Implementation Roadmap: Application Service Framework

Version: 3.0
Date: July 29, 2025

1. Introduction

This document provides a definitive, implementation-focused roadmap for the Application Service framework. It moves beyond architectural overviews to specify the granular, technical details required to build a fully reactive, functional, and behavior-driven system. We will explore the practical application of a sophisticated technology stack, emphasizing a shift towards a pure Semantic Web foundation and deterministic, structural embeddings over probabilistic LLM-based ones.

This version provides exhaustive detail on:

- **The Reference Model**: A deep dive into ID/IDOccurrence, prime number semantics, and its implementation as a foundational property graph on a Semantic Web store.
- **FCA-based Embeddings**: A novel approach to creating deterministic embeddings from the products of prime numbers in Formal Concept Analysis (FCA) contexts, replacing the need for external LLM embeddings for structural similarity.
- **The Graph Model**: A set-oriented approach to knowledge representation, with functional interfaces for inference over Kinds.
- **The Activation Model**: A practical guide to implementing DCI, DDD, and the Actor-Role pattern for dynamic, message-driven use case execution.
- **Core Technologies**: A switch to Apache Jena/RDF4J as the primary data store, and detailed implementation patterns for JCA, JAF, and dynamic ContentType handling.
- **Multidimensional Analytics**: A formal model for OLAP-style querying directly on the knowledge graph.

Phase 1: Semantic Core & Reactive Data Ingestion

Objective

To establish a robust, scalable, and fully reactive microservices foundation built upon a Semantic Web data store, with a versatile, non-blocking data ingestion pipeline.

1.1. Core Technology Shift: Semantic Web Store

In this iteration, we replace Neo4j with a dedicated Semantic Web stack.

- **Primary Store**: **Apache Jena** with a **TDB2** persistent backend.
- **Reasoning**: This choice provides native support for RDF, RDFS, OWL, and SPARQL 1.1, which is more aligned with the framework's goals of semantic inference and ontology alignment than a labeled property graph. The Registry Service will now expose a full SPARQL endpoint via **Jena Fuseki**. All services will interact with the core graph data via SPARQL queries.

1.2. The Reference Model: A Deep Dive

This is the foundational layer, representing the grammar of the system. It's where raw strings are formalized into identifiable and relatable concepts.

1.2.1. Entities: ID and IDOccurrence

- **ID**: Represents the canonical, context-free "idea" of a resource. It is immutable.
  - **primeID (long)**: A unique prime number assigned upon first encounter. This is the core identifier. A centralized "Prime Number Service" (e.g., using Redis INCR on a pre-computed list) will dispense these to guarantee uniqueness.
  - **urn (String)**: The canonical URN for the resource, e.g., urn:appservice:id:937.
  - **did (String)**: The W3C Decentralized Identifier, e.g., did:key:z6Mkp..., providing a global, verifiable identity.
- **IDOccurrence**: Represents an ID appearing in a specific role within a specific context (a statement). It is the "instance" of an idea in action.
  - **occurringId (ID)**: A reference to the canonical ID.
  - **context (IDOccurrence)**: A reference to the statement (IDOccurrence) in which this entity is participating. This creates a linked data structure.

1.2.2. Statement Types: Data vs. Schema

The model supports two parallel universes of statements, allowing it to describe both the world and the rules governing it.

- **Data Statements**: (Interaction, Actor, Transform) - These represent the actual "moves in the game." They are concrete happenings.
  - Example: (interaction:123, actor:user_Alice, transform:SetPaymentMethod)
- **Schema Statements**: (Context, Role, Dataflow) - These define the "rules of the game." They are abstract definitions.
  - Example: (context:Purchase, role:Buyer, dataflow:StandardCheckout)

1.2.3. Implementation as a Property Graph on RDF

While conceptually distinct, all three models (Reference, Graph, Activation) can be implemented on a single Jena TDB2 store. We use RDF's reification vocabulary (rdf:Statement, rdf:subject, rdf:predicate, rdf:object) to create a property graph

structure.

- A single resource (e.g., "Alice") is a URI: urn:appservice:id:937.
- Labels are applied using rdf:type: <urn:appservice:id:937> rdf:type :Resource .
- Properties are standard triples: <urn:appservice:id:937> :hasName "Alice" .
- The IDOccurrence is represented by the reified statement itself. A statement like (Alice, worksFor, Google) is stored as:
  @prefix : <urn:appservice:ont#> .
  @prefix id: <urn:appservice:id#> .

  :stmt_456 a rdf:Statement ;
  rdf:subject id:937 ; # Alice
  rdf:predicate :worksFor ;
  rdf:object id:101 . # Google

  The URI :stmt_456 is the identifier for this specific IDOccurrence of Alice.

1.3. FCA-based Embeddings: A Deterministic Approach

We will replace LLM-based embeddings with deterministic, structural embeddings derived from FCA contexts and prime number products. This provides explainable similarity based on shared roles and relationships.

- **Contextual Prime Product Embedding (CPPE)**: For any IDOccurrence (i.e., a resource in a specific statement), we can calculate an embedding based on its relational context.
  1. **Define FCA Contexts**: For a given relation (predicate), we can form an FCA context. Example: For the predicate :worksFor:
     - **Objects (G)**: The set of all subjects of :worksFor statements (e.g., {id:Alice, id:Bob}).
     - **Attributes (M)**: The set of all objects of :worksFor statements (e.g., {id:Google, id:StartupX}).
  2. Calculate Prime Product: The CPPE for id:Google within the :worksFor context is the product of the primeIDs of all employees who work there. CPPE(Google, worksFor) = primeID(Alice) * primeID(Bob) * ...
- **Similarity Calculation & Inference**:
  - **Similarity**: The similarity between two entities in the same context is the **Greatest Common Divisor (GCD)** of their CPPEs. GCD(CPPE(Google), CPPE(StartupX)) reveals the primeID product of their shared employees, giving a measure of personnel overlap.
  - **Relational Inference**: We can infer complex relationships. Consider the goal

of finding an "uncle".

1. Calculate the CPPE for "Person A" in the :brotherOf context (the product of their siblings' primes).
2. Calculate the CPPE for "Person B" in the :fatherOf context (the product of their children's primes).
3. If GCD(CPPE_brotherOf(A), CPPE_fatherOf(B)) > 1, it means A is the brother of B's father. The system can then materialize a new triple: (A, :uncleOf, ChildOfB). This inference is stored and queryable.

Phase 2: The Semantic Graph Model

Objective

To transform the formal Reference Model into a semantically rich Graph Model based on set theory, inferring types (Kinds) and order.

2.1. The Set-Oriented Graph Model

The model reifies statements into higher-order concepts called Kinds. Kinds are sets of IDOccurrences that share common structural properties.

- **Kind Aggregation**: Kinds are formed by grouping entities with common attributes. A SubjectKind like :Customer is formed by grouping all Subjects that interact with a similar set of (Predicate, Object) pairs (e.g., they all perform :purchase on :Products).
- Reification: A Kind is a subclass of its base type. A SubjectKind is-a Subject. In RDF, this is a simple rdfs:subClassOf relationship.
  <:Customer> rdfs:subClassOf <:SubjectKind> .
  <:SubjectKind> rdfs:subClassOf <:Subject> .

2.2. Functional Interfaces for Inference

The core logic for querying this model is expressed through functional interfaces, implemented with SPARQL.

- **Function<URI, Set<URI>> getPredicateKindsForSubjectKind**:
  - "Given the :Customer type, what types of actions can they perform?"
  - **Implementation**: A SPARQL query that finds all PredicateKinds connected to the given SubjectKind in schema statements.
- **Function<URI, Tuple<Set<URI>, Set<URI>>> getValidSubjectObjectKindsForPredicateKind**:
  - "Given the :Purchase action type, what are the valid types of subjects and objects?"
  - **Implementation**: A SPARQL query that returns the SubjectKind and

ObjectKind from schema statements where the PredicateKind is :Purchase.

2.3. Order Inference and Materialization

Order is crucial for understanding processes. It's inferred from hierarchies in type (schema) and state (data).

- **Order from Type/Schema Hierarchies**: A more general type is considered to be "before" a more specific one. The Aggregation Service infers type hierarchies via FCA. This is materialized as rdfs:subClassOf.
  - Example: FCA reveals all attributes of :Person are a subset of :Employee.
  - Materialization: <:Employee> rdfs:subClassOf <:Person> . This implies Person is a prerequisite for Employee.
- **Order from State/Data Hierarchies**: States in a process follow a defined sequence.
  - Example: The system observes order events transitioning through states: Placed -> Paid -> Shipped.
  - Materialization: It creates explicit ordering relationships using a custom ontology:
    <:Paid> :precededBy <:Placed> .
    <:Shipped> :precededBy <:Paid> .
    This allows for path-based SPARQL queries to determine process prerequisites and validate state transitions.

Phase 3: The Dynamic Activation Model

Objective

To infer and enable the execution of business processes using DCI, DDD, and a Dynamic Object Model.

3.1. Activation Model Entities

- **DOM (Dynamic Object Model)**: An Actor's data is not a static class but a flexible Instance whose capabilities can change.
  - **Instance**: An IDOccurrence with a map of attributes: Map<URI, Instance>.
  - **Class**: An Instance that defines the structure (fields) for other Instances.
- **DCI (Data, Context, Interaction)**:
  - **Context**: Defines a use case schema (roles).
  - **Role**: A Class that defines behavior. It has previous, current, and next Dataflow maps, defining the valid state transitions.
  - **Interaction**: An instance of a Context with concrete Actors.
- **Actor-Role Pattern**:

    - **Actor**: An Instance playing a Role. Its state is defined by its available Transforms.
        - state: Map<Context, ActorState(previous, current, next)>
    - **Transform**: A declarative message instructing an Actor on how to mutate its state.

3.2. Dataflow via Transform Messages

An Interaction's dataflow is a message-driven, distributed state machine.

1. **Schema Definition**: A Dataflow in a Role definition is a sequence of Transform definitions. A Transform definition specifies an operation (SET_FIELD, MUTATE_FIELD) and its inputs/outputs.
2. **Interaction Orchestration**: The Interaction orchestrator (a stateful service) reads the Dataflow schema. To advance the process, it creates a concrete Transform message from the definition, populating it with the actual Actor DIDs.
3. **Message Passing**: This Transform message is published to a Kafka topic.
   ```
   // Example Transform Message
   {
    "transformId": "txf_987",
    "interactionId": "interaction_123",
    "targetActorId": "did:key:z6...", // Actor: Alice
    "operation": "SET_FIELD",
    "payload": {
     "fieldName": "shippingAddress",
     "value": { "street": "123 Main St", "city": "Anytown" }
    }
   }
   ```
4. **Actor State Mutation**: The target Actor (a stateful microservice instance) consumes this message. It applies the operation to its internal DOM Instance data, changing its state. It then emits an ActorStateChanged event, which the Interaction orchestrator consumes to trigger the next Transform.

3.3. JAF & Dynamic ContentType Handlers

Instead of hard-coding a Spring bean for each ContentType, we will use a generic, data-driven approach.

- **GenericContentTypeDataHandler**: A single Spring bean that is a ApplicationContextAware factory.
- **Dynamic Registration**: When the **Alignment Service** infers a new ContentType (e.g., :sell-able), it materializes not just the type but also its associated Dataflow

definition (a sequence of Transform schemas) in the Jena store.
- **Runtime Lookup**: When the Activation Service needs to process a Verb on a ContentType, it asks the GenericContentTypeDataHandler. The handler queries the Jena store for the Dataflow associated with that ContentType and Verb, and then uses that definition to orchestrate the Interaction. This makes the system's behaviors entirely configurable at runtime without redeployment.

Phase 4: Multidimensional Features & Analytics

Objective

To implement powerful, OLAP-style analytics directly on the live, operational knowledge graph.

4.1. The Dimensional Service & Data Model

A dedicated helper service, the DimensionalService, manages all dimensional information.

- **Storage - Nested Context Statements**: We encode dimensional data using RDF reification to create explicit, queryable paths. A single sale event is encoded as a series of nested statements.
  1. **The "What"**: stmt1 a rdf:Statement; rdf:subject :order_456; rdf:predicate :hasValue; rdf:object "118.00"^^xsd:decimal .
  2. **The "When"**: stmt2 a rdf:Statement; rdf:subject stmt1; rdf:predicate :has_dimension; rdf:object :dim_Time .
  3. **The "Which"**: stmt3 a rdf:Statement; rdf:subject stmt2; rdf:predicate :has_dimension; rdf:object :dim_Product .
  4. **The "Where"**: stmt4 a rdf:Statement; rdf:subject stmt3; rdf:predicate :has_dimension; rdf:object :dim_Region .
- **Functional Retrieval**: This nested structure creates a queryable path: (stmt4)->(stmt3)->(stmt2)->(stmt1). An OLAP-style query like "Show me all sales for the Pro-Lite shoe in Boston" becomes a SPARQL SELECT query that finds all paths matching this pattern.

4.2. Alignment Feature Materialization

The **Alignment Service** is responsible for creating a unified understanding of measurement and order.

- Ontology Matching: It finds that user.creation_date from one system and customer.signup_timestamp from another are semantically equivalent. It materializes this link:
  <:creation_date> owl:sameAs <:signup_timestamp> .

- **Dimensional Alignment**: It finds price_eur: 100 and price_usd: 118 linked to the same product. It uses the DimensionalService to confirm they are comparable along the Currency dimension and materializes the aligned value in a canonical unit (e.g., USD) in the Reference Model.

Appendix A: Business Intelligence & Analytics

Each organization's ApplicationService instance becomes a source for powerful, real-time analytics.

- **Use Case Flow Analysis**:
  - **Report**: "Average Time between Payment and Shipment by Warehouse."
  - **Data**: Analyze the materialized state transition graphs (:Paid :precededBy :Placed). The timestamps of these Interaction events are sliced by the Warehouse dimension.
  - **Indicator**: AVG(timestamp(Shipped_Event) - timestamp(Paid_Event)) grouped by warehouse.
- **Product Concept Affinity**:
  - **Report**: "Cross-Sell Opportunity Identification."
  - **Data**: Analyze the FCA-derived concept lattices. Find products that are frequently co-occurring in the same :Purchase contexts but are not in the same explicit product category.
  - **Indicator**: "Concept Affinity Score" calculated as the Jaccard similarity of their FCA attribute sets (or the GCD of their CPPEs). A high score between ProductCategory:CampingGear and CustomerAttribute:Owns4x4Vehicle suggests a marketing opportunity.
- **Customer Lifetime Value (LTV)**:
  - **Report**: "True Customer LTV".
  - **Data**: Since all data is unified around a single customer DID, we can combine purchasing data (from ERP), marketing interaction data (from CRM), and customer support data (from a ticketing system).
  - **Indicator**: A comprehensive LTV calculation that was previously impossible becomes a straightforward query across the unified graph.