StoreFlow — A1.4 Shipping Engine & Fulfilment Workflows (Part 2)

7. Fulfilment Workflow Overview

The fulfilment subsystem handles the movement of an order from creation to completion.

It unifies both pickup and shipping workflows under a consistent state machine.

Core Principles:

• All fulfilment actions must be traceable.

• Merchant staff trigger all state changes.

• Customers observe changes in real-time.

• All fulfilment operations emit audit logs and WebSocket events.

• State transitions are strictly validated through domain rules.

8. Pickup Workflow – Detailed Lifecycle

Pickup lifecycle is designed for speed and simplicity.

Sequence:

1. Order Created → status = pending

2. Merchant accepts → status = accepted

3. Staff begin preparation → status = in_progress

4. Order ready at counter → status = ready

5. Customer collects → status = completed

Transitions & Timestamps:

• accepted_at set when status first becomes accepted

• ready_at set when status becomes ready

• completed_at set when customer collection is confirmed

Actor Roles:

• Manager or staff may update status.

• Owner may override status in exceptional cases.

WebSocket Events:

• OrderStatusUpdated broadcast to store channel.

• UI updates order card in real-time.

Error Handling:

• Cannot move backwards in lifecycle.

• Cannot transition from ready → in_progress.

• Completed is terminal.

9. Shipping Workflow – Detailed Lifecycle

Shipping lifecycle includes more states than pickup.

Sequence:

1. Order Created → pending

2. Merchant accepts → accepted

3. Merchant begins packing → packing

4. Order handed to carrier:

• tracking_code set

• tracking_url optional

• shipping_status = shipped

• status = shipped

5. Final delivery:

• shipping_status = delivered

• status = delivered

• completed_at timestamp set

Shipping Metadata:

• shipping_method stored from checkout

• shipping_name/address snapshot frozen on order

Real-Time Behavior:

• Customers see “Your order has shipped!” immediately.

• Tracking URL rendered on customer order detail page.

Validation:

• packing cannot occur if fulfilment_type != shipping

• shipped requires a valid tracking_code

• delivered is terminal state

10. Fulfilment Sequence Diagram (Text Version)

TEXT-BASED SEQUENCE DIAGRAM:

Customer → Storefront: Place Order

Storefront → Backend: POST /checkout (order created: pending)

Backend → WebSocket(store): OrderCreated event

Manager/Staff → Dashboard: Accept Order

Dashboard → Backend: PATCH /orders/{id}/status to accepted

Backend → WebSocket(store): OrderStatusUpdated(accepted)

Staff → Dashboard: Start Preparation (pickup) OR Start Packing (shipping)

Backend → WebSocket(store): OrderStatusUpdated(in_progress/packing)

Staff → Dashboard: Mark Ready (pickup) OR Mark Shipped (shipping)

Backend → WebSocket(store + customer): StatusUpdated(ready/shipped)

Customer → Storefront: Poll or receive real-time updates

Staff → Dashboard: Mark Completed/Delivered

Backend → WebSocket(store + customer): StatusUpdated(completed/delivered)

End.

11. Data Flow Through Fulfilment

Data Flow Steps:

1. Order is created with all snapshot data.

2. Fulfilment operations modify only:

- status

- shipping_status (shipping orders)

- tracking_code/url (shipping)

- timestamps

Invariant Data:

• Item prices

• Shipping cost

• Customer snapshot

• Product snapshot

These fields are immutable post-creation.

Persistence:

• Status updates executed inside DB transactions.

• Audit log inserted alongside status update, not after.

12. WebSocket Event Mapping

Each fulfilment state change emits:

EVENT: OrderStatusUpdated

Payload:

{

"order_id": 123,

"public_id": "ABCD1234",

"old_status": "accepted",

"new_status": "in_progress",

"timestamp": "...",

"fulfilment_type": "pickup",

"store_id": 77

}

Channels:

• store.{store_id}.orders → Dashboard staff/manager/owner

• customer.{public_id}.order → Customer order tracking UI

Customer Privacy Note:

• Customer-level channel uses public_id (not customer_id).

• Prevents leaking internal numeric identifiers.

13. Fulfilment Logic – Pseudocode (Pickup)

```
function updateOrderStatus(order, newStatus, user):
ensure user.canUpdate(order)
ensure validTransition(order.status, newStatus)

begin transaction
oldStatus = order.status
order.status = newStatus

if newStatus == 'accepted':
order.accepted_at = now()

if newStatus == 'ready':
order.ready_at = now()

if newStatus == 'completed':
order.completed_at = now()

save(order)

createAuditLog(
merchant_id=order.merchant_id,
user_id=user.id,
entity='order',
entity_id=order.id,
action='status_changed',
meta={oldStatus,newStatus}
)

commit

broadcast OrderStatusUpdated event
```

14. Fulfilment Logic – Pseudocode (Shipping)

```
function shipOrder(order, trackingCode, trackingUrl, user):

ensure order.fulfilment_type == 'shipping'

ensure order.status in ['accepted','packing']

begin transaction

oldStatus = order.status

order.status = 'shipped'

order.shipping_status = 'shipped'

order.tracking_code = trackingCode

order.tracking_url = trackingUrl

order.ready_at = now()

save(order)

createAuditLog(...)

commit

broadcast OrderStatusUpdated + ShippingUpdated
```

15. Caching & Performance Considerations in Fulfilment

Fulfilment operations must be extremely fast:

Caching:

• Fulfilment results are not cached—these are writes.

• However, downstream consumers (dashboard UI) use WebSockets to avoid polling.

Indexes:

• idx_orders_status ensures fast filtering of active orders.

• idx_orders_store_status supports store-level operations board.

Concurrency:

• Staff may click actions simultaneously.

• Use DB transaction isolation to prevent race conditions.

Safety:

• In rare cases, conflicting updates will cause a soft failure:

e.g., two staff attempt to change status at the same time.

• UI should gracefully refresh state after conflict.