International Journal of Emerging Trends in Computer Science and Information Technology
ISSN: 3050-9246 | https://doi.org/10.63282/3050-9246.IJETCSIT-V6I4P117
Eureka Vision Publication | Volume 6, Issue 4, 121-124, 2025

Original Article

Understanding Spring State Machine Using Proper Use Cases

Sasikanth Mamidi
Independent Researcher, USA.

Received On: 18/10/2025 ***Revised On: 10/11/2025*** ***Accepted On: 18/11/2025*** ***Published On: 25/11/2025***

Abstract - Spring State machine turns temporal business logic into explicit, testable models. This paper shows how to design and operate real workflows with hierarchical states, guards, and actions, using a fuel-retail case where reliability, safety, and latency is paramount. We present an architecture that isolates domain logic behind ports/adapters (EMV/acquirer, dispenser, receipt), persists context for recovery, and exposes rich observability through metrics, structured logs, and traces. The case study follows the full card-present journey, card read, authorization, pump enablement, fueling, EOT computation, and receipt, highlighting idempotence, compensations, and controlled concurrency. Performance measurements quantify transition latency and memory footprint under Redis and JPA persistence. We close with guidance on testing, model versioning, and safe evolution so teams can replace brittle conditional flows with diagrams that act as living documentation and an executable contract.

Keywords - Spring State Machine, State Machine, State Charts, Hierarchical States, Guards, Actions, Event-Driven Architecture, EMV Payments, Fueling Workflow, Observability.

1. Introduction

State machines provide a precise vocabulary to describe how software behaves over time, especially when many external actors exchange asynchronous signals. In retail fueling, payment terminals, pump dispensers, acquirer hosts, and receipt printers collaborate under strict timing, safety, and compliance constraints. Historically, teams have woven this collaboration with scattered flags, nested conditionals, and callbacks, making failure modes hard to reason about and audits painful. Spring State machine introduces a formal structure that elevates temporal behavior to the same status as data models: states describe phases of work, transitions specify when and how movement occurs, and actions localize side effects. The result is both executable orchestration and self-documenting intent. When paired with Spring Boot, the framework integrates smoothly with REST, messaging, metrics, and persistence, allowing domain logic to remain pure while adapters bridge the outside world.

This paper explains the framework through use cases rather than a catalog of APIs. We begin by positioning state machines against alternative coordination techniques such as imperative controllers and message-driven sagas, then motivate when to choose explicit orchestration over implicit choreography. We outline modeling practices that keep machines readable domain-centric naming, clear separation between guards and actions, minimal extended state, and hierarchical decomposition. The heart of the paper is a fuel transaction case study that follows the customer journey. card insertion, host approval, pump authorization, fueling, computation of EOT details, and receipt issuance. We close with performance measurements and operational guidance, demonstrating that a well-designed machine can meet strict latency targets and improve reliability, while remaining evolvable as payment rules, device protocols, and product features change.

2. Problem Statement

Fuel retail transactions require coordinating several unreliable systems. Card readers (often EMV), acquirer gateways, dispenser controllers, receipt services, and store back-office systems. Each component introduces latency, partial failure, and non-determinism, yet the customer expects a seamless, fast experience at the forecourt. Common implementations rely on nested control structures and implicit state encoded in Booleans and callbacks. This approach obscures business rules, makes race conditions likely (e.g., card removal during authorization, nozzle lift before approval), and complicates recovery. Operators and auditors face a visibility gap: they can see logs but not the transaction's exact phase, complicating troubleshooting and SLA enforcement. As the number of pumps and promotions grows, so does the combinatorial space of corner cases, further stressing ad-hoc controllers.

The orchestration layer therefore needs to externalize state, render transitions explicit, and provide hooks for cross-cutting concerns such as metrics, tracing, idempotence, and compensations. It must support long-running interactions with persisted context, hierarchical decomposition to keep the model manageable, and concurrency to coordinate devices without deadlocks. It must also protect invariant "no dispenser activation before host approval," "receipt content reflects persisted EOT details," "duplicate events never produce duplicate side effects", while integrating with payment and device protocols that vary by market. Finally, it should evolve safely. Models need versioning, in-flight instances need migration strategies, and teams need a way to test changes against realistic event streams before exposing them to customers.

3. Objectives

The primary objective is to demonstrate how Spring State machine can encode the full lifecycle of a card-present fueling transaction in a way that is correct by construction, testable in isolation, and observable in production. By modeling with explicit states and guardable transitions, we clarify domain invariants, such as the prerequisites for pump authorization and the rules for computing totals and allow the runtime to enforce them mechanically. A second objective is to show how hierarchical states and concurrent regions reduce cognitive load without losing precision, letting payment and device flows proceed in sequence or in parallel as policy and hardware permit. We aim to preserve a crisp domain core while adapters handle the messy outer edge: EMV exchanges, acquirer requests, dispenser protocols, and receipt formatting.

A further objective is architectural portability. We articulate a set of patterns and adapters for external systems, pure domain actions, deterministic guards, and idempotent side effects that let teams replace payment gateways, dispenser protocols, or printers without remapping the core machine. We also target operational excellence, consistent metrics for transition counts and durations, structured logs that carry correlation IDs, and traces that narrate each step. Finally, we propose governance for safe change, versioned state models stored alongside code, data migration for in-flight instances when invariants evolve, and model-level tests that exercise event sequences rather than lines of code. By the end, readers should be able to build a production grade machine that is easy to reason about, simple to operate, and resilient to failure.

4. Literature Review

Finite State Machines (FSMs) and Harel state charts underpin many safety-critical systems because they make temporal behavior explicit. State charts extend FSMs with hierarchy, concurrency, and history semantics, reducing diagram complexity while maintaining rigor. In software, the GoF State pattern popularized encapsulating state-dependent behavior, but hand-rolled implementations often devolve into scattered flags. Workflow engines and BPM suites addressed long-running processes, yet their heavyweight modeling and execution semantics are overkill for many transactional domains. Spring State machine occupies an attractive middle ground: it borrows the expressive power of state charts, composes naturally with Spring Boot, and keeps execution lightweight enough for high-throughput services.

Empirical studies of payments orchestration highlight failure hotspots, authorization timeouts, duplicate submissions, and reconciliation drift, where explicit state models improve outcomes by centralizing guard logic and making retries transparent. In retail fueling and forecourt control, device controllers emit asynchronous signals (nozzle lift, pump ready, fuel start/stop) that must be synchronized with host approvals and safety interlocks. Event-driven microservices literature often advocates choreography, but for safety-critical sequences with strict ordering and compensations, orchestration via a state machine provides crisper guarantees. Prior art in open-source libraries (e.g., SCXML engines, Akka FSM, XState) and industrial controllers validates the approach. Spring State machine distinguishes itself with idiomatic Spring integration, multiple persistence options, and support for hierarchical modeling. This body of work informs our modeling choices, test strategies, and performance expectations.

5. System Architecture

The proposed system centers on a Spring Boot service that owns the fueling transaction lifecycle via Spring State machine, flanked by adapters to external systems. A thin HTTP/JSON command API accepts card events from the payment terminal, host responses from the acquirer, device events from the dispenser controller and maps them to domain events published into the machine. Outbound actions call adapter ports implemented by infrastructure components. a payments client for EMV host calls, a dispenser client for pump authorization and status, and a receipt client for formatting and printing. The machine persists context so that restarts and retries do not lose progress. Persistence can be in-memory for tests, Redis for speed, or JPA for durability. Structured logging and tracing propagate a transaction ID from first event to receipt, allowing precise observability.

Concurrency is handled with one machine instance per transaction, while multi-pump coordination uses separate instances correlated by store and dispenser identifiers. Guards ensure exclusivity. no pump is authorized twice and idempotence replaying an approval event does not duplicate side effects. Hierarchical states keep the diagram manageable. a superstate “InProgress” contains “Payment Flow” and “Fueling Flow” as concurrent regions where hardware permits, or as sequential substates where policy requires payment before dispenser activation. Entry and exit actions initialize timers, schedule retries, and collect metrics. Interceptors capture transitions for auditing, and error subgraphs route failures to compensations (reversals, deauthorizations, customer messaging). The architecture is intentionally modular so that replacing a gateway or adding promotions only touches adapters and, when necessary, localized model branches.

6. Implementation Strategy

Implementation begins with modeling. Enumerate the domain states Idle, Card Inserted, Awaiting Host Approval, Approved, Pump Authorized, Fueling, Finalizing, Completed, Failure and sketch hierarchical groupings. Identify events that trigger transitions: CARD_READ, HOST_APPROVED, HOST_DECLINED, PUMP_READY, FUEL_START, FUEL_STOP, EOT_TOTALS, RECEIPT_PRINTED, and TIMEOUT variants. Express invariants as guards, authorize the pump only if the approval code is present and the dispenser is assigned; compute totals only after fuel stop and dispenser latch closed. Embed idempotence by reading and updating a single EOT context, running the same action twice yields the same result. Keep extended state minimal and domain-centric approval code, assigned dispenser, product, unit price, quantities, taxes, and

timestamps, so that diagrams stay readable and tests remain focused.

Next, wire adapters and observability. Define interfaces for payments, dispenser, and receipt services. Provide test doubles for integration tests. Use interceptors to publish metrics, transition counts, guard rejections, retries and traces with the transaction ID as the span root. Configure persistence (JPA or Redis) and enable state machine regions if device and payment can advance independently. Write model-level tests that inject events and assert resulting states, guards, and actions without spinning up web servers or external systems. For failure paths, script timeouts, host declines, printer errors, and dispenser faults. verify compensations, reversals posted, pump deauthorized, and customer messaging triggered. Finally, package the machine as a library with versioned models so applications can evolve the diagram safely, extending or replacing branches under feature flags and migrating in-flight instances as needed.

7. Case Study & Performance Evaluation

Consider a single fueling transaction. The customer inserts a card, the terminal emits CARD_READ with tokenized PAN and requested pre-authorization amount. The machine leaves Idle for Card Inserted, triggers an action to request host authorization, and moves to Awaiting Host Approval. On HOST_APPROVED, the approval code and limit are persisted, and the machine transitions to Approved, where it calls the dispenser client to authorize the assigned pump and product. When the dispenser replies PUMP_READY, the machine enters Pump Authorized and awaits FUEL_START. During Fueling, device events stream in. the machine records quantity and price calculations without producing customer-visible side effects. On FUEL_STOP and nozzle replacement, the machine transitions to Finalizing, computes End-of-Transaction (EOT) details from the persisted context, totals, taxes, discounts, card token, and EMV tags and triggers receipt formatting. After RECEIPT_PRINTED, the machine enters Completed and emits a reconciliation event to back-office systems. If any step fails, error states launch compensations, reversals to the acquirer, deauthorization for the pump, or a reprint flow for the receipt.

We evaluated latency and resilience under production-like load. With pre-warmed machine factories and Redis persistence, median transition latency remained sub-millisecond for internal transitions and under a few tens of milliseconds when actions invoked external adapters. End-to-end from CARD_READ to Completed, the P50 remained single-digit seconds, with P95 bounded primarily by acquirer response times. Memory overhead per live machine instance averaged a few kilobytes of context plus framework structures, supporting thousands of concurrent transactions per node. Under fault injection gateway timeouts, duplicated device events, transient printer failures guards and idempotent actions prevented duplicates and ensured that retries remained safe. Operators used metrics and traces to spot bottlenecks, reveal slow approvals, and confirm that compensations triggered exactly once, validating the model's reliability.

8. Results

The implementation delivered measurable improvements across reliability, operability, and evolvability. By relocating business rules into guards and idempotent actions, we eliminated classes of race conditions and reduced defect density in areas previously dominated by conditional sprawl. Incident investigations benefited from model-aligned observability. operators could answer "where is my transaction?" by inspecting live counts per state and opening traces that narrate the precise sequence of events. Because every side effect hangs off a named transition, post-mortems could map directly to the diagram, aligning engineering, QA, and compliance conversations. The explicit error subgraph constrained compensations reversals, deauthorizations, and reprints, so that failure handling was deliberate rather than ad-hoc.

From a product perspective, the state model simplified introducing features that once threatened regressions. Adding a pre-authorization top-up, for example, became a localized subgraph gated by clear guards and compensations. The same was true for loyalty accrual and targeted promotions, actions emitted domain events without polluting the core machine, while guards ensured they could never alter safety or payment ordering. Audit and compliance gained confidence because the machine's structure mapped directly to payment rules and forecourt interlocks. Receipts were generated from a single source of truth (the EOT context) rather than reconstructed heuristics. Developer experience improved as tests focused on event sequences and invariant enforcement instead of incidental wiring, leading to faster cycle times and more predictable releases.

9. Conclusion & Future Work

Spring State machine demonstrates that explicit orchestration can coexist with cloud-native development without sacrificing agility. For fueling transactions, the framework offered a disciplined way to encode invariants, coordinate unreliable devices, and surface precise operational signals. The case study showed that a carefully designed model renders the happy path simple and the failure paths deliberate, avoiding the silent complexity that accumulates in ad-hoc controllers. Equally important, the diagram becomes a contract among engineering, product, and compliance. a shared artifact that expresses what is allowed, when, and under what conditions then enforces it at runtime.

Future work spans engineering, operations, and product. On the engineering side, we intend to formalize properties "authorization precedes dispensing," "receipts reflect persisted EOT" and verify them with model-checking and property-based tests. We will explore fully reactive execution to simplify back-pressure and timeouts, and extend persistence to support long-lived reservations and offline modes when connectivity degrades. Operationally, we will expand metrics to include customer-centric SLOs, add

automated anomaly detection over transition streams, and harden migration tooling for versioned models and in-flight upgrades. Product efforts will integrate digital receipts and loyalty accrual without bloating the core machine by housing these concerns in sidecar actions and event listeners. Together, these directions deepen the model's guarantees while keeping the implementation straightforward, auditable, and adaptable to evolving payment and device landscapes.

References

[1] Spring Team, "Spring Statemachine — Reference Documentation," *Spring.io*, [Online]. Available: https://docs.spring.io/spring-statemachine/reference/

[2] Vegesna, R. V. (2025). Developing software for automated firmware updates in fuel controllers. Journal of Artificial Intelligence & Cloud Computing, 1–3. https://doi.org/10.47363/jaicc/2025(4)e263

[3] Bazzi Abir, Ma Di (2023) MT-SOTA: A Merkle-Tree-Based Approach for Secure Software Updates over the Air in Automotive Systems. Applied Sciences 13. 9397. 10.3390/ app13169397

[4] Vegesna, R. V. (2024). Designing an archival system for Long-Term Fuel System data analysis. International Scientific Journal of Engineering and Management, 03(09), 1–3. https://doi.org/10.55041/isjem02155

[5] Tammaa, Ahmed. (2022). MongoDB Case Study on Forbes. 10.13140/RG.2.2.32766.46408.

[6] antkorwin.com, "Spring Statemachine," *antkorwin.com*, [Online]. Available: https://antkorwin.com/statemachine/statemachine.html

[7] oohm.io, "Building a Non-Blocking State Machine in Spring Boot," *oohm.io*, [Online]. Available: https://oohm.io/blog/Building-a-Non-Blocking-State-Machine-in-Spring-Boot/

[8] Kanji, R. K. (2022). A Unified Data Warehouse Architecture for Multi-Source Forest Inventory Integration and Automated Remote Sensing Analysis. *Sarcouncil Journal of Engineering and Computer Sciences*, *1*, 10-16.