Functional Requirements Specification: Unified Application Integration Framework

Version 1.0

Date: 2025-07-14

1. Introduction

1.1 Purpose

This document provides a detailed breakdown of the functional requirements for the Unified Application Integration Framework. It elaborates on the Software Requirements Specification (SRS) by defining the specific behaviors, actions, inputs, and outputs for each component of the system. The purpose is to provide a clear and unambiguous guide for the development team, outlining precisely *what* the system must do.

1.2 Scope

This FRS covers the complete set of functions for all services within the framework's architecture, including data ingestion, the three layers of semantic augmentation (Aggregation, Alignment, Activation), service orchestration, shared helper services, and the final API exposure. Each requirement is intended to be testable.

2. Functional Requirements

FR-1: Datasource Service

This service is the entry point for all external data into the framework.

- **FR-1.1: Pluggable Data Source Connection**
  - **Description:** The service must provide a mechanism to connect to heterogeneous data sources. It shall support, at a minimum, relational databases via JDBC and REST APIs. The connection mechanism must be extensible to support other sources like document stores or message queues in the future.
  - **Inputs:** Connection parameters for a given data source (e.g., JDBC URL, username, password; API base URL, authentication token).
  - **Outputs:** An active connection to the specified data source.
  - **Processing:** The service shall use a factory pattern to instantiate the appropriate connector based on the source type. Connection details will be provided via a configuration file or an administrative UI.
- **FR-1.2: Source Schema Discovery**

    - **Description:** For structured sources like RDBMS, the service must be able to introspect the database schema to identify tables, columns, primary keys, and foreign keys.
    - **Inputs:** An active database connection.
    - **Outputs:** A structured representation of the source schema.
    - **Processing:** The service will use JDBC DatabaseMetaData to retrieve schema information. This information will be used to guide the transformation process in FR-1.3.
- **FR-1.3: Data-to-Quad Transformation**
    - **Description:** The service must transform data from the source format into a stream of raw Statement<String, String, String, String> quads. This process must be consistent and reversible where possible [Source: ApplicationService.odt, ETL].
    - **Inputs:** Data retrieved from a source (e.g., a row from a database table).
    - **Outputs:** A stream of quad Statement objects.
    - **Processing Rules:**
        - For a relational table row, each column's value shall be transformed into a quad: (Context: table_name, Subject: row_primary_key, Predicate: column_name, Object: column_value).
        - The service must handle data type conversions to their string representations.
        - The output stream of quads will be published to a dedicated Kafka topic.
- **FR-1.4: Data Synchronization**
    - **Description:** The service must implement a strategy to keep the framework's data synchronized with the source applications.
    - **Inputs:** A schedule (for polling) or a subscription endpoint (for event-driven updates).
    - **Outputs:** A continuous stream of new or updated data quads.
    - **Processing:** The service shall support both polling-based synchronization (e.g., querying for new rows based on a timestamp or ID) and, where possible, event-based synchronization (e.g., listening to database change data capture (CDC) events). It must maintain provenance information (source, timestamp) for each quad.

FR-2: Aggregation Service

This service consumes raw data and enriches it with basic semantic structure.

- **FR-2.1: URI to ID Assignment**
    - **Description:** The service must consume the raw string quads and assign a unique, persistent ID object to every unique URI (string) encountered in the

subject, predicate, and object positions.
    - **Inputs:** A stream of Statement<String, String, String, String>.
    - **Outputs:** A stream of Statement<ID, ID, ID, ID> (the Reference Model).
    - **Processing:**
        1. For each string in the input statement, check if an ID already exists in the **Registry Service**.
        2. If not, create a new ID object.
        3. The new ID must be assigned a unique prime number [Source: ApplicationService.odt, Aggregation]. A centralized counter or algorithm must ensure prime uniqueness.
        4. The new ID must be assigned a URN.
        5. The mapping from the string URI to the new ID is stored in the **Registry Service**.
        6. The output statement is constructed using the corresponding ID objects.
- **FR-2.2: Type Inference via FCA**
    - **Description:** The service must infer the "type" of a subject based on the set of predicates associated with it. Subjects sharing the same set of predicates are considered to be of the same type [Source: ApplicationService.odt, Aggregation].
    - **Inputs:** A stream of Reference Model statements.
    - **Outputs:** A set of (Subject_ID, rdf:type, Type_ID) statements added to the model.
    - **Processing:**
        1. Maintain a map Map<Set<Predicate_ID>, Type_ID>.
        2. For each Subject_ID, gather the set of all its Predicate_IDs.
        3. Look up this set in the map. If a Type_ID exists, assign this type to the subject.
        4. If not, create a new Type_ID, add it to the map, and then assign it to the subject.
        5. This process leverages Formal Concept Analysis (FCA), where subjects are objects and predicates are attributes.
- **FR-2.3: State Inference**
    - **Description:** The service must infer the "state" of a subject based on the specific values of its attributes (predicates). Subjects of the same type with identical predicate-value pairs are considered to be in the same state [Source: ApplicationService.odt, Aggregation].
    - **Inputs:** A stream of Reference Model statements, augmented with type information from FR-2.2.
    - **Outputs:** A set of (Subject_ID, hasState, State_ID) statements added to the

model.
    - **Processing:**
        1. Maintain a map Map<Type_ID, Map<Set<Pair<Predicate_ID, Object_ID>>, State_ID>>.
        2. For each subject, gather the set of all its <Predicate, Object> pairs.
        3. Using the subject's Type_ID, look up this set of pairs in the map to find the corresponding State_ID.
        4. If not found, create a new State_ID and update the map.
- **FR-2.4: Embedding Calculation**
    - **Description:** The service must calculate an embedding vector for each ID and IDOccurrence.
    - **Inputs:** An ID or IDOccurrence.
    - **Outputs:** A double[] embedding vector stored with the ID object.
    - **Processing:** The embedding for an ID shall be a function of its prime ID and the embeddings of its occurrences. The embedding for an IDOccurrence is a function of its own ID's embedding, its context's embedding, and the occurring ID's embedding [Source: ApplicationService.odt, Aggregation]. This can be implemented using techniques like graph embedding algorithms (e.g., Node2Vec) or simpler compositional methods. The resulting embeddings are stored in the **Index Service**.

FR-3: Alignment Service

This service takes the typed and state-inferred graph and aligns it with broader semantic models.

- **FR-3.1: Domain Ontology Alignment**
    - **Description:** The service must match and align inferred types from different source applications into a unified "Domains Upper Ontology".
    - **Inputs:** The Graph Model containing inferred types from multiple sources.
    - **Outputs:** A set of owl:equivalentClass or rdfs:subClassOf statements linking source types to the upper ontology concepts.
    - **Processing:**
        1. Use ML clustering (on embeddings from the **Index Service**) or FCA-based similarity measures [Ref: Similarity measures in formal concept analysis] to identify candidate types for alignment.
        2. For example, if Type_A from the Inventory app and Type_B from the Orders app are found to be highly similar, a new upper concept IntegratedProduct can be created, and statements (Type_A, rdfs:subClassOf, IntegratedProduct) and (Type_B, rdfs:subClassOf, IntegratedProduct) are generated.

3. This alignment is stored in the **Naming Service**.

- **FR-3.2: Dimensional Order Alignment**
  - **Description:** The service must arrange entities and their values along ordered dimensions (e.g., time, geography, state progressions) in an "Order Upper Ontology" [Source: ApplicationService.odt, Alignment].
  - **Inputs:** Type and state hierarchies from the Aggregation Service.
  - **Outputs:** A set of statements defining ordering relationships (e.g., (State_Married, follows, State_Single)).
  - **Processing:**
    1. Analyze inferred state hierarchies (e.g., Child -> Young -> Old).
    2. Materialize these sequences using a standard vocabulary (e.g., flow:next, time:before).
    3. For quantitative values, it must align them into dimensional units (e.g., map various price attributes to a single Currency dimension).
- **FR-3.3: Link Completion**
  - **Description:** The service must infer and add missing relationships (links) to the graph based on logical patterns.
  - **Inputs:** The aligned Graph Model.
  - **Outputs:** New triple statements representing inferred links.
  - **Processing:** This can be implemented using rule-based inference (e.g., SPARQL CONSTRUCT queries or a rules engine like Drools) or statistical relational learning models. An example rule is the "uncle" relationship: (?s :brotherOf ?o), (?o :fatherOf ?o2) -> (?s :uncleOf ?o2) [Source: ApplicationService.odt, Dimensional Features].

FR-4: Activation Service

This service interprets the semantically rich graph to discover and execute application behaviors.

- **FR-4.1: Use Case (Context) Inference**
  - **Description:** The service must analyze the aligned graph to identify potential use cases, modeled as DCI Contexts. A Context is defined by a set of interacting Roles [Source: ApplicationService.odt, Activation].
  - **Inputs:** The aligned Graph Model.
  - **Outputs:** A catalog of Context definitions stored in the **Registry Service**.
  - **Processing:** Identify patterns of interaction between different types in the graph. For example, a consistent pattern of Order types being created by Customer types and fulfilled by Inventory types would be inferred as a Sales context, with Buyer, Seller, and Product as its Roles.
- **FR-4.2: Role Fulfillment Inference**

    - **Description:** The service must determine which entity types are capable of playing which Roles in a given Context.
    - **Inputs:** A Context definition.
    - **Outputs:** Mappings between Roles and the entity types that can fulfill them.
    - **Processing:** This is based on matching the attributes required by a Role with the attributes possessed by an entity type.
- **FR-4.3: Interaction Instantiation and Execution**
    - **Description:** The service must allow a client (via the Producer API) to instantiate a Context into a concrete Interaction by assigning specific Actors (instances) to the Roles. It must then manage the state transitions of this Interaction.
    - **Inputs:** A Context ID and a map of Role IDs to Actor IDs.
    - **Outputs:** A new Interaction instance with a unique ID and an initial state.
    - **Processing:**
        1. Create a new Interaction resource.
        2. Link the assigned Actors to the Interaction.
        3. Determine the initial step in the Interaction's dataflow. The dataflow logic itself is inferred from existing transaction patterns or defined declaratively [Source: ApplicationService.odt, Activation].
        4. Subsequent calls to execute steps in the Interaction will trigger state transitions according to the defined dataflow.
- **FR-4.4: Goal-Based Scenario Generation**
    - **Description:** The service must be able to respond to a query describing a desired outcome by generating a list of possible Interactions (and potential Actor assignments) that could achieve it [Source: ApplicationService.odt, Producer].
    - **Inputs:** A description of a goal (can be structured or natural language processed by Spring AI).
    - **Outputs:** A list of candidate Interaction plans.
    - **Processing:** This requires backward-chaining inference. Starting from the desired final state, the service works backward through the possible dataflow transitions to find valid starting points and actor combinations.

FR-5: Producer API Service

This service is the single, unified gateway for external clients to interact with the framework.

- **FR-5.1: Expose Contexts as Navigable Resources**
    - **Description:** The API must provide endpoints to list all available Contexts (use cases).

    - **Inputs:** An HTTP GET request to /contexts.
    - **Outputs:** A HAL-formatted JSON response listing available Contexts, each with a _link to start a new Interaction.
- **FR-5.2: Manage Interaction Lifecycle**
    - **Description:** The API must provide endpoints to create, retrieve, update, and execute Interactions.
    - **Inputs:** HTTP POST, GET, PUT requests to /interactions and /interactions/{id}.
    - **Outputs:** HAL-formatted JSON representations of Interaction resources, with links for valid next actions (e.g., "next_step": { "href": "/interactions/{id}/execute" }).
- **FR-5.3: Generic Form/Wizard Metadata Generation**
    - **Description:** When a client needs to provide data for an Interaction step, the API must respond with metadata describing the required fields, types, and constraints.
    - **Inputs:** An HTTP GET request to a resource representing an Interaction step.
    - **Outputs:** A JSON object describing the form fields (e.g., using a schema like JSON Schema or a custom format) [Source: ApplicationService.odt, Producer].

FR-6: Helper Services (Functional View)

- **FR-6.1: Registry Service Functions**
    - **FR-6.1.1: Statement Storage:** Must provide CRUD operations for Statement objects across all four model layers.
    - **FR-6.1.2: SPARQL Endpoint:** Must expose a standard SPARQL 1.1 endpoint for complex queries against the entire graph model [Source: ApplicationService.odt, Registry Service].
    - **FR-6.1.3: Provenance Tracking:** For every statement, it must store metadata about its source, the service that created/modified it, and the timestamp.
- **FR-6.2: Naming Service Functions**
    - **FR-6.2.1: Ontology Storage:** Must provide an interface to store and manage the Domains and Order upper ontologies.
    - **FR-6.2.2: Alignment Lookup:** Must provide a function to resolve an inferred type/predicate to its corresponding concept in an upper ontology.
- **FR-6.3: Index Service Functions**
    - **FR-6.3.1: Embedding Storage:** Must provide an interface to store and retrieve the embedding vector for any given ID.
    - **FR-6.3.2: Similarity Search:** Must provide a function that, given an ID and a context, returns a ranked list of the most similar IDs based on cosine similarity of their embeddings [Source: ApplicationService.odt, Index Service].