Implementation Roadmap: Application Service Framework

Version: 4.0 (Final)
Date: July 29, 2025

1. Introduction

This document provides the definitive, implementation-focused roadmap for the Application Service framework. It moves beyond high-level architecture to specify the granular, technical details required to build a fully reactive, functional, and behavior-driven system. This final version consolidates all previous architectural decisions, emphasizing a pure Semantic Web foundation, deterministic structural embeddings via Formal Concept Analysis (FCA), and a sophisticated, message-driven activation layer.

The appendices that follow provide an exhaustive exploration of the core models, the conversational API protocol, a complete federated use case, and the resulting business intelligence capabilities, serving as a complete blueprint for implementation.

Appendix A: The Unified Model Architecture

The framework's power lies in its layered, yet unified, model architecture. While conceptually distinct, the Reference, Graph, Activation, FCA, and Dimensional models are all implemented on a single underlying Semantic Web store (Apache Jena), with the Reference Model's reified statements acting as the unifying data structure.

A.1. The Reference Model (The Grammar of Identity)

This is the foundational layer, transforming raw data into formal, identifiable concepts.

- **Entities & Implementation**:
  - **ID**: Represents the canonical, context-free "idea" of a resource. It is immutable.
    - **Implementation**: A Java record ID(long primeID, String urn, String did).
  - **IDOccurrence**: Represents an ID appearing in a specific role within a context (a statement).
    - **Implementation**: A Java record IDOccurrence(ID occurringId, URI contextStatementUri). The contextStatementUri points to the reified rdf:Statement that represents the context.
- Unified Property Graph Representation (on Jena):
  We use RDF reification to create a property graph structure. A single resource (e.g., "Alice") is a URI (urn:appservice:id:937).

- **Nodes**: A resource is created with rdf:type. <urn:appservice:id:937> rdf:type :Resource .
- **Properties**: Standard triples. <urn:appservice:id:937> :hasName "Alice" .
- **Edges/Statements**: A statement (Alice, worksFor, Google) is reified, and this reified statement becomes the context for its participants.
  @prefix : <urn:appservice:ont#> .
  @prefix id: <urn:appservice:id#> .

  :stmt_456 a rdf:Statement, :IDOccurrence ; # The statement is an IDOccurrence
  rdf:subject id:937 ;
  rdf:predicate :worksFor ;
  rdf:object id:101 .

  This approach allows us to attach properties to edges (statements) and query the graph based on the roles entities play, unifying all model layers.

- **Statement Types & Messages**:
  - **Schema Statements**: Define the rules. (Context, Role, Dataflow). They describe the *types* of interactions possible.
  - **Data Statements**: Represent concrete events. (Interaction, Actor, Transform). They describe *instances* of interactions.
- **Inference & Traversal**:
  - **Capability**: Find all contexts a specific user participated in.
  - Implementation: A SPARQL query that selects all rdf:Statements where the rdf:subject or rdf:object is the user's ID.
    SELECT ?stmt WHERE { ?stmt a rdf:Statement; ?p <urn:appservice:id:user_Alice> . }

A.2. The FCA Model (The Logic of Structure)

This model provides deterministic, explainable embeddings and type inference.

- **FCA Contexts & Prime ID Products**:
  - **Model**: We define three FCA context relations from a single statement: (predicate, subject, object), (subject, predicate, object), and (object, subject, predicate).
  - **Implementation**: The Index Service can construct these contexts on demand. For a given relation (e.g., the predicate :worksFor), the context objects are the subjects, and the attributes are the objects.
  - Embedding: The Contextual Prime Product Embedding (CPPE) for an entity within a context is the product of the primeIDs of the entities it's related to.

CPPE(Google, worksFor) = primeID(Alice) * primeID(Bob) * ...

- **Inference & Traversal**:
  - **Similarity**: Similarity(A, B) = GCD(CPPE(A), CPPE(B)). The Greatest Common Divisor reveals the product of shared relations, providing a quantitative similarity score.
  - **Role Inference**: Complex roles like "uncle" can be inferred by composing CPPEs. Find "Person A" who is a :brotherOf "Person B", and "Person B" who is a :fatherOf "Person C". The system can infer a new triple: (A, :uncleOf, C). This is a deterministic graph traversal and arithmetic operation, far more efficient and explainable than vector-based similarity.

A.3. The Graph Model (The Semantics of Sets)

This model elevates the Reference Model by reifying statements into higher-order Kinds.

- **Entities & Implementation**:
  - **Kind**: A set of IDOccurrences that share common structural properties. A SubjectKind like :Customer is formed by grouping all Subjects that interact with a similar set of (Predicate, Object) pairs.
  - **Implementation**: A Kind is a rdfs:Class that is a rdfs:subClassOf its base type (e.g., <:Customer> rdfs:subClassOf <:SubjectKind>).
- **Inference & Traversal (Functional Interfaces)**:
  - **Capability**: "Given the :Customer type, what types of actions can they perform?"
  - **Implementation (Function<URI, Set<URI>>)**: A SPARQL query that finds all PredicateKinds connected to the given SubjectKind in schema statements.

A.4. The Activation Model (The Pragmatics of Behavior)

This model enables the execution of business processes using DCI, DDD, and a Dynamic Object Model.

- **Entities & Implementation**:
  - **DOM**: An Actor's data is a flexible Instance (record Instance(ID id, Class clazz, Map<URI, Instance> attributes)).
  - **DCI**: A Context (record Context(URI id, List<Role> roles)) defines a use case. An Interaction is an instance of a Context.
  - **Actor-Role**: An Actor is an Instance playing a Role. Its state is defined by its available Transforms (record ActorState(Transform previous, Transform current, List<Transform> next)).
- **Inference & Traversal**:
  - **Capability**: "Given a desired outcome (e.g., ProductShipped), what sequence

of Transforms is required?"
    - **Implementation**: A backward-chaining graph traversal (SPARQL query) starting from the desired final state and traversing the :precededBy relationships defined in the Dataflow schemas to construct a valid plan.

A.5. The Dimensional Model (The Analytics of Measurement)

This model provides powerful, OLAP-style analytics directly on the live graph.

- **Entities & Implementation**:
    - **Dimensional Context Statement**: We encode dimensional data using RDF reification to create explicit, queryable paths. A single sale event is encoded as a series of nested statements.
        - stmt1: The core fact (order_456 hasValue 118.00).
        - stmt2: Slices stmt1 by the Time dimension.
        - stmt3: Slices stmt2 by the Product dimension.
        - stmt4: Slices stmt3 by the Region dimension.
- **Inference & Traversal (Functional Retrieval)**:
    - **Capability**: "Show me all sales for the Pro-Lite shoe in Boston."
    - **Implementation**: A SPARQL SELECT query that finds all paths matching the nested statement pattern, filtering at each dimensional level. This provides a functional, real-time BI capability directly on the operational graph.

Appendix B: The Semantic Engine & COST/HAL Protocol

The Activation layer protocol is a conversational state dataflow, COST (COnversational State Transfer), implemented with HAL. It uses placeholders to guide the interaction, making the client dynamic and the server's logic flexible.

B.1. Encoding Behavior: Dataflow and Transforms

A Role's behavior (Dataflow) is defined as a sequence of declarative Transform definitions stored in the Jena graph.

- **TransformDef Record**:

```
record TransformDef(
   String name,
   OperationType operation, // e.g., ASSIGN, TRANSFER, COMPUTE, INVOKE_TOOL
   Map<String, FieldDef> inputs, // Named inputs for the operation
   Map<String, FieldDef> outputs // Named outputs produced
) {}

record FieldDef(String role, String field) {}
```

- Example: ProductBuy Context Dataflow
  The business logic aBuyer.accountBalance -= anAmount.amount is encoded as a TransformDef:
  ```
  {
    "name": "DebitBuyerAccount",
    "operation": "COMPUTE",
    "inputs": {
      "currentBalance": { "role": "Buyer", "field": "accountBalance" },
      "debitAmount": { "role": "Amount", "field": "amount" }
    },
    "outputs": {
      "newBalance": { "role": "Buyer", "field": "accountBalance" }
    }
  }
  ```

  The Activation Service reads this definition, populates it with actual Actor IDs, and dispatches it as an executable Transform message.

B.2. The COST/HAL Conversation with Placeholders

The power of COST lies in embedding placeholders within the HAL _links, turning the API into a guided conversation.

1. **Server Initiates**: The server starts a ProductBuy Interaction. It sends the initial HAL state. The _links.next array contains an action template.
   ```
   // Part of a HAL response for an Interaction
   "_links": {
     "next": [
       {
         "href": "/interactions/123/transform",
         "method": "POST",
         "name": "SelectProductForPurchase",
         "title": "Select a Product to Buy",
         "schema": { // The placeholder definition
           "type": "object",
           "properties": {
             "selectedProduct": {
               "type": "string",
               "description": "The DID of the product to purchase.",
               "_links": {
   ```

```
            "possibleValues": { "href":
"/interactions/123/roles/Product/possibleActors" }
          }
        }
      }
    }
  }
 ]
}
```

2. **Client Discovers Options (Client-Side Population)**:
   - The client UI sees the SelectProductForPurchase action and its schema.
   - It sees that the selectedProduct placeholder has a possibleValues link.
   - It performs a GET on /interactions/123/roles/Product/possibleActors.
   - The server responds with a list of available products (potential Actors for the Product Role). The UI renders this as a dropdown.
3. **Client Responds with Data**:
   - The user selects a product (e.g., did:ion:product_456).
   - The client constructs the Transform message body according to the schema: { "transformName": "SelectProductForPurchase", "payload": { "selectedProduct": "did:ion:product_456" } }.
   - It POSTs this body to the href.
4. **Server Infers and Prompts for Confirmation (Server-Side Population)**:
   - The server receives the selected product. Its internal logic determines that for this product, "FedEx" is the only valid shipping partner.
   - Instead of asking the client to choose, it pre-populates the next step and asks for confirmation.
   - It responds with a new HAL state where the next action is ConfirmShippingPartner, and the response body already contains the populated data:

     ```
     {
       "shippingPartner": {
         "name": "FedEx",
         "did": "did:ion:shipper_fedex"
       }
     }
     ```

5. **Client Confirms**: The UI displays "Shipping with: FedEx" and a "Confirm" button. Clicking it sends the ConfirmShippingPartner Transform message, and the

conversation continues. This conversational, placeholder-driven approach makes the interaction incredibly flexible and decoupled.

Appendix C: End-to-End Integration Use Case: A Federated Supply Chain

This appendix depicts a complete, multi-organization use case, demonstrating how three independent entities can form a seamless, automated, and intelligent supply chain.

C.1. The Participants & Their Systems

- **Manufacturer**: SportProducts Manufacturing Inc. (SPM)
  - **Systems**: SCM for production, ERP for orders.
  - **AppService Instance**: spm.appservice.com
- **Consumer**: Sport and Fitness Stores (SFS)
  - **Systems**: Legacy ERP for inventory, modern CRM for sales.
  - **AppService Instance**: sfs.appservice.com
- **Provider**: Sports Goods Raw Materials LLC (SGRM)
  - **Systems**: Simple order management database.
  - **AppService Instance**: sgrm.appservice.com

C.2. The Use Case: From Low Stock to Federated BI

Step 1: Low Inventory Trigger at the Retailer (SFS)

- **Services Layout & Roles**:
  - SFS.Datasource: Continuously ingests inventory levels from SFS's ERP via its JCA adapter.
  - SFS.Alignment: Aligns the raw stock number into a canonical Measure. It has previously aligned the "Pro-Lite Running Shoe" from the ERP with SPM's official product definition, creating an owl:sameAs link between did:sfs:product_789 and did:spm:product_ProLite.
  - SFS.Activation: An internal Context named MonitorInventory is constantly running.
- **Messages & Dataflow**:
  1. SFS.Datasource produces a raw statement: ("store_boston", "stock_PLRS", "49").
  2. SFS.Aggregation/Alignment processes this into a Graph Model statement.
  3. The SFS.Activation engine's MonitorInventory Interaction evaluates a rule. The stock level is below the threshold, so it starts a new ReplenishStock Interaction.

Step 2: Automated Order Placement via MCP (SFS -> SPM)

- **Services Layout & Roles**:
  - SFS.Activation: The ReplenishStock Interaction needs to place an order. It knows the canonical product is from SPM. It now acts as an **MCP Client**.
  - SPM.Activation: Exposes its capabilities as an **MCP Server**.
- **Messages & Dataflow**:
  1. The SFS AppService authenticates to the SPM AppService using **DID-Auth**.
  2. SFS queries SPM's MCP endpoint for available Tools related to did:spm:product_ProLite.
  3. SPM's MCP server responds, offering a PurchaseOrder Tool.
  4. SFS sends a standardized ToolCall message via MCP to SPM's endpoint, containing the required quantity.
  5. SPM.Activation receives this, starts a FulfillOrder Interaction, and begins a COST/HAL conversation back with the SFS agent to confirm pricing and delivery dates.

Step 3: Manufacturer Checks Raw Materials (SPM)

- **Services Layout & Roles**:
  - SPM.Activation: The FulfillOrder Interaction's Dataflow includes a Transform to check internal stock for raw materials (did:sgrm:material_eva_foam).
  - SPM.Datasource: Provides access to SPM's SCM system data.
- **Messages & Dataflow**:
  1. It queries its own Graph Model and finds the stock of "EVA Foam" is insufficient.
  2. This triggers a new internal Interaction: ProcureMaterials. This Interaction identifies the supplier for did:sgrm:material_eva_foam as SGRM. SPM's AppService now prepares to act as an MCP Client.

Step 4: Manufacturer Orders Raw Materials via MCP (SPM -> SGRM)

- The process from Step 2 repeats, with SPM acting as the MCP Client and SGRM as the MCP Server.

C.3. Business Intelligence Leverage

- **Internal BI**:
  - **SFS**: Can analyze its sales data, sliced by store, product, and time, to optimize marketing. They can create a KPI for "Sell-Through Rate".
  - **SPM**: Can analyze production data. By correlating FulfillOrder Interaction times with the ProcureMaterials Interaction times, they can create an indicator for "Production Delay due to Material Shortage."

- **Federated BI**:
  - Because all entities use a common (though decentralized) identity system (DIDs) and a common dimensional model, they can agree to share specific, anonymized ContextMeasure data.
  - They can build a federated view of the entire supply chain's health, analyzing end-to-end efficiency from raw material order to final consumer sale, without exposing sensitive internal operational data.