WorkFlowEngine (SPECS)

Workflow Engine Specification v1 (DB-Backed LifeCycle + Hooks + ACK + Monitor)

0) Purpose

Provide a **dumb, reliable, DB-backed workflow/state engine** that:

- Transitions an instance through states based on triggers (macro workflow)
- Emits outbound events for **transitions** and **hooks/work items** (micro orchestration)
- Tracks delivery/processing via **acknowledgements** and **re-sends** until completed
- Automatically triggers **timeout events** for stale instances
- Produces a timeline for reporting; runtime tables are mainly for app activity/status storage

Non-goal: implement business logic. Business meaning lives in the application and in JSON policies/routes, not in engine code.

---

1) Non-negotiable coding rules

1.1 SQL + DAL

- **SQL queries are first-class**: every DB operation must have a named query constant (in the relevant QRY_* class).
- **WorkFlowDAL is the only DB entry point** for the engine.
  - Engine/components do **not** call leaf DALs directly.
- Adding new DB capability:

1. Add query to the relevant QRY_* class
2. Extend relevant DAL(s)
3. Expose via WorkFlowDAL

1.2 Formatting + model style

- Keep model constructors **minimal**; prefer new X { A = a, B = b }.
- Keep method signatures/constructors **in one line** (no wrapping).
- Keep each SQL query string **on one line**.

---

2) Core contracts

2.1 Inputs

Trigger request (application → engine)

- env_code, def_name (or def_id/def_version resolved by engine)
- external_ref (primary correlation key owned by application)
- event_id / event_code (trigger)
- request_id (idempotency key; required for safe retries)
- actor (optional)
- payload (optional; opaque to engine)

Ack request (application → engine)

- ack_guid
- consumer_code (or consumer_id)
- outcome/status update
- optional: message/error, retry_at

2.2 Outputs (engine → application)

The engine exposes **two C# events**:

1. **Work event** (actionable): raised for both transition events and hook events
2. **Notice event** (informational): failures, retries, duplicates, warnings, etc.

2.3 Base event requirement (mandatory fields)

All actionable outbound events (transition + hook) derive from **one base event** and MUST include:

- external_ref ✅ (so app can correlate)
- ack_guid ✅ (so app can store + ack later)
- event info ✅ (what happened / what is requested)

Additional common fields (recommended):

- env_code, def_name/def_version
- instance_id (internal id)
- timestamp (occurred_at)

- payload (hook/transition specific)

---

3) ACK semantics (Delivered vs Processed)

ACK is tracked per consumer (or per configured consumers list).

3.1 Delivery (fast)

- Goal: consumer confirms it **received** the event.
- If Delivered is still **Pending for 30 seconds**:
  - **Monitor must re-raise the same event** (same ack_guid, same contents) until delivery is completed.

3.2 Processed (slow)

- Goal: consumer confirms it **completed processing**.
- Processing can take time (application internal work).
- If Processed not completed within **5 minutes**:
  - Monitor may re-raise/remind (less aggressive than delivery), still using same ack_guid.

Key rule: **Resends must be idempotent** because ack_guid and/or request_id makes repeated delivery safe.

---

4) Monitor (the heart of reliability + idempotency + automation)

Monitor has two major responsibilities:

4.1 ACK re-dispatch

- Frequently queries for:
  - delivery-pending ACKs older than 30s
  - processing-pending ACKs older than 5m
- Re-raises actionable events until the relevant ACK stage is complete.
- Re-raise uses **the original ack_guid** (never create a new one for the same dispatch).

4.2 Stale-instance timeout automation

- Frequently scans instances in states with timeouts.
- If instance remains in a state longer than configured duration:

  - Monitor triggers the configured **timeout_event** automatically (system-triggered).
  - Supports repeat timeout modes (e.g., reminders).

Monitor is the primary mechanism that:

- keeps delivery reliable,
- makes retries safe (idempotent resends),
- and advances time-driven workflows without app intervention.

---

5) End-to-end flow

5.1 Import time (JSON → DB)

1. IBlueprintImporter.ImportAsync(...) reads:
    - definition JSON (states, events, transitions, timeouts)
    - policies JSON (policies, routes, emitted app events)
2. Writes normalized rows to SQL:
    - environment, definition, def_version
    - state/event/transition tables
    - policy + route tables + relationships

5.2 Runtime trigger (application → engine)

1. App calls TriggerAsync(request)
2. Engine loads blueprint (cached) from IBlueprintManager
3. Engine ensures instance exists for (def_version, external_ref)
4. Engine applies transition:
    - validates allowed transition
    - writes lifecycle log/timeline
    - updates instance current_state
    - enforces request_id idempotency (duplicate/no-op becomes Notice)
5. Engine resolves policy/routes and emits hooks if applicable

5.3 Emission (engine → application)

For each outbound actionable event (transition/hook):

1. Create ACK record(s) and consumer tracking rows
2. Raise event to application including:
   - external_ref
   - ack_guid
   - event information (transition/hook payload)

5.4 Ack updates (application → engine)

- Application calls AckAsync(ack_guid, consumer, stage/outcome, ...)
- Engine persists ACK status
- Monitor uses ACK state to decide resend/remind

---

6) Component responsibilities (high level)

6.1 WorkFlowDAL

- Single DB gateway for engine.
- Consolidates all DALs.
- Enforces query-first discipline (QRY_* constants).

6.2 IBlueprintImporter

- Loads JSON and maps directly to SQL.
- Owns import/versioning rules (env/def/def_version and policy linkage).
- Critical for making DB the canonical runtime source.

6.3 IBlueprintManager

- Builds LifeCycleBlueprint from DB representation.
- Caches for runtime performance.
- Supports invalidation after import changes.

6.4 IStateMachine

- Ensures instance creation.
- Validates + applies transitions.

- Writes lifecycle/timeline and state updates.
- Enforces request_id idempotency at transition level.

6.5 IPolicyEnforcer

- Evaluates routes/policies for a given transition/state context.
- Emits hooks/work items and prepares outbound hook events + their ACKs.

6.6 IAckManager

- Records ACK updates from application.
- Provides "pending dispatch" queries for monitor.
- Separates Delivered vs Processed semantics.

6.7 IInstanceMonitor

- Periodic run loop:
  - ACK re-dispatch (30s delivery / 5m processed)
  - timeout scanning + automatic timeout triggers
- Emits Notices for retries/escalations when needed.

6.8 IWorkFlowEngine / ILifeCycleEngine

- Orchestrator API surface:
  - TriggerAsync, AckAsync
  - raises actionable events and notice events
- "Dumb": no business logic; executes blueprint + reliability loop.

---

7) Notices (engine → application)

Notices are informational reminders; application may log/alert but doesn't have to act.

Typical notice types:

- DuplicateRequest / IdempotentReplay
- NoOpAlreadyInState
- TransitionRejected (invalid event from state)
- DeliveryRetryRaised (ack delivery pending > 30s)

- ProcessingReminderRaised (processed pending > 5m)
- DispatchFailure / ConsumerFailure (with retry_at)
- TimeoutTriggered (system-triggered event)

---

8) Runtime tables philosophy

Runtime storage exists primarily for:

- timeline reporting (“what happened when”)
- allowing application to store its own activity status
  It should not add “smart behavior” beyond reporting/observability.
