Implementation Roadmap: Application Service Framework

Version: 1.1
Date: 2025-07-23

1. Introduction

This document provides a detailed, implementation-focused roadmap for the Application Service framework. It breaks down each phase into specific technical tasks, architectural decisions, and technology choices, offering a clear path from foundational infrastructure to a fully functional, user-facing application.

Phase 1: Core Infrastructure & Data Ingestion (Months 1-3)

Objective: Establish a robust, scalable microservices foundation and a versatile data ingestion pipeline.

1.1. Components & Implementation Details:

- **Datasource Service (Java, Spring Boot):**
  - **Core Logic:** Implement a DataSourceAdapter interface with concrete strategies for each data source type.
    - JdbcAdapter: Use spring-boot-starter-data-jdbc and JdbcTemplate for direct SQL execution. A configuration file will map tables and columns to predicate names (e.g., users.name -> hasName). The adapter will dynamically query table metadata to handle schema evolution.
    - RestApiAdapter: Use spring-webflux's non-blocking WebClient. It will support paginated APIs by following next links in response headers or bodies.
    - FileAdapter: Use the Jackson library for JSON/XML parsing. It will watch a designated directory for new or updated files.
  - **Transformation:** The core transformation logic will convert source entities into SPO triples. For a database row (PK=123, table='Product', column='Name', value='Laptop'), the output will be a message: ("product:123", "hasName", "Laptop", "source:db1"). The subject URI is a composite of the entity type and its primary key.
  - **Synchronization:** Implement a polling mechanism using @Scheduled annotations in Spring for sources without push notifications. For event-driven sources, it will expose a webhook endpoint to receive update events. Provenance is maintained by adding a context string (e.g., the source application's name) to each triple.
- **Augmentation Service (Java, Spring Cloud Stream):**
  - **Message Bus:** Use Apache Kafka as the backbone. Define clear, versioned

Avro schemas for all message types to ensure compatibility.
    - **Topics:**
        - datasource-raw-triples-v1: For raw data from the Datasource Service.
        - aggregation-reference-model-v1: For typed and identified data.
        - alignment-graph-model-v1: For semantically enriched data.
        - activation-dci-model-v1: For executable use cases.
    - **Orchestration & Saga Pattern:** Implement the Saga pattern using a state machine. For a multi-step process like "Ingest and Align," the service listens for a RAW_TRIPLE_INGESTED event, triggers the Aggregation Service, then listens for an AGGREGATION_COMPLETE event to trigger the Alignment Service. State transitions and compensating actions (e.g., deleting partially processed data on failure) are logged to a dedicated Kafka topic (saga-log-v1).
    - **Resiliency:** Use Spring Retry for transient failures and a dead-letter queue (DLQ) pattern for messages that repeatedly fail processing.
- **Registry Service (Helper Service - Java, Spring Boot, Neo4j):**
    - **Database:** Use a Neo4j graph database.
    - **Schema:** Nodes will have the label :Resource and a uri property (which is indexed). Relationships will represent the predicates.
    - **API:** A RESTful API built with Spring Boot and spring-data-neo4j.
        - POST /v1/graph/statements: A batch endpoint that accepts a list of triples and executes a single, optimized Cypher UNWIND ... MERGE query for high-performance writes.
        - GET /v1/graph/resource?uri={uri}: Retrieves a resource and its immediate relationships.
    - **Provenance:** Store provenance data (e.g., sourceApplication, ingestionTimestamp) as properties on the nodes and relationships.

Phase 2: Semantic Core & Knowledge Representation (Months 4-7)

Objective: Transform raw data into an interconnected, semantically rich knowledge graph.

2.1. Components & Implementation Details:

- **Aggregation Service (Java, Spring AI, Python):**
    - **Core Logic:** Consumes from the datasource-raw-triples-v1 topic.
    - **ID & Embedding Generation:** For each new URI, generate a unique ID and an embedding vector. This can be a separate Python service called via RPC, using models like Sentence-BERT from the Hugging Face library to create meaningful embeddings. The mapping of URI to ID and embedding is cached

in Redis.
    - **Type/State Inference:** Use in-memory Caffeine caches for high-speed aggregation.
        - Map<String, Set<String>> subjectToPredicates: This map tracks all attributes for a given subject.
        - A background job periodically analyzes this map. Subjects with a high Jaccard similarity in their predicate sets are grouped into an inferred Type.
    - **FCA (Formal Concept Analysis):** Use the fcalib Java library. Create a formal context where "objects" are the subject URIs and "attributes" are their predicates. The resulting concept lattice directly forms the is-a type hierarchy.
    - **Output:** Produces Statement<ID, ID, ID, ID> messages to the aggregation-reference-model-v1 topic.
- **Alignment Service (Java, RDF4J):**
    - **Core Logic:** Consumes from the aggregation-reference-model-v1 topic.
    - **Ontology Matching:** Use the RDF4J framework's MemoryStore for in-memory graph operations. Load the Reference Model and pre-defined upper ontologies (e.g., Schema.org, custom domain ontologies in OWL format). Use the SPARQL engine with SHACL rules to find and materialize equivalences (owl:sameAs, rdfs:subClassOf).
    - **Link Completion:** Implement this with SPARQL CONSTRUCT queries. For example, a query can find paths like (A)-[:hasRole]->(B) and (B)-[:partOf]->(C) to infer a new link (A)-[:contributesTo]->(C).
    - **Output:** Produces enriched Statement<Context, Subject, Predicate, Object> messages to the alignment-graph-model-v1 topic.
- **Naming Service (Helper Service - Java, Apache Jena):**
    - **Core Logic:** A dedicated service that manages ontologies.
    - **Storage:** Use Apache Jena with a TDB2 persistent backend.
    - **API:** Expose a full SPARQL 1.1 endpoint using Jena Fuseki. This allows other services to query the ontologies directly. It will also have custom REST endpoints like POST /v1/align/concepts which takes two sets of concepts and returns a mapping of potential matches with confidence scores.

Phase 3: Activation & Use Case Enablement (Months 8-10)

Objective: Infer and enable the execution of business processes and use cases from the knowledge graph.

3.1. Components & Implementation Details:

- **Activation Service (Java, Spring Boot):**
    - **Core Logic:** Consumes from the alignment-graph-model-v1 topic.

    - **DCI (Data, Context, Interaction):** Implement the DCI pattern.
        - **Context Inference:** Use graph traversal algorithms (e.g., Depth First Search) or Cypher queries on the Registry to find recurring patterns that represent potential use cases. For example, a pattern of (Order)-[contains]->(Product)<-[trackedIn]-(Inventory) infers a ReplenishStock Context.
        - **Role & Actor:** Roles are the types of nodes in the pattern (e.g., Product, Inventory). Actors are specific instances (e.g., product:123).
        - **Interaction:** An Interaction is an instantiated Context. It's a stateful object that tracks the assigned Actors and the progress of the use case.
    - **Dataflow & Rules:** Use a rules engine like Drools to define the business logic. A rule might be: WHEN Inventory.level < Inventory.threshold THEN CREATE ReplenishStock.Interaction. The actual data transformations between actors can be defined using XSLT or implemented as simple Java methods.
    - **Output:** Produces Statement<Context, Interaction, Role, Actor> to the activation-dci-model-v1 topic.
- **Index Service (Helper Service - Python, Vector DB):**
    - **Core Logic:** A service for similarity-based retrieval.
    - **Database:** Use a dedicated vector database like Milvus or Pinecone.
    - **API:**
        - POST /v1/index/resources: Adds a resource's embedding to the index.
        - POST /v1/search/similar: Takes a vector and context filters (e.g., "find products similar to this one") and returns a list of matching resource URIs. This is used to find suitable Actors for a Role.

Phase 4: API & User Interface (Months 11-12)

Objective: Expose the framework's capabilities through a developer-friendly API and an intuitive user interface.

4.1. Components & Implementation Details:

- **Producer Service (API/Frontend - Java/Spring Boot, React):**
    - **Backend API:** A Spring Boot application that provides the public-facing interface.
    - **REST API:**
        - GET /v1/contexts: Lists available use cases.
        - POST /v1/interactions: Creates a new instance of a use case.
        - GET /v1/interactions/{id}: Retrieves the state of a specific transaction.
        - POST /v1/interactions/{id}/roles/{roleName}/assign: Assigns an actor to a role.

- **Hypermedia (HATEOAS):** Use spring-boot-starter-hateoas. Each response will contain _links that guide the client. An Interaction response will have links like assign-actor or complete-step.
- **Frontend (React):**
  - A Single-Page Application (SPA) built with React and TypeScript.
  - Use a component library like Material-UI or Ant Design for a consistent look and feel.
  - Implement a generic form renderer that builds input forms dynamically based on the JSON schema of the Roles provided by the API.
  - Use WebSockets to connect to the Augmentation Service (through an API Gateway) to receive real-time updates on the status of Interactions.
- **Authentication:** Implement OAuth 2.0 with an identity provider like Keycloak or Auth0. The API Gateway will enforce authentication and authorization policies.