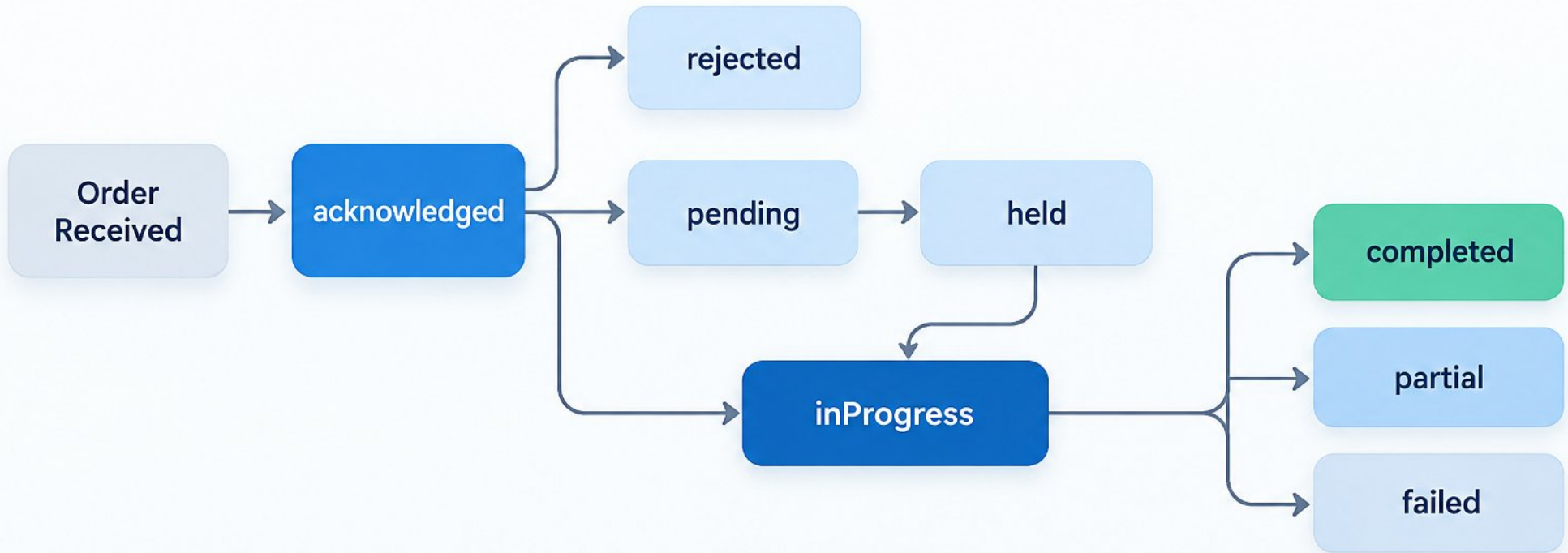


productCharacteristic and CancelProductOrder

- productCharacteristic captures customer choices as name/value pairs (TEL_MSISDN, downloadSpeed)
- CancelProductOrder task states: acknowledged, inProgress, done, terminatedWithErrors



Product Order Lifecycle and State Model



- Order states: acknowledged, rejected, pending, held, inProgress, completed, failed, partial
- Order state derives from item states: all completed = completed; mixed = partial
- pending waits on the requester; held pauses on a provider-side issue

- partial is terminal and real — decide on compensation, resubmission, or contact
- Pitfalls: improvised per-channel states, billing from capture, stale state during SOM/ROM
- Milestones and jeopardy alerts flag slippage or point of no return

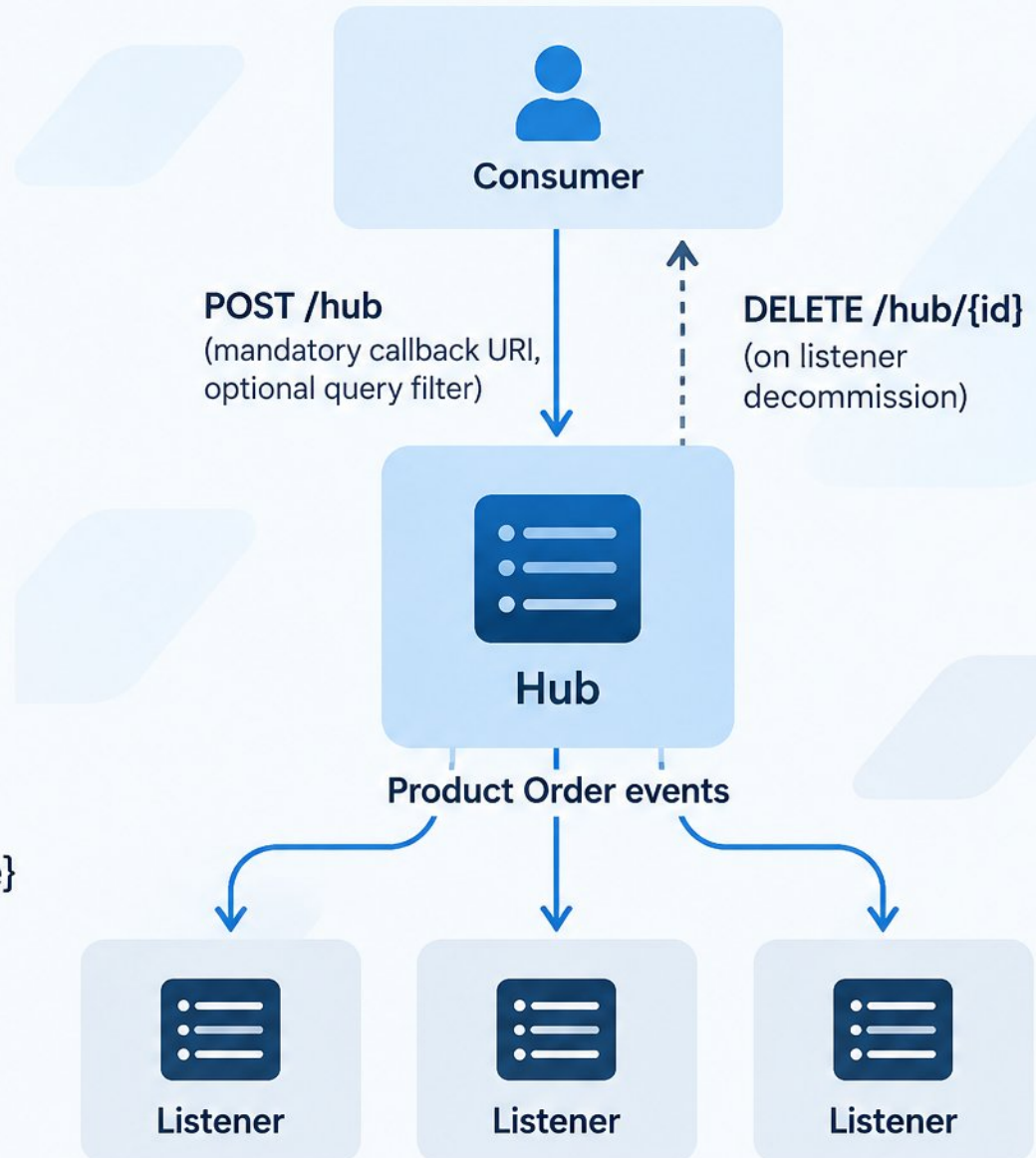
API Operations: CRUD, Query Parameters, and Cancellation

- POST /productOrder creates the order; GET retrieves one or lists with filters
- PATCH /productOrder/{id} performs partial update; DELETE is administrative only
- fields selects attributes; offset and limit paginate listing calls
- Create in acknowledged state, or draft held for later submission via requestedInitialState
- Cancellation task: POST /cancelProductOrder with its own state machine
- Design for 201 Created, 202 Accepted, and the 4xx/5xx error set
- Separate synchronous validation responses from asynchronous fulfillment progress



Events, Hub Registration, and Event-Driven Integration

- Register callback: POST /hub with mandatory callback URI, optional query filter
- Unregister anytime: DELETE /hub/{id} on listener decommission
- Product Order events: Create, StateChange, AttributeValueChange, Delete, InformationRequired
- Also Milestone, JeopardyAlert, ErrorMessage, plus CancelProductOrder events
- Clients implement POST /listener/{eventType} using the TMF event envelope
- Best practices: idempotent listeners keyed on eventId, quick 2xx, HTTPS, clean unsubscribe



Payload Modeling and Validation Rules

- At least one productOrderItem required; each item needs id and action
- productOffering references the commercial offer; productCharacteristic captures config
- Typed characteristics (String, Number, Object) selected via @type discriminator
- Change orders: pass full item or use extensions to signal the delta
- CTK enforces mandatory only — realistic deployments need optional fields
- Errors use code, reason, message, status, referenceError (TMF630)

Example productOrder payload

```
{
  "productOrderItem": [
    {
      "id": "...",
      "action": "...",
      "productOffering": {
        "id": "..."
      },
      "productCharacteristic": [
        {
          "name": "...",
          "@type": "String",
          "value": "..."
        },
        {
          "name": "...",
          "@type": "Number",
          "value": ...
        },
        {
          "name": "...",
          "@type": "Object",
          "value": { ... }
        }
      ]
    }
  ]
}
```

Each item needs id and action

productOffering references the commercial offer

Typed characteristics (String, Number, Object) selected via @type discriminator



Catalog
(offering)



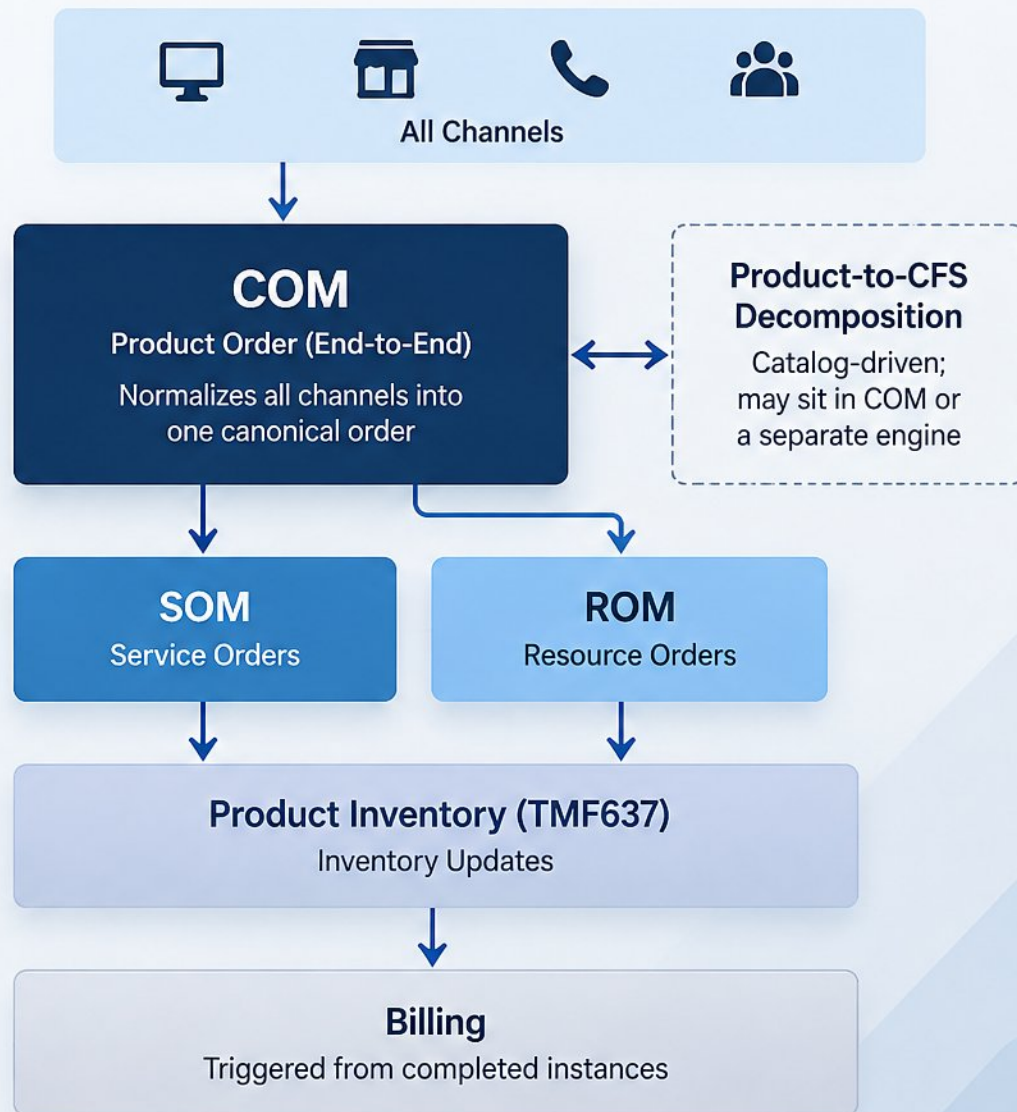
Product Order



Product Instance
(inventory)

Integration Patterns: COM, Decomposition, and Inventory Synchronization

- COM owns the product order end-to-end and normalizes all channels into one canonical order
- Product-to-CFS decomposition is catalog-driven; may sit in COM or a separate engine
- Treat TMF622 as a stable integration contract, not a workflow engine
- Sync: order submission, qualification checks, catalog queries. Async: fulfillment, activation, inventory updates
- Product Inventory (TMF637) triggers billing from completed instances, not order capture
- Avoid: synchronous inventory dependencies, embedding qualification in ordering, modeling internals on TMF schemas



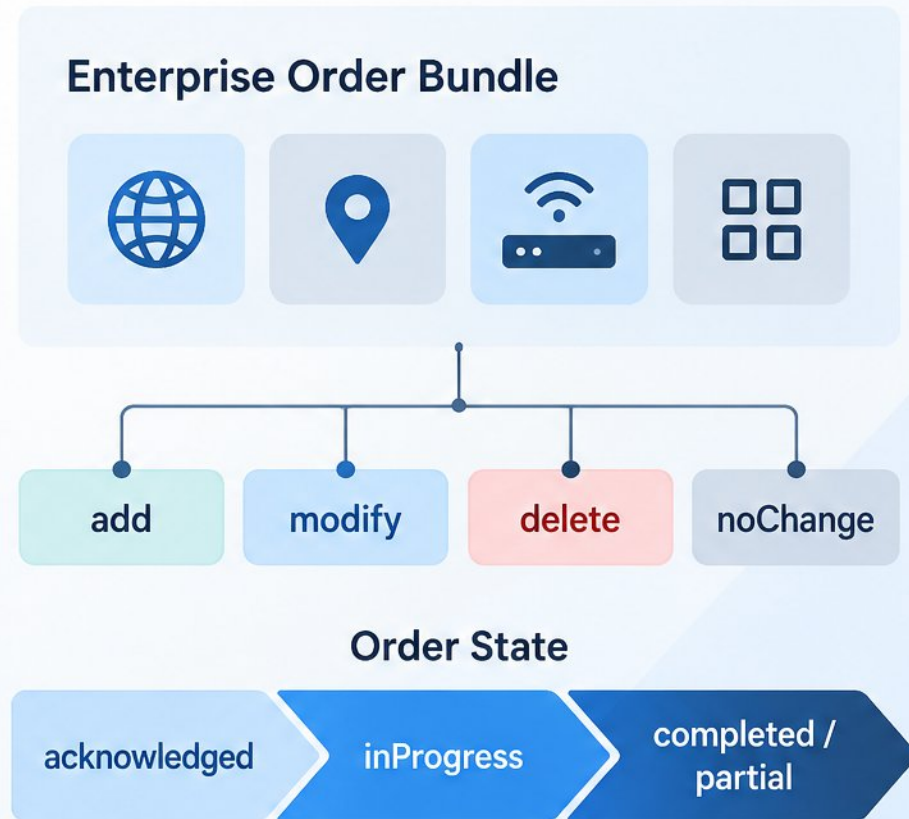
Scenario 1: B2C Digital-Channel Ordering

- Customer confirms a plan; quote submitted as POST /productOrder; order moves to acknowledged
- Validation covers address qualification, credit/fraud checks, duplicate detection — most fallout happens here
- Order moves to inProgress while service orders issue downstream; channel renders progress via state-change events
- productCharacteristic carries customer config: speed, SIM type, installation preference
- Backend-for-frontend APIs serve app and web over one TMF622 interface
- Reference: Orange Poland's Flex uses TM Forum Open APIs for digital ordering



Scenario 2: B2B and Enterprise Ordering

- Enterprise orders bundle broadband, static IP, router rental, and apps
- Each item carries its own action: add, modify, delete, or noChange
- category separates enterprise from residential, driving routing, SLA, and priority
- Modify orders read the current instance from TMF637 and send only the delta
- Unchanged items pass as noChange to give downstream context
- Approvals and contract terms map to pending or held hold points
- B2B exposure needs a stable, versioned contract with backward compatibility



Scenario 3: Wholesale and Partner Ordering

- Partners should see product offerings and commercial terms, not your service decomposition
- Same TMF622 contract serves retail and wholesale flows with different offerings and pricing
- Product orders can flow provider-to-provider, including termination validation back to the originator
- Submit endpoint may differ from event-receiving endpoint — plan distributed event routing
- Give partners clear version management; they integrate on their own release cycles
- KPIs: partner onboarding time, API vs. manual order volume, partner-reported error rates



Governance, Conformance, and Version Management

Conformance Journey



Catalog-to-order-to-inventory



- Open API conformance is self-applied, self-tested: run the CTK, then submit results for formal TM Forum validation



- TMF622 has an active CTK; v5.0 added out-of-the-box proxy settings for corporate environments



- CTK validates mandatory properties only — optional fields still matter to real customers



- Versioning follows semver: major changes break compatibility, minor versions preserve it; only the major version appears in the endpoint URL



- Consumers should tolerate new minor-version fields and unknown event types



- Aligning TMF622 with TMF620, TMF637, and TMF641 is the real governance task

Key Takeaways and Adoption Roadmap

-  TMF622 standardizes capture, tracking, and completion — but only with a well-modeled catalog
-  Success factors: catalog alignment, disciplined state governance, event-driven integration
-  First steps: inventory interfaces, run the CTK, pilot one scenario end to end
-  Decide what to standardize versus extend; extensions must not breach compliance
-  Build in resilience: idempotent listeners, correlation IDs, clean version management



PersonWise.

PersonWise • AI digital-human course platform



Scan the code or click anywhere to watch this course live, with voice Q&A

personwise.ai/t/b6887735/public