Software Requirements Specification: Unified Application Integration Framework

Version 1.0

Date: 2025-07-14

1. Introduction

This document provides a detailed Software Requirements Specification (SRS) for a reactive, semantics-driven framework designed to integrate diverse legacy and modern applications. The framework's core purpose is to parse source data, infer a layered semantic model, and expose the combined functionalities and data as a unified set of use cases through a generic API.

1.1 Purpose

The goal of this SRS is to provide a complete and unambiguous description of the functions, features, and constraints of the Unified Application Integration Framework. It is intended for developers, architects, and project managers to understand the system's architecture, requirements, and implementation guidelines. The system will ingest data from various sources, apply semantic enrichment through a series of processing layers, and expose the inferred application behaviors as interactive "Contexts" and "Interactions" [Source: ApplicationService.odt, Overview].

1.2 Scope

The project scope covers the design, development, and deployment of a microservices-based architecture that includes:

- **Data Ingestion:** An ETL-like service to extract and unify data from sources like relational databases, APIs, and documents.
- **Semantic Augmentation:** A pipeline of services (Aggregation, Alignment, Activation) to build a multi-layered knowledge graph.
- **Use Case Exposure:** A producer service that exposes the inferred use cases and transactions through a generic REST API and a browsable front-end.
- **Helper Services:** A set of shared services for managing the core model, ontologies, and resource indexing.

The framework will leverage reactive programming principles, semantic web technologies, and machine learning to achieve its goals.

1.3 Definitions, Acronyms, and Abbreviations

Term	**Definition**
SRS	Software Requirements Specification
ETL	Extract, Transform, Load
API	Application Programming Interface
DCI	Data, Context, and Interaction - A software architecture pattern [Ref: dci.github.io]
FCA	Formal Concept Analysis - A mathematical method for data analysis [Ref: en.wikipedia.org/wiki/Formal_concept_analysis]
RDF	Resource Description Framework - A W3C standard for data interchange [Ref: w3.org/RDF]
SPARQL	SPARQL Protocol and RDF Query Language
SPO	Subject, Predicate, Object - The components of an RDF triple.
CSPO	Context, Subject, Predicate, Object - An RDF quad.
DID	Decentralized Identifier - A W3C standard for verifiable, decentralized digital identity [Ref: w3.org/TR/did-core/]
LLM	Large Language Model
MCP	Model-Context-Protocol - A protocol for interacting with AI models [Ref: modelcontextprotocol.io]
HAL	Hypertext Application Language - A standard for defining hypermedia controls in APIs [Ref: stateless.group/hal_specification.html]

1.4 References

This document is based on:

- ApplicationService.odt (The source document for this specification).
- Spring Project Documentation (spring.io/reactive,

docs.spring.io/spring-ai/reference/)

- Eclipse RDF4J Documentation (rdf4j.org)
- Formal Concept Analysis research papers (See Appendix).
- DCI, DDD, and Reactive Architecture literature (See Appendix).
- W3C Standards for RDF, SPARQL, and DIDs.

1.5 Overview

The remainder of this document details the system's overall architecture, core data models, functional and non-functional requirements, and provides concrete implementation guidelines and examples to facilitate development.

2. Overall Description

2.1 Product Perspective

The framework is a middleware solution that sits between existing application data sources and new consumer applications. It acts as a "generator of unified interfaces" by creating a semantic abstraction layer over heterogeneous systems [Source: ApplicationService.odt, Overview]. It does not replace existing applications but rather integrates them, enabling new, cross-domain use cases.

Figure 1: High-Level System Architecture, adapted from [Source: ApplicationService.odt, Diagram 1].

2.2 Product Functions

- **Integrate Diverse Data:** Connect to and synchronize with various data sources (SQL, NoSQL, APIs, files).
- **Infer Semantic Models:** Automatically infer types, states, relationships, and hierarchies from raw data.
- **Align Ontologies:** Match and merge concepts from different domains into a unified upper ontology.
- **Discover Use Cases:** Infer potential application behaviors (Contexts) and their transactional instances (Interactions).
- **Expose Unified API:** Provide a generic, navigable REST API for browsing and executing these inferred use cases.

2.3 User Characteristics

- **System Administrators/Integrators:** Responsible for configuring datasources, managing the services, and monitoring the integration process. They will interact with management interfaces for each service.
- **Developers:** Will use the Producer API to build new applications (e.g., web

front-ends, chatbots, other services) on top of the integrated system.
- **End-Users:** Will interact with the applications built by developers, effectively using the functionality of the underlying legacy systems through a unified, modern interface.

2.4 Constraints

- **Technology Stack:** The implementation shall be based on a reactive Java framework, specifically **Spring Boot** with **Project Reactor** for its reactive capabilities [Ref: spring.io/reactive].
- **Data Storage:** The core graph model shall be stored in a triple/quad store that supports SPARQL, with **Eclipse RDF4J** being the recommended implementation [Ref: rdf4j.org].
- **Communication:** Inter-service communication shall be primarily event-driven, using a message broker like Apache Kafka.
- **API Standard:** The public-facing API shall adhere to REST principles and use **HAL** (Hypertext Application Language) to ensure discoverability and navigability [Ref: stateless.group/hal_specification.html].

2.5 Assumptions and Dependencies

- **Data Accessibility:** The framework assumes it has read/write access to the backend datasources of the applications to be integrated.
- **Data Quality:** The quality and consistency of the source data will directly impact the quality of the inferred models.
- **Infrastructure:** The system will be deployed as a set of containerized microservices, requiring an orchestration platform like Kubernetes.

3. System Architecture

3.1 Architectural Style

The system shall be implemented using a **Reactive Microservices Architecture**. This choice is driven by the need for a scalable, resilient, and responsive system capable of handling streams of data and events efficiently [Source: ApplicationService.odt, Services]. Each service is an independently deployable component that communicates asynchronously.

3.2 Technology Stack

Component	Technology	Rationale	Reference
Backend Framework	Spring Boot 3+	Robust, mature	spring.io

		ecosystem for building microservices.	
Reactive Programming	Project Reactor	Core of Spring WebFlux, enables non-blocking, event-driven logic.	spring.io/reactive
Triple Store	Eclipse RDF4J	Mature Java framework for RDF processing and storage.	rdf4j.org
Messaging	Apache Kafka	High-throughput, persistent event streaming platform.	kafka.apache.org
LLM/MCP Integration	Spring AI	Simplifies integration with LLMs and supports protocols like MCP.	docs.spring.io/spring-ai/
Containerization	Docker / Kubernetes	Standard for deploying and managing microservices.	docker.com

3.3 Communication Patterns

- **Internal Communication (Service-to-Service):** Asynchronous, event-driven communication via Kafka streams. This decouples services and improves fault tolerance. The **Saga pattern** shall be used to manage distributed transactions and maintain data consistency across services [Source: ApplicationService.odt, Augmentation Service].
- **External Communication (Client-to-API):** Synchronous, request-response communication via a REST API exposed by the Producer Service. The API will be reactive (non-blocking) from end-to-end using Spring WebFlux.

4. Core Concepts & Data Models

The framework is built upon a layered data model that evolves as data passes through the augmentation pipeline.

4.1 The Layered Statement Model

The fundamental unit of data is a Statement. Its representation changes as it's processed by each service layer, adding semantic depth at each stage.

1. **Datasource Model:** Statement<String, String, String, String>
   - Raw triples/quads where each component is a simple string extracted from the source.
   - Example: ("products_table", "row_123", "product_name", "Laptop")
2. **Reference Model (Aggregation):** Statement<ID, ID, ID, ID>
   - Strings are replaced by internal, unique ID objects. Each ID has a unique prime number and an embedding vector for similarity calculations.
   - [Source: ApplicationService.odt, Aggregation]
3. **Graph Model (Alignment):** Statement<Context, Subject, Predicate, Object>
   - IDs are resolved into semantic entities, representing a formal knowledge graph with aligned ontologies.
   - [Source: ApplicationService.odt, Alignment]
4. **Activation Model (Activation):** Statement<Context, Interaction, Role, Actor>
   - The graph is interpreted in terms of application behavior, based on the DCI pattern.
   - [Source: ApplicationService.odt, Activation]

4.2 Core Class Definitions

The following Java records define the core data structures. They should be serializable to JSON for transport between services.

```
// 1. Core Identifier with Prime Number and Embedding for FCA
public record ID(long primeId, String urn, double[] embedding, List<IDOccurrence>
occurrences) {}

// 2. An occurrence of an ID within a specific context
public record IDOccurrence(ID occurringId, IDOccurrence context, double[]
embedding) {}

// 3. The generic Statement structure, specialized by layer
// Using generics to represent the layered model
public record Statement<C, S, P, O>(C context, S subject, P predicate, O object) {}

// DCI-based classes for the Activation Layer
public record Context(ID id, String label, Map<String, Role> roles) {}
public record Role(ID id, String label, Map<Context, Dataflow> transitions) {}
public record Interaction(ID id, String label, Map<String, Actor> actors) {}
public record Actor(ID id, String label, Instance instanceData) {}
public record Dataflow(Role targetRole, Rule executionRule) {}
public record Instance(ID id, Class type, Map<String, Instance> attributes) {}
```

Code Example 1: Core Data Model in Java

4.3 Set-Based Representation and Reification

The framework utilizes a set-based interpretation of the knowledge graph to facilitate inference. As depicted in the Venn diagram [Source: ApplicationService.odt, Diagram 2], the sets of all Subjects, Predicates, and Objects in the graph have meaningful intersections:

- **SubjectKind:** Predicate ∩ Object - Represents a type defined by the predicates it can have and the objects it can be linked to.
- **PredicateKind:** Subject ∩ Object - Represents a relationship type defined by the kinds of subjects and objects it connects.
- **ObjectKind:** Subject ∩ Predicate - Represents a value type or literal defined by the subjects and predicates it is associated with.
- **ContextKind (Statement):** Subject ∩ Predicate ∩ Object - The intersection of all three represents a fully contextualized statement or event.

Figure 2: Venn Diagram of Set-Based Model [Source: ApplicationService.odt, Diagram 2]

This model allows for powerful functional programming techniques to be applied to streams of statements for inference, e.g., finding all subjects of a certain SubjectKind.

4.4 Formal Concept Analysis (FCA) and Embeddings

FCA is used to infer type hierarchies and cluster entities [Source: ApplicationService.odt, Aggregation].

- **Prime IDs:** Each unique URI (ID) is assigned a unique prime number at creation. This allows for efficient set-based calculations.
- **FCA Context:** An FCA context is a triplet (G, M, I) where G is a set of objects (e.g., Subjects), M is a set of attributes (e.g., Predicates), and I is a binary relation. For example, a Predicate FCA Context would be (Subjects, Objects, I) for a given predicate.
- **Embeddings:** Each ID and IDOccurrence has an embedding vector. The embedding for an ID is derived from its prime ID and the embeddings of its occurrences. Similarity between entities can be calculated using vector similarity

(e.g., cosine similarity) on these embeddings, which is crucial for the Index Service.

5. Functional Requirements

5.1 FR-1: Datasource Service

- **FR-1.1: Data Ingestion:** The service must provide a pluggable connector architecture to support various data sources (JDBC for RDBMS, custom connectors for APIs/documents).
- **FR-1.2: Data Transformation:** It must transform source data into raw Statement<String, String, String, String> quads. For a relational table, this means representing each cell as (tableName, rowPK, columnName, cellValue) [Source: ApplicationService.odt, ETL].
- **FR-1.3: Synchronization:** The service must periodically poll or subscribe to changes in the source data to ensure the unified model remains up-to-date. It must handle data provenance to track the origin of every statement.

5.2 FR-2: Augmentation Service (Orchestrator)

- **FR-2.1: Service Orchestration:** This service acts as the central coordinator. It consumes streams from the Datasource service and dispatches them to the Aggregation, Alignment, and Activation services in sequence.
- **FR-2.2: Event Dispatching:** It shall manage the Kafka topics for inter-service communication and implement the Saga pattern for managing long-running, distributed transactions.
- **FR-2.3: Context Management:** It maintains the conversational state for interactions with the Producer API, routing requests and responses between the client and the appropriate backend service.

5.3 FR-3: Aggregation Service

- **FR-3.1: Type/State Inference:** The service shall analyze streams of Reference Model statements to infer types and states.
    - **Type Inference:** Subjects sharing a common set of predicates are inferred to belong to the same type [Source: ApplicationService.odt, Aggregation].
    - **State Inference:** Subjects of the same type sharing common predicate-value pairs are inferred to be in the same state.
- **FR-3.2: Hierarchy Generation:** It must build type and state hierarchies based on subset/superset relationships of attributes and values (e.g., Employee is a subtype of Person).
- **FR-3.3: ID/Embedding Generation:** It consumes raw statements, assigns a unique ID (with a prime number) to each new URI, calculates its embedding, and

produces Reference Model statements.

5.4 FR-4: Alignment Service

- **FR-4.1: Upper Ontology Alignment:** The service shall align inferred concepts with high-level, canonical ontologies.
  - **Domains Upper Ontology:** Merges concepts from different integrated applications (e.g., Product in inventory and Item in orders are mapped to a single IntegratedProduct concept).
  - **Order Upper Ontology:** Arranges entities and values along dimensions like time, space, or other ordered scales (e.g., Single -> Married -> Divorced) [Source: ApplicationService.odt, Alignment].
- **FR-4.2: Ontology Matching:** Using ML clustering and FCA, it must find and map equivalent entities and relationships between domains.
- **FR-4.3: Link Inference:** It shall infer missing links or attributes in the graph. For example, if (S, brotherOf, O) and (O, fatherOf, O2), it can infer (S, uncleOf, O2) based on predefined rules or learned patterns.

5.5 FR-5: Activation Service

- **FR-5.1: Use Case (Context) Inference:** This service analyzes the aligned Graph Model to discover potential use cases (Contexts). A Context is defined by a set of interacting Roles.
- **FR-5.2: Role and Actor Inference:** It determines which entity types can play which Roles in a given Context.
- **FR-5.3: Interaction Instantiation:** It allows for the creation of new Interactions (instances of a Context). An Interaction assigns specific Actors (instances of entity types) to the Roles of the Context.
- **FR-5.4: Dataflow Inference:** It must materialize the business logic of a transaction. This can be done by generating declarative rules or even executable scripts (e.g., XSLT transforms) that define the data flow between actors in an interaction [Source: ApplicationService.odt, Activation].

5.6 FR-6: Producer API Service

- **FR-6.1: Generic REST API:** It must expose the Activation Model through a reactive, non-blocking REST API (using Spring WebFlux).
- **FR-6.2: HATEOAS/HAL Compliance:** API responses must include hypermedia links (_links) that allow clients to navigate the application state. For example, an Interaction resource should contain links to activate, update, or cancel it.
- **FR-6.3: Generic UI Rendering:** The service should be able to generate metadata for forms/wizards based on a Context's definition, allowing a generic client to render an appropriate UI for any transaction.

- **FR-6.4: Conversational Interaction:** It must support "goal-seeking" queries, where a user can describe a desired outcome, and the service uses the Activation Service to find possible Interactions that could achieve it [Source: ApplicationService.odt, Producer].

5.7 FR-7: Helper Services

- **FR-7.1: Registry Service:** A repository for the core graph model, storing statements from all layers. It must support SPARQL queries and provide provenance tracking. It acts as the "shared state" [Source: ApplicationService.odt, Registry Service].
- **FR-7.2: Naming Service:** Manages the upper ontologies and provides functions for matching and aligning concepts. It is heavily used by the Alignment Service.
- **FR-7.3: Index Service:** A repository of all addressable resources (IDs) and their embeddings. It provides fast similarity search capabilities (e.g., "find all resources similar to this one") [Source: ApplicationService.odt, Index Service].

6. Non-Functional Requirements

ID	Requirement	Description
NFR-1	**Performance**	All I/O operations must be non-blocking. The system should handle high-throughput streams of data with low latency, leveraging the reactive stack.
NFR-2	**Scalability**	All microservices must be stateless (or manage state externally in Kafka/RDF4J) and capable of being scaled horizontally and independently.
NFR-3	**Reliability**	The system must be fault-tolerant. The failure of one service should not cascade and bring down the entire system. The Saga pattern will ensure eventual consistency for distributed transactions.
NFR-4	**Security**	The system shall support

		decentralized identity management using W3C DIDs for identifying resources and actors. All API endpoints must be secured (e.g., using OAuth2).
NFR-5	**Interoperability**	The framework must strictly adhere to open standards (RDF, SPARQL, HAL, DIDs) to facilitate integration with other systems.
NFR-6	**Manageability**	Each service must expose an administrative interface for configuration, monitoring, and management [Source: ApplicationService.odt, Services].

7. Implementation Guidelines & Examples

7.1 Reactive Endpoint with Spring WebFlux

This example shows a simplified reactive controller endpoint in the Producer API Service.

```
@RestController
@RequestMapping("/interactions")
public class InteractionController {

   private final ActivationService activationService;

   // Constructor injection
   public InteractionController(ActivationService activationService) {
      this.activationService = activationService;
   }

   @GetMapping("/{id}")
   public Mono<Interaction> getInteractionById(@PathVariable String id) {
      return activationService.findInteractionById(id);
   }
```

```
    @PostMapping("/{contextId}/start")
    public Mono<Interaction> startNewInteraction(@PathVariable String contextId,
@RequestBody Map<String, String> actorAssignments) {
        // actorAssignments maps Role URNs to Actor URNs
        return activationService.createInteraction(contextId, actorAssignments);
    }

    @PostMapping("/{interactionId}/execute")
    public Mono<Interaction> executeNextStep(@PathVariable String interactionId,
@RequestBody Map<String, Object> stepData) {
        // The service determines the 'next' state and returns the updated interaction
        return activationService.processInteraction(interactionId, stepData);
    }
}
```

Code Example 2: Reactive Controller in Producer Service

7.2 RDF4J Integration

The helper services (Registry, Naming) will use RDF4J to interact with the triple store.

```
import org.eclipse.rdf4j.repository.Repository;
import org.eclipse.rdf4j.repository.RepositoryConnection;
import org.eclipse.rdf4j.repository.sparql.SPARQLRepository;
import org.eclipse.rdf4j.query.TupleQuery;
import org.eclipse.rdf4j.query.TupleQueryResult;

public class RegistryRepository {

    private final Repository repository;

    public RegistryRepository(String sparqlEndpoint) {
        // Connect to a remote SPARQL endpoint (e.g., Fuseki, GraphDB)
        this.repository = new SPARQLRepository(sparqlEndpoint);
    }

    public void findSubjectsByType(String typeURI) {
        String queryString = """
            PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
            SELECT ?subject WHERE {
                ?subject rdf:type <%s> .
            }
            """.formatted(typeURI);

        try (RepositoryConnection conn = repository.getConnection()) {
            TupleQuery query = conn.prepareTupleQuery(queryString);
            try (TupleQueryResult result = query.evaluate()) {
                while (result.hasNext()) {
                    System.out.println(result.next().getValue("subject"));
                }
            }
        }
    }
}
```

Code Example 3: Querying the Triple Store with RDF4J

7.3 Spring AI for Goal-Seeking

The Producer API can use Spring AI to translate natural language queries into structured calls to the Activation Service.

```
@Service
public class ConversationalService {

    private final ChatClient chatClient; // From Spring AI

    public ConversationalService(ChatClient chatClient) {
        this.chatClient = chatClient;
    }

    // This method would be called by the Producer API when a user asks a question
    public Mono<ActivationPlan> getPlanForGoal(String naturalLanguageGoal) {
        String promptTemplate = """
            Given the user's goal: "{goal}", determine the primary Context (use case)
            and the key Roles that need to be filled.
            Return the result as a JSON object with keys "context" and "roles".
            Example goal: "I want to launch a new product to the market."
            Example output: { "context": "urn:context:product_launch", "roles": 
["manufacturer", "advertiser"] }

            Goal: {goal}
            """;

        Prompt prompt = new Prompt(new 
UserMessage(promptTemplate.replace("{goal}", naturalLanguageGoal)));

        return chatClient.call(prompt)
                .map(response -> {
                    // Parse the JSON response from the LLM into an ActivationPlan object
                    // This plan can then be used to query the Activation Service
                    return parseJsonToPlan(response.getResult().getOutput().getContent());
                });
    }

    // Inner record for the plan
    public record ActivationPlan(String context, List<String> roles) {}
}
```

Code Example 4: Using Spring AI for Natural Language Understanding

8. Appendices

8.1 Glossary

(A more extensive glossary would be built here based on the final terminology)

8.2 References

(A comprehensive list of all cited URLs and documents would be compiled here)