Below is **clear, structured, beginner-friendly documentation** for your entire assembly-order system — including architecture, concurrency model, state machine flow, event streaming, sequencing guarantees, and frontend consumption.

🏭 Assembly Order Processing System — Full Documentation

A Beginner-Friendly Guide to Understanding the Architecture, Concurrency Model, State Machine, and Event Flow

---

1. 🧭 High-Level Overview

This system simulates an *automated assembly line* that processes transport orders.

Each **order** goes through:

1. Creation
2. Confirmation
3. Transport Arrival
4. Assembly
5. Validation
6. Completion / Failure

These steps are orchestrated using:

- **A per-order coroutine state machine** (`AssemblyStateMachine`)
- **A global queue** to serialize order creation
- **A single-assembly-at-a-time semaphore** to prevent parallel assembly
- **MutableStateFlow** to propagate state transitions
- **Server-Sent Events (SSE)** to stream logs, state transitions, and status updates to the frontend

The system guarantees:

✔ Orders are created in queue order ✔ Only **one order is assembled at any given time** ✔ Real-time logs and state transitions are streamed to the UI ✔ Every order runs inside its own isolated coroutine scope ✔ Frontend receives state, logs, statuses separately

---

2. 🧩 Core Components

The system has four major pieces:

🏭 1. AssemblyService

The **central orchestrator**.

Responsibilities:

- Receives order creation requests
- Manages the order queue (max 100)
- Spawns an isolated coroutine for each order
- Maintains per-order state (`orderStates`)
- Publishes events to frontend via `_events` (a `MutableSharedFlow`)
- Ensures only one assembly happens at a time using `assemblyGate` semaphore

---

🤖 2. AssemblyStateMachine

A pure state machine that:

- Holds the current state (`MutableStateFlow`)
- Performs all state transitions
- Emits logs `"Transition → XYZ"`
- Calls out to ports (I/O actions)
- Handles all timeouts

The state machine is deterministic — the same blueprint yields the same transitions unless timeouts occur.

🔌 3. AssemblyPorts

A dependency-injection container of callbacks.

The state machine does NOT know:

- How to send an order
- How to wait for confirmation
- How to wait for transport
- How assembly is validated
- How logs are forwarded
- How statuses are emitted
- How concurrency is enforced

All these details are injected through `AssemblyPorts`.

This makes the state machine pure and testable.

---

4. Frontend (React Dashboard)

A UI that listens to:

- `"state"` events
- `"status"` events
- `"log"` events

And builds:

- State history
- Status history
- Logs
- Progress UI
- Per-order detail cards

Communication uses **Server-Sent Events (SSE)** — simple, lightweight, always-open stream from backend → browser.

---

3. ⚙ Execution Flow (Step-by-Step)

Below is the full lifecycle of **ONE** order from creation to completion.

Step 1 — Client requests an order

```
POST /assembly/transport-order?demo=true
```

AssemblyService:

- Adds request to `queue`
- Responds with an `AssemblyTransportOrder` (orderId assigned)

Step 2 — Global Queue Processes It

The queue is consumed by:

```
scope.launch {
    for(req in queue) {
        runOne(...)
    }
}
```

This guarantees:

- Orders are processed one-by-one in FIFO order
- No overload beyond capacity 100
- Each order is created in the order requested

But *assembly itself* still respects the semaphore — two orders may be created simultaneously, but only one assembles at a time.

▯ Step 3 — runOne() is called

`runOne()` sets up the entire execution environment for a single order:

It creates:

- Per-order state flows
- Per-order flows for confirmation, arrival, validation
- Per-order coroutine scope (`orderScope`)
- The state machine instance
- A state collector that forwards transitions to SSE

Then it starts:

- `stateCollector` (listens to machine.state)
- `machineJob` (runs the state machine)

▯ Step 4 — StateMachine.run()

The state machine walks through **14 possible system states**:

```
CREATING_ORDER
ORDER_CREATED
SENDING_ORDER
RECEIVING_CONFIRMATION
EVALUATING_CONFIRMATION
ORDER_ACCEPTED / ORDER_DENIED / ORDER_TIMED_OUT
WAITING_FOR_TRANSPORT
ASSEMBLING
ASSEMBLY_COMPLETED / ASSEMBLY_INVALID / ASSEMBLY_TIMED_OUT
```

Every transition:

- Updates its internal `MutableStateFlow`
- Emits a log event: `"Transition → <STATE>"`
- Is forwarded to the frontend by the state collector

▯ Step 5 — The system waits for external events

The system suspends and waits for:

✔ Order confirmation

Via:

```
awaitOrderConfirmation()
```

Controlled by:

```
PUT /assemble/confirm-order?orderId=x&accepted=true
```

✔ Transport arrival

Via:

```
awaitTransportArrival()
```

Controlled by:

```
PUT /assembly/signal-transport-arrived?orderId=x
```

✔ Assembly validation

Via:

```
performAssemblyAndValidate()
```

Controlled by:

```
PUT /assembly/validate-assembly?orderId=x&valid=true
```

Each step has a timeout:

Step	Timeout
Confirmation	5s
Transport Arrival	300s
Validation	40s

If a timeout occurs, the state machine transitions to a terminal error state.

⛔ Step 6 — Assembly Semaphore

```
assemblyGate.acquire()
...
assemblyGate.release()
```

This ensures:

- Only **one** order is assembled at a time
- Others must wait in `WAITING_FOR_TRANSPORT` or before assembly

Even if you POST 100 orders at once, they will assemble sequentially.

✅ Step 7 — Completion & Cleanup

When the machine finishes:

- Final state is emitted
- `orderStates[orderId]` is updated
- Confirmation / arrival / validation flows are removed
- `orderScope` is cancelled (which also stops the collectors)

---

4. 📡 Event Streaming (SSE)

The backend sends **three types** of events:

1. `state`

Emitted whenever the state machine transitions:

```
{
  "kind": "state",
  "state": "ASSEMBLING",
  "orderId": "order-xyz",
  "ts": 1710000000000
}
```

2. `status`

Business status (ACCEPTED, COMPLETED, etc)

```
{
  "kind": "status",
  "message": "COMPLETED",
  "orderId": "order-xyz",
  "ts": 1710000000000
}
```

3. `log`

Human-readable debug logs

```
{
  "kind": "log",
  "message": "Transition → ASSEMBLY_COMPLETED",
  "orderId": "order-xyz",
  "ts": 1710000000000
}
```

The React frontend merges these into:

- Timeline of states
- Timeline of statuses
- Log list

---

5. 🖥️ Frontend Responsibilities

The React app:

- Connects once via an EventSource
- Listens to `state`, `status`, `log`
- Keeps a record per-order:

```
{
  order: AssemblyTransportOrder,
  lastState: AssemblySystemStates,
  lastStatus: OrderStatus,
  logs: [],
  stateHistory: [],
  statusHistory: []
}
```

It renders:

- Order card
- Live state badge
- Progress bar based on state
- Collapsible logs/history
- Status badge (ACCEPTED, COMPLETED, etc)

It does **not** infer states — it only displays what SSE sends, ensuring correctness.

---

6. 🧭 State Machine Flow Diagram

```
 CREATING_ORDER
       ↓
ORDER_CREATED
       ↓
SENDING_ORDER
       ↓
RECEIVING_CONFIRMATION
       ↓
EVALUATING_CONFIRMATION
       ├─ null → ORDER_TIMED_OUT → END
       ├─ false → ORDER_DENIED → END
       └─ true → ORDER_ACCEPTED
                        ↓
             WAITING_FOR_TRANSPORT
                        ↓ (or timeout → ORDER_TIMED_OUT)
                   ASSEMBLING
                        ↓ (validation timeout → ASSEMBLY_TIMED_OUT)
                        ↓ (invalid → ASSEMBLY_INVALID)
                        ↓ (valid → ASSEMBLY_COMPLETED)
```

---

7. ✅ Key Guarantees

Guarantee	Explanation
FIFO order creation	Queue enforces creation order
Single active assembly	Semaphore enforces mutual exclusion
Real-time updates	All state/log/status updates streamed via SSE
No state loss	State collector mirrors every transition
Per-order isolation	Each order has its own coroutine scope
Cleanup after finish	Flows removed, state stored, scope cancelled

---

8. 🧪 Demo Mode

When `demo=true`, the backend simulates:

- Auto-confirmation after 2s
- Auto-transport after 20s
- Assembly duration random 10–20 seconds
- Always VALID validation

This helps test UI and concurrency.

---

9. 🧠 Lifecycle Summary

For every order:

1. Add to queue
2. Create flows
3. Start collectors
4. Start state machine
5. Suspend on real events (confirmation, arrival, validation)
6. Acquire assembly lock
7. Assemble

8. Release lock
9. Emit final state
10. Cancel scope & cleanup

For the frontend:

- Always listening
- Automatically updates state, status, logs
- No polling needed

---

10. ✅ Conclusion

This architecture cleanly separates:

✔ Business logic (state machine)

✔ Concurrency and orchestration (AssemblyService + semaphore)

✔ I/O and external events (AssemblyPorts)

✔ Real-time UI updates (SSE)

✔ Visual representation (React Dashboard)