APEX - Technical Reference Guide

Version: 1.0 **Date:** 2025-08-22 **Author:** Mark Andrew Ray-Smith Cityline Ltd

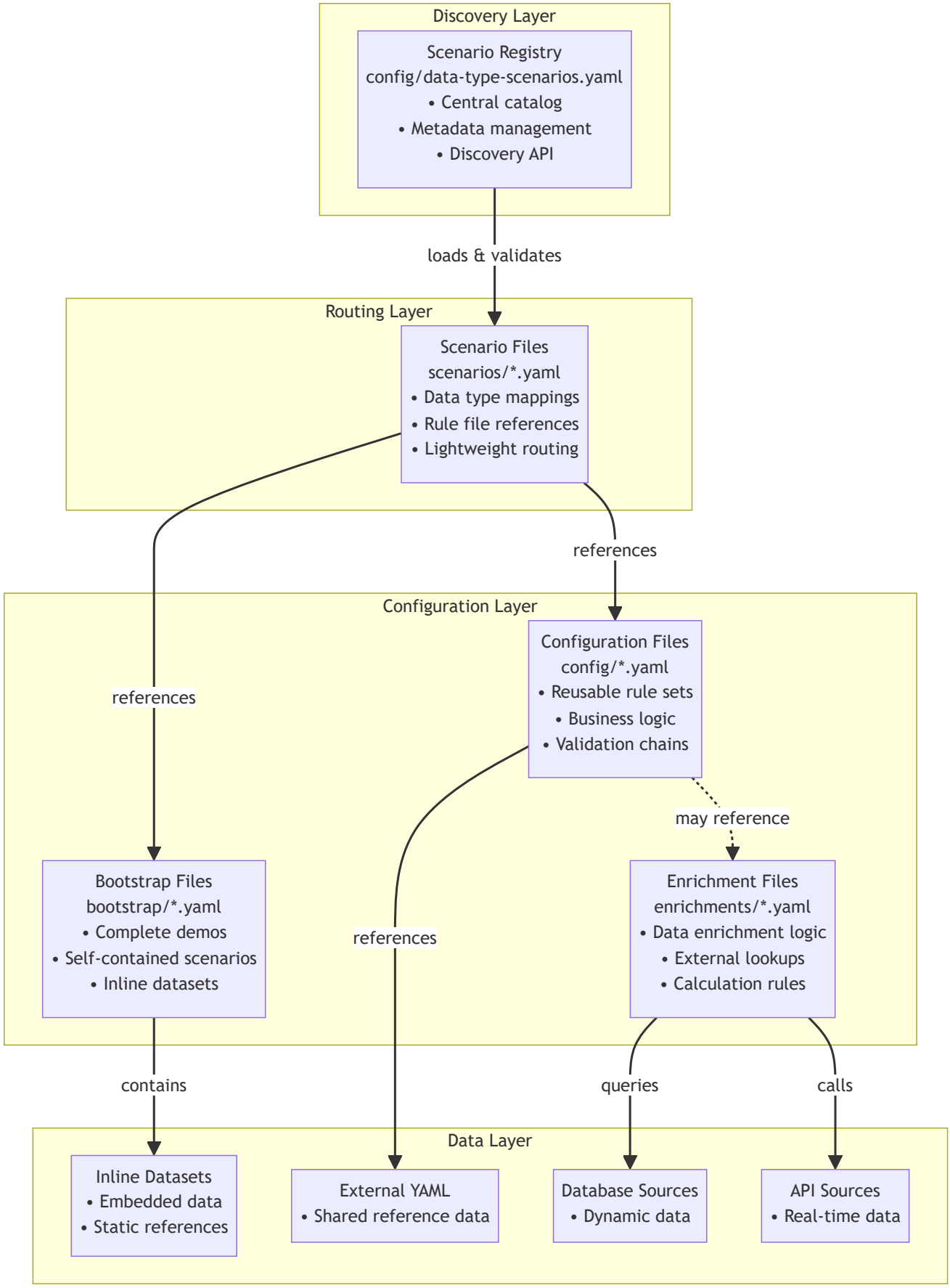
Welcome to the APEX Technical Reference Guide! This document provides detailed technical information for developers, architects, and system integrators working with APEX. While the User Guide focuses on getting started and common use cases, this reference dives deep into the technical architecture, advanced patterns, implementation details, scenario-based configuration management, and enterprise YAML validation systems.

Scenario-Based Configuration Architecture

Technical Overview

APEX's scenario-based configuration system provides a sophisticated architecture for managing complex rule configurations through a three-layer hierarchy. This system enables enterprise-scale configuration management with centralized discovery, type-safe routing, and comprehensive dependency tracking.

Architecture Components



Core Components

1. DataTypeScenarioService

The central service for scenario management and data type routing:

```
public class DataTypeScenarioService {

    // Load scenarios from registry
    public void loadScenarios(String registryPath) throws ScenarioException;

    // Get scenario for specific data type
    public ScenarioConfiguration getScenarioForData(Object data) throws ScenarioException;

    // Get scenario by ID
    public ScenarioConfiguration getScenario(String scenarioId) throws ScenarioException;

    // List all available scenarios
    public List<ScenarioConfiguration> getAvailableScenarios();

    // Validate scenario configuration
    public ScenarioValidationResult validateScenario(String scenarioId);
}
```

2. ScenarioConfiguration

Represents a loaded scenario with metadata and rule file references:

```
public class ScenarioConfiguration {
    private String scenarioId;
    private String name;
    private String description;
    private List<String> dataTypes;
    private List<String> ruleConfigurations;
    private Map<String, Object> metadata;

    // Getters and utility methods
    public boolean supportsDataType(Class<?> dataType);
    public boolean supportsDataType(String dataTypeName);
    public List<String> getRuleConfigurations();
}
```

3. YamlDependencyAnalyzer

Advanced dependency analysis and validation system:

```
public class YamlDependencyAnalyzer {

    // Analyze complete dependency chain
    public YamlDependencyGraph analyzeYamlDependencies(String rootFile);

    // Generate comprehensive reports
    public String generateTextReport(YamlDependencyGraph graph);
    public String generateTreeReport(YamlDependencyGraph graph);

    // Validate dependency health
    public DependencyHealthReport validateDependencies(YamlDependencyGraph graph);
}
```

YAML Validation System

Technical Architecture

APEX includes a comprehensive YAML validation system that ensures configuration integrity across all file types:

```
public class YamlMetadataValidator {

    // Validate single file
    public YamlValidationResult validateFile(String filePath);

    // Validate multiple files
    public YamlValidationSummary validateFiles(List<String> filePaths);

    // Type-specific validation
    private void validateScenarioContent(Map<String, Object> yamlContent, YamlValidationResult result);
    private void validateBootstrapContent(Map<String, Object> yamlContent, YamlValidationResult result);
    private void validateRuleConfigContent(Map<String, Object> yamlContent, YamlValidationResult result);
}
```

Validation Features

Metadata Validation:

- Required fields: `name`, `version`, `description`, `type`
- Type-specific required fields (e.g., `business-domain` for scenarios)
- Semantic versioning format validation
- Ownership and contact information validation

Content Validation:

- Scenario files: `scenario-id`, `data-types`, `rule-configurations`
- Registry files: Valid registry entries with required fields
- Bootstrap files: Expected content sections
- Rule config files: Rules, enrichments, or rule-chains sections

Dependency Validation:

- Complete dependency chain analysis
- Missing file detection
- Circular dependency detection
- YAML syntax validation

File Type System

APEX supports the following standardized file types:

Type	Purpose	Required Fields	Content Validation
`scenario`	Data type routing	`business-domain`, `owner`	`scenario` section with `data-types` and `rule-configurations`
`scenario-registry`	Central registry	`created-by`	`scenario-registry` list with valid entries

Type	Purpose	Required Fields	Content Validation
`bootstrap`	Complete demos	`business-domain` , `created-by`	`rule-chains` or `categories` sections
`rule-config`	Reusable rules	`author`	`rules` , `enrichments` , or `rule-chains` sections
`dataset`	Reference data	`source`	`data` , `countries` , or `dataset` sections
`enrichment`	Data enrichment	`author`	Enrichment-specific content
`rule-chain`	Sequential rules	`author`	Rule chain definitions

Implementation Patterns

1. Scenario-Driven Processing

```
@Service
public class ScenarioBasedProcessor {

    @Autowired
    private DataTypeScenarioService scenarioService;

    @Autowired
    private RuleEngineService ruleEngine;

    public ProcessingResult process(Object data) {
        // 1. Discover scenario for data type
        ScenarioConfiguration scenario = scenarioService.getScenarioForData(data);

        // 2. Load and execute rule configurations
        List<String> ruleFiles = scenario.getRuleConfigurations();
        ProcessingResult result = new ProcessingResult();

        for (String ruleFile : ruleFiles) {
            RuleConfiguration rules = loadRuleConfiguration(ruleFile);
            RuleExecutionResult ruleResult = ruleEngine.execute(rules, data);
            result.addRuleResult(ruleResult);
        }

        return result;
    }
}
```

2. Configuration Validation Pipeline

```
@Component
public class ConfigurationValidationPipeline {

    public ValidationReport validateAllConfigurations() {
        YamlMetadataValidator validator = new YamlMetadataValidator();

        // 1. Discover all YAML files
        List<String> yamlFiles = discoverYamlFiles();
```

```
        // 2. Validate metadata and structure
        YamlValidationSummary validationSummary = validator.validateFiles(yamlFiles);

        // 3. Analyze dependencies
        YamlDependencyAnalyzer dependencyAnalyzer = new YamlDependencyAnalyzer();
        Map<String, YamlDependencyGraph> dependencyGraphs = new HashMap<>();

        for (String file : yamlFiles) {
            if (isRootConfigurationFile(file)) {
                YamlDependencyGraph graph = dependencyAnalyzer.analyzeYamlDependencies(file);
                dependencyGraphs.put(file, graph);
            }
        }

        // 4. Generate comprehensive report
        return ValidationReport.builder()
            .validationSummary(validationSummary)
            .dependencyGraphs(dependencyGraphs)
            .recommendations(generateRecommendations(validationSummary, dependencyGraphs))
            .build();
    }
}
```

3. Dynamic Scenario Registration

```
@RestController
@RequestMapping("/api/scenarios")
public class ScenarioManagementController {

    @PostMapping("/register")
    public ResponseEntity<ScenarioRegistrationResult> registerScenario(
            @RequestBody ScenarioRegistrationRequest request) {

        // 1. Validate scenario configuration
        YamlValidationResult validation = validateScenarioConfiguration(request);
        if (!validation.isValid()) {
            return ResponseEntity.badRequest()
                .body(ScenarioRegistrationResult.failure(validation.getErrors()));
        }

        // 2. Register scenario
        scenarioService.registerScenario(request.getScenarioConfiguration());

        // 3. Update registry
        registryManager.updateRegistry(request.getRegistryEntry());

        return ResponseEntity.ok(ScenarioRegistrationResult.success());
    }
}
```

Performance Considerations

Caching Strategy

```
@Configuration
@EnableCaching
public class ScenarioCacheConfiguration {

    @Bean
```

```
    @Primary
    public CacheManager scenarioCacheManager() {
        CaffeineCacheManager cacheManager = new CaffeineCacheManager();
        cacheManager.setCaffeine(Caffeine.newBuilder()
            .maximumSize(1000)
            .expireAfterWrite(Duration.ofHours(1))
            .recordStats());
        return cacheManager;
    }
}

@Service
public class CachedScenarioService {

    @Cacheable(value = "scenarios", key = "#scenarioId")
    public ScenarioConfiguration getScenario(String scenarioId) {
        return loadScenarioFromFile(scenarioId);
    }

    @Cacheable(value = "dataTypeScenarios", key = "#dataType.name")
    public ScenarioConfiguration getScenarioForDataType(Class<?> dataType) {
        return findScenarioForDataType(dataType);
    }
}
```

Lazy Loading and Preloading

```
@Component
public class ScenarioPreloader {

    @EventListener(ApplicationReadyEvent.class)
    public void preloadCriticalScenarios() {
        List<String> criticalScenarios = configurationProperties.getCriticalScenarios();

        criticalScenarios.parallelStream().forEach(scenarioId -> {
            try {
                scenarioService.getScenario(scenarioId);
                logger.info("Preloaded critical scenario: {}", scenarioId);
            } catch (Exception e) {
                logger.error("Failed to preload scenario: {}", scenarioId, e);
            }
        });
    }
}
```

External Data Source Integration

Overview

One of APEX's most powerful features is its ability to connect to external data sources seamlessly. Think of this as giving your rules access to live data from databases, web APIs, files, and other systems - all through a simple, unified interface.

What does this mean for you? Instead of having to write custom code to fetch data from different systems, APEX handles all the complexity of connecting to various data sources. Your rules can simply reference data like `#userDatabase.getUser(123)` or `#priceAPI.getCurrentPrice('AAPL')`, and APEX takes care of the rest.

Enterprise-grade features included:

- **Connection pooling**: Efficiently manages database connections for high performance
- **Health monitoring**: Automatically checks if data sources are available and healthy
- **Caching**: Stores frequently accessed data in memory for faster access
- **Automatic failover**: If one data source fails, APEX can automatically try backup sources
- **Load balancing**: Distributes requests across multiple data sources for better performance

Supported Data Source Types

APEX supports five main types of data sources, each designed for different use cases. You can mix and match these types based on your needs:

1. Database Sources

Connect to relational databases where your business data lives.

- **Supported databases**: PostgreSQL, MySQL, Oracle, SQL Server, H2
- **Best for**: Customer data, transaction records, master data, audit logs
- **Features**: Connection pooling, prepared statements, transaction support
- **Example use**: Looking up customer credit scores, validating account balances

2. REST API Sources

Integrate with web services and external APIs.

- **Protocols**: HTTP/HTTPS with various authentication methods (Bearer tokens, API keys, OAuth2)
- **Best for**: Real-time data, third-party services, microservices integration
- **Features**: Circuit breakers, retry logic, timeout handling
- **Example use**: Getting current exchange rates, validating addresses with postal services

3. File System Sources

Process data from files on your local system or network drives.

- **Supported formats**: CSV, JSON, XML, fixed-width text files, plain text
- **Best for**: Batch data processing, configuration files, data imports
- **Features**: File watching, automatic parsing, encoding support
- **Example use**: Processing daily trade files, loading reference data from CSV files

4. Cache Sources

High-speed in-memory data storage for frequently accessed information.

- **Features**: LRU (Least Recently Used) eviction, TTL (Time To Live) support
- **Best for**: Frequently accessed lookup data, computed results, session data
- **Performance**: Microsecond access times for cached data
- **Example use**: Caching currency conversion rates, storing user preferences

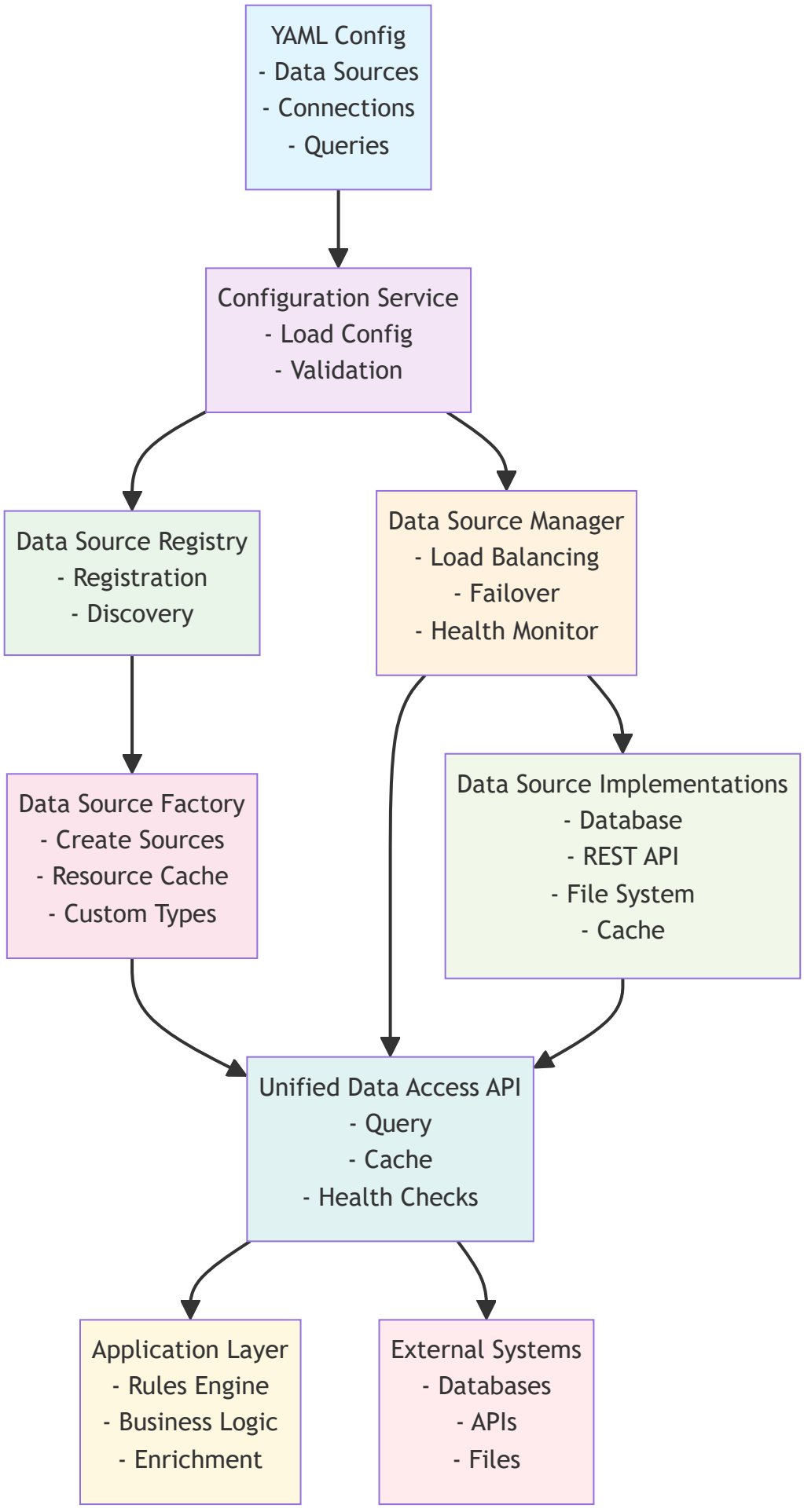
5. Custom Sources

Build your own data source integrations for specialized needs.

- **Architecture**: Extensible plugin system with well-defined interfaces
- **Best for**: Legacy systems, proprietary protocols, specialized data formats
- **Features**: Full control over data retrieval and caching logic

- **Example use**: Connecting to mainframe systems, custom message queues

External Data Source Architecture


YAML Config
- Data Sources
- Connections
- Queries
Configuration Service
- Load Config
- Validation
Data Source Registry
- Registration
- Discovery
Data Source Manager
- Load Balancing
- Failover
- Health Monitor
Data Source Factory
- Create Sources
- Resource Cache
- Custom Types
Data Source Implementations
- Database
- REST API
- File System
- Cache
Unified Data Access API
- Query
- Cache
- Health Checks
Application Layer
- Rules Engine
- Business Logic
- Enrichment
External Systems
- Databases
- APIs
- Files

Key External Data Source Components

Understanding the architecture helps you work more effectively with APEX's data integration features. The system is organized into three main layers:

Core Interfaces (The Foundation)

These are the basic building blocks that define how data sources work:

- `ExternalDataSource` - The main contract that all data sources must implement. Think of this as the "blueprint" that ensures all data sources work the same way.
- `DataSourceConfiguration` - Holds all the settings for a data source (like connection strings, timeouts, etc.)
- `DataSourceType` - An enumeration that identifies what kind of data source you're working with (DATABASE, REST_API, FILE_SYSTEM, etc.)
- `ConnectionStatus` - Tracks whether a data source is healthy and available
- `DataSourceMetrics` - Collects performance data (response times, success rates, error counts)

Management Components (The Orchestrators)

These components coordinate and manage multiple data sources:

- `DataSourceConfigurationService` - The high-level service that loads and manages your YAML configurations. This is usually your main entry point.
- `DataSourceManager` - The smart coordinator that handles load balancing between multiple data sources and automatic failover when sources go down.
- `DataServiceManager` - A simpler manager for programmatic configuration, great for testing and demos.
- `DemoDataServiceManager` - Pre-configured with sample data sources for quick demos and testing.
- `DataSourceRegistry` - The central directory that keeps track of all available data sources and monitors their health.
- `DataSourceFactory` - The factory that creates and configures data source instances, handling resource management and caching.

Implementation Classes (The Workers)

These are the actual implementations that do the work of connecting to different systems:

- `DatabaseDataSource` - Handles database connections with connection pooling, prepared statements, and transaction management.
- `RestApiDataSource` - Manages HTTP/HTTPS API calls with circuit breakers, retries, and authentication.
- `FileSystemDataSource` - Processes files with format-specific readers and file system monitoring.
- `CacheDataSource` - Provides high-speed in-memory caching with intelligent eviction policies.

Programmatic Data Service Configuration

While YAML configuration is great for production systems, sometimes you need to set up data sources programmatically - especially for testing, demos, or when building dynamic systems. APEX provides a simple programmatic API through the `DataServiceManager`.

When to use programmatic configuration:

- Unit testing and integration testing
- Demo applications and proof-of-concepts

- Dynamic systems where data sources are determined at runtime
- Embedded applications where YAML files aren't practical

Basic setup example:

```
// Create a data service manager
DataServiceManager dataManager = new DataServiceManager();

// Load individual data sources one at a time
dataManager.loadDataSource(new MockDataSource("ProductsDataSource", "products"));
dataManager.loadDataSource(new CustomDataSource("CustomerDataSource", "customer"));

// Or load multiple data sources at once for efficiency
dataManager.loadDataSources(
    new MockDataSource("InventoryDataSource", "inventory"),
    new MockDataSource("LookupServicesDataSource", "lookupServices")
);

// Request data for use in your rules
List<Product> products = dataManager.requestData("products");
Customer customer = dataManager.requestData("customer");
```

Line-by-line explanation:

- **Line 2**: Create a new `DataServiceManager` instance - this is your central coordinator for managing all data sources
- **Line 5**: Load a single data source with a descriptive name ("ProductsDataSource") and data type identifier ("products")
- **Line 6**: Load another data source for customer data - each data source has a unique name and type
- **Line 9-12**: Alternative approach to load multiple data sources in a single method call for better performance
- **Line 10**: Create an inventory data source that will provide inventory-related data
- **Line 11**: Create a lookup services data source for reference data and enrichment
- **Line 15**: Request typed data from the "products" data source - returns a strongly-typed List
- **Line 16**: Request customer data - the generic method automatically handles type casting based on the data source configuration

What's happening here:

1. **Create the manager**: `DataServiceManager` is your central coordinator
2. **Load data sources**: Register your data sources with descriptive names
3. **Request data**: Use simple method calls to get data for your rules
4. **Type safety**: The generic `requestData()` method returns properly typed data

Demo Data Service Manager

The `DemoDataServiceManager` is your best friend for getting started quickly, running demos, or testing your rules. It comes pre-loaded with realistic sample data that covers common business scenarios.

Perfect for:

- Learning APEX without setting up real databases
- Creating demos and proof-of-concepts
- Unit testing your rules with consistent test data
- Prototyping new rule logic before connecting to production systems

```
// One line to get a fully configured demo environment
DemoDataServiceManager demoManager = new DemoDataServiceManager();
demoManager.initializeWithMockData();

// Now you have access to realistic sample data for various scenarios
```

Line-by-line explanation:

- **Line 2**: Create a new `DemoDataServiceManager` instance - this extends the basic DataServiceManager with pre-configured demo data
- **Line 3**: Initialize the manager with realistic mock data for all supported data types - this loads sample products, customers, inventory, etc.
- **Line 5**: Comment indicating that the manager is now ready to provide data for testing and demonstration purposes

Available data types and what they contain:

- `products` : List of financial products (bonds, equities, derivatives) with realistic attributes
- `inventory` : Available inventory items with quantities, prices, and categories
- `customer` : Sample customer data with demographics, account information, and transaction history
- `templateCustomer` : A template customer object perfect for testing validation rules
- `lookupServices` : Pre-configured lookup service data for enrichment testing
- `sourceRecords` : Trade records that can be used for matching and validation scenarios
- `matchingRecords` : Dynamic matching results that demonstrate complex rule logic
- `nonMatchingRecords` : Records that fail matching criteria, useful for testing negative cases

Why use the demo manager?

- **No setup required**: Works immediately without any configuration
- **Realistic data**: Sample data reflects real-world business scenarios
- **Comprehensive coverage**: Includes data for most common rule testing scenarios
- **Consistent results**: Same data every time, making tests predictable

Custom Data Source Implementation

Create custom data sources for specific business needs:

```
public class DatabaseDataSource implements DataSource {
    private final String name;
    private final String dataType;
    private final DatabaseConnection connection;

    public DatabaseDataSource(String name, String dataType, DatabaseConnection connection) {
        this.name = name;
        this.dataType = dataType;
        this.connection = connection;
    }

    @Override
    public <T> T getData(String dataType, Object... parameters) {
        if (!supportsDataType(dataType)) {
            return null;
        }

        // Implement database query logic
        String query = buildQuery(dataType, parameters);
        return connection.executeQuery(query);
```

```
    }

    @Override
    public boolean supportsDataType(String dataType) {
        return this.dataType.equals(dataType);
    }

    @Override
    public String getName() {
        return name;
    }

    @Override
    public String getDataType() {
        return dataType;
    }
}
```

Line-by-line explanation:

- **Line 1**: Class declaration implementing the `DataSource` interface - this ensures all data sources have a consistent API
- **Line 2**: Private field to store the human-readable name of this data source (e.g., "Customer Database")
- **Line 3**: Private field to store the data type this source provides (e.g., "customer", "product")
- **Line 4**: Private field to hold the database connection object for executing queries
- **Line 6**: Constructor that initializes all required fields - name for identification, dataType for routing, connection for data access
- **Line 7-9**: Store the provided parameters in instance fields for later use
- **Line 12**: Generic method that retrieves data of type T based on the requested dataType and optional parameters
- **Line 13-15**: Guard clause that returns null if this data source doesn't support the requested data type
- **Line 18**: Build a SQL query string based on the data type and parameters (implementation would be specific to your database schema)
- **Line 19**: Execute the query using the database connection and return the typed result
- **Line 23**: Method to check if this data source can provide data for a specific type
- **Line 24**: Simple string comparison to determine if this source handles the requested data type
- **Line 27-29**: Getter method to return the name of this data source
- **Line 31-33**: Getter method to return the data type this source provides

Integration with Rules Engine

Both YAML-based and programmatic data source configurations integrate seamlessly with the rules engine:

```
// YAML-based external data source integration
DataSourceConfigurationService configService = DataSourceConfigurationService.getInstance();
YamlRuleConfiguration yamlConfig = loadConfiguration("data-sources.yaml");
configService.initialize(yamlConfig);

// Use in rules
ExternalDataSource userDb = configService.getDataSource("user-database");
Map<String, Object> params = Map.of("id", userId);
Object user = userDb.queryForObject("getUserById", params);

// Use with load balancing
DataSourceManager manager = configService.getDataSourceManager();
List<Object> results = manager.queryWithFailover(DataSourceType.DATABASE, "getAllUsers", Collections.emptyMap());

// Programmatic data service integration
DemoDataServiceManager dataManager = new DemoDataServiceManager();
dataManager.initializeWithMockData();
```

```
// Create evaluation context with data
Map<String, Object> facts = new HashMap<>();
facts.put("products", dataManager.requestData("products"));
facts.put("customer", dataManager.requestData("customer"));
facts.put("inventory", dataManager.requestData("inventory"));

// Evaluate rules with data context
RulesEngine engine = new RulesEngine(configuration);
RuleResult result = engine.evaluate(facts);
```

Line-by-line explanation:

- **Line 2**: Get the singleton instance of the configuration service that manages YAML-based data source configurations
- **Line 3**: Load the YAML configuration file containing data source definitions (databases, APIs, files, etc.)
- **Line 4**: Initialize the configuration service with the loaded YAML configuration
- **Line 7**: Retrieve a specific data source by name from the configuration service
- **Line 8**: Create a parameter map for the database query - in this case, passing a user ID
- **Line 9**: Execute a named query ("getUserById") with parameters and get the result object
- **Line 12**: Get the data source manager that handles load balancing and failover between multiple data sources
- **Line 13**: Execute a query with automatic failover - if one database fails, it tries backup databases
- **Line 16**: Create a demo data service manager for testing and demonstration purposes
- **Line 17**: Initialize with realistic mock data for all supported data types
- **Line 20**: Create a facts map that will hold all data available to the rules engine
- **Line 21**: Add product data to the facts map using the key "products"
- **Line 22**: Add customer data to the facts map using the key "customer"
- **Line 23**: Add inventory data to the facts map using the key "inventory"
- **Line 26**: Create a rules engine instance with the provided configuration
- **Line 27**: Evaluate all rules against the facts map and return the combined result

Bootstrap Demo Architecture

Overview

APEX includes four comprehensive bootstrap demonstrations that showcase complete end-to-end scenarios with real-world financial data and infrastructure. These demos are architected as self-contained applications that demonstrate different aspects of APEX capabilities while providing practical learning experiences.

Bootstrap Demo Architecture Pattern

All bootstrap demos follow a consistent architectural pattern that includes infrastructure setup, configuration loading, scenario execution, and results analysis. Each demo is designed to be completely self-contained with automatic environment detection and graceful fallback mechanisms.

Key Architectural Principles:

- **Self-Contained Execution**: All dependencies and data sources included
- **Progressive Complexity**: Each scenario builds on previous concepts
- **Real-World Data**: Authentic financial services data and conventions
- **Performance Monitoring**: Comprehensive metrics and benchmarking
- **Educational Focus**: Clear documentation and learning objectives

Individual Bootstrap Architectures

1. Custody Auto-Repair Bootstrap

Technical Focus: Weighted rule-based decision making with sub-100ms processing

Architecture Highlights:

- PostgreSQL database with comprehensive settlement tables
- Asian markets data (Japan, Hong Kong, Singapore)
- 5 progressive scenarios from premium clients to exception handling
- Weighted scoring algorithm across client, market, and instrument factors
- Complete audit trail for regulatory compliance

2. Commodity Swap Validation Bootstrap

Technical Focus: Progressive API complexity and multi-layered validation

Architecture Highlights:

- 5 comprehensive database tables with realistic market data
- Energy (WTI, Brent), Metals (Gold, Silver), Agricultural (Corn) markets
- 6 learning scenarios demonstrating API progression
- Ultra-simple → Template-based → Advanced configuration patterns
- Performance monitoring with sub-100ms targets

3. OTC Options Bootstrap Demo

Technical Focus: Multiple data integration methods and Spring Boot integration

Architecture Highlights:

- Three different data lookup approaches (inline, database, external files)
- PostgreSQL counterparty data with external YAML datasets
- Major commodity coverage (Natural Gas, Oil, Metals, Agricultural)
- Complete Spring Boot application with dependency injection
- Realistic OTC Options structures and market conventions

4. Scenario-Based Processing Demo

Technical Focus: Automatic data type routing and centralized configuration

Architecture Highlights:

- Intelligent data type detection and routing
- Centralized scenario registry with lightweight configuration
- Support for OTC Options, Commodity Swaps, Settlement Instructions
- Graceful degradation for unknown data types
- Flexible routing with multiple scenarios per data type

Common Technical Patterns

All bootstrap demos implement these common patterns:

Infrastructure Setup Pattern:

```
// Phase 1: Environment Detection and Setup
detectEnvironment();
createDatabaseTables();
generateSampleData();
loadConfiguration();
validateSetup();

// Phase 2: APEX Integration
loadYamlConfiguration();
createRulesEngine();
setupEnrichmentServices();
configurePerformanceMonitoring();

// Phase 3: Scenario Execution
executeScenarios();
collectMetrics();
generateReports();
```

Performance Monitoring Pattern:

- Start/end timing for all operations
- Sub-100ms processing targets
- Comprehensive metrics collection
- Success rate analysis
- Throughput calculations

Configuration Loading Pattern:

- YAML configuration validation
- Metadata verification
- Dependency checking
- Business rule validation

Integration with APEX Core

Bootstrap demos demonstrate deep integration with APEX core components including RulesEngineService, EnrichmentService, ExpressionEvaluatorService, and comprehensive performance monitoring systems.

Architecture Overview

System Architecture

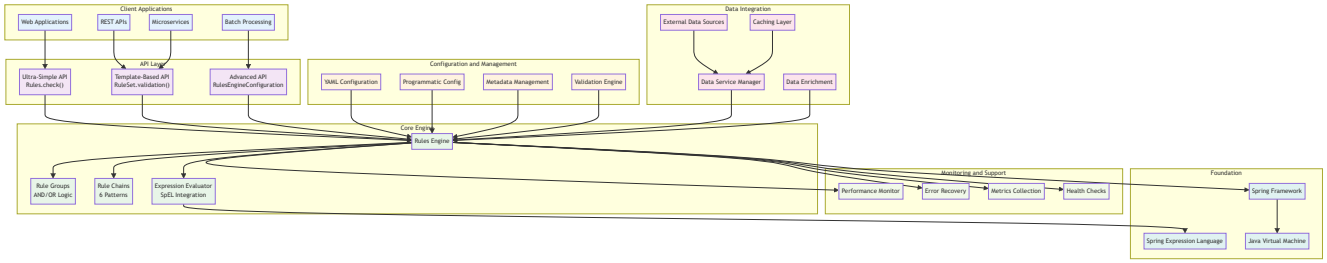
Understanding APEX's architecture helps you make better decisions about how to use it in your systems. APEX is built with a layered architecture that balances simplicity for basic use cases with power for complex enterprise scenarios.

The key principle: Start simple and add complexity only when you need it.

The architecture is designed around several core ideas:

- **Progressive complexity**: Three API layers from ultra-simple to advanced
- **Separation of concerns**: Each layer has a specific responsibility

- **Enterprise-ready**: Built-in monitoring, error handling, and performance optimization
- **Extensible**: Plugin architecture for custom data sources and rule patterns

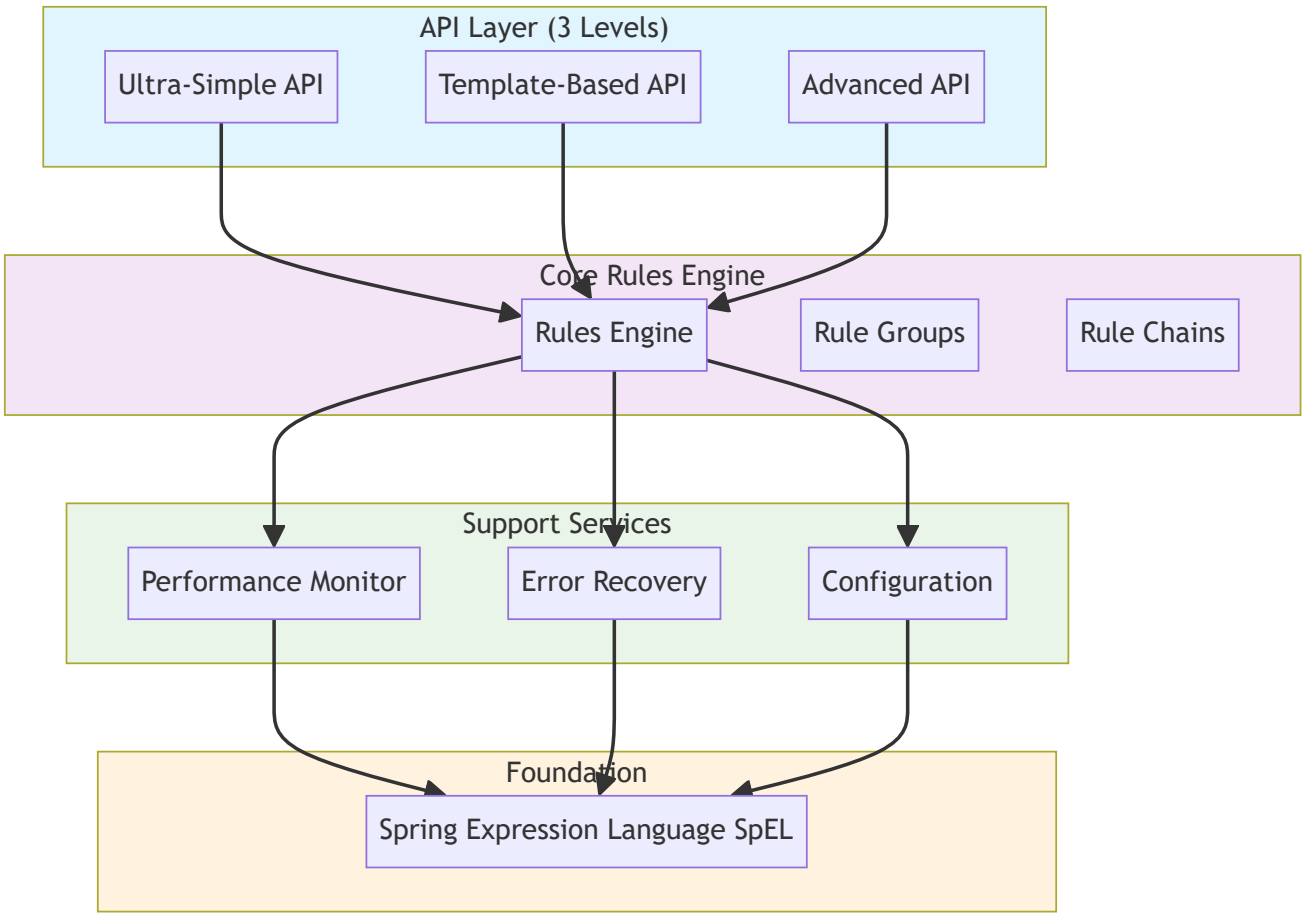


Core Components

APEX's architecture is like a well-organized building with different floors serving different purposes. Each layer builds on the one below it, providing more functionality while maintaining the simplicity of the lower layers.

Think of it this way:

- **Foundation floor**: Spring Expression Language (SpEL) provides the basic rule evaluation engine
- **Core floor**: The Rules Engine handles rule execution, groups, and chains
- **Service floor**: Support services provide monitoring, error handling, and configuration
- **API floor**: Three different APIs provide different levels of complexity and power



Key Classes and Interfaces

Here are the main classes you'll work with when using APEX. Don't worry about memorizing all of these - you'll naturally learn the ones you need as you use the system.

Core Engine Classes (The Heart of APEX)

These classes handle the actual rule execution:

- `RulesEngine` - The main engine that executes your rules. This is like the conductor of an orchestra, coordinating all the other components.
- `RulesEngineConfiguration` - Holds all your configuration settings. Think of this as the blueprint that tells the engine how to behave.
- `Rule` - Represents a single business rule. Each rule has a condition, message, and metadata.
- `RuleGroup` - A collection of related rules that work together with AND/OR logic (like "all validation rules must pass").
- `RuleGroupBuilder` - A helper class that makes it easy to create rule groups programmatically.
- `RuleResult` - Contains the results of rule execution, including whether rules passed, failed, or had errors.

API Layer Classes (Your Entry Points)

These classes provide different ways to interact with APEX:

- `Rules` - Static utility methods for quick, one-line rule evaluations. Perfect for simple scenarios.
- `RuleSet` - Template-based rule creation with pre-built patterns for common scenarios like validation.
- `ValidationBuilder` - A fluent interface for building validation rule sets in a readable way.
- `RulesService` - An instance-based service that works well with dependency injection frameworks like Spring.

Performance Monitoring (Keeping Things Fast)

These classes help you understand and optimize rule performance:

- `RulePerformanceMonitor` - The central service that tracks how fast your rules execute.
- `RulePerformanceMetrics` - Detailed metrics for individual rules (execution time, success rate, etc.).
- `PerformanceSnapshot` - A point-in-time view of overall system performance.
- `PerformanceAnalyzer` - Analyzes performance data and provides insights and recommendations.

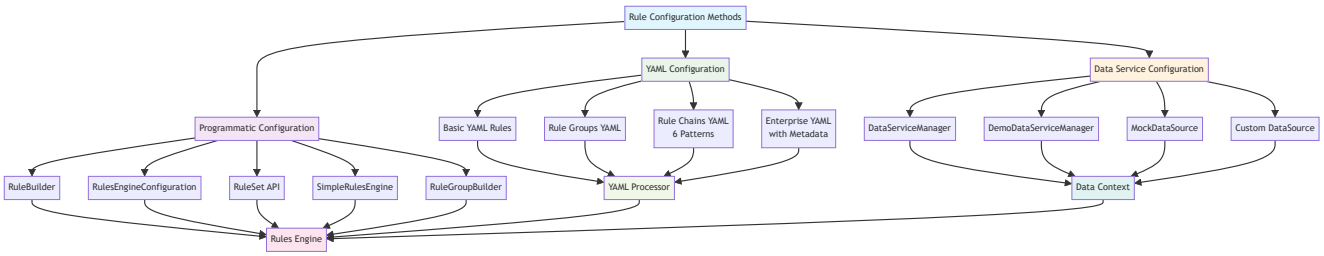
Error Handling (When Things Go Wrong)

These classes help APEX handle errors gracefully:

- `ErrorRecoveryService` - Manages how APEX responds to errors (continue processing, stop, retry, etc.).
- `RuleEngineException` - The base class for all APEX-specific exceptions, with detailed error information.
- `ErrorRecoveryStrategy` - Defines different strategies for handling errors (like CONTINUE_ON_ERROR or FAIL_FAST).

Rule Configuration Methods

The APEX Rules Engine supports multiple configuration approaches to meet different use cases and complexity requirements:



Implementation Patterns

APEX provides three distinct implementation patterns, each designed for different use cases and complexity levels. Think of these as different tools in your toolbox - use the simplest one that meets your needs.

1. Ultra-Simple API Pattern (For Quick Tasks)

When to use: One-off rule evaluations, simple validation checks, prototyping **Complexity**: Very Low **Setup time**: Seconds

This pattern is perfect when you just need to check one rule quickly:

```
// Static utility methods for instant rule evaluation
public class Rules {
    // Check if a rule passes (returns true/false)
    public static boolean check(String expression, Object data) {
        return RulesEngine.getDefault().evaluate(expression, data).isSuccess();
    }

    // Calculate a value using a rule expression
    public static <T> T calculate(String expression, Object data, Class<T> returnType) {
        return RulesEngine.getDefault().calculate(expression, data, returnType);
    }
}

// Usage examples:
boolean isEligible = Rules.check("#age >= 18", customer);
BigDecimal discount = Rules.calculate("#amount * 0.1", order, BigDecimal.class);
```

Line-by-line explanation:

- **Line 2**: Static utility class that provides the simplest possible API for rule evaluation
- **Line 4**: Static method that evaluates a rule expression and returns true if the rule passes, false otherwise
- **Line 5**: Get the default rules engine instance, evaluate the expression against the data, and check if the result is successful
- **Line 9**: Generic static method that evaluates an expression and returns a calculated value of the specified type
- **Line 10**: Use the default rules engine to calculate a value from the expression, with automatic type conversion
- **Line 14**: Example usage - check if a customer is eligible (age >= 18) using the customer object
- **Line 15**: Example usage - calculate a 10% discount on an order amount and return it as a BigDecimal

Benefits: No setup required, works immediately, perfect for simple scenarios **Limitations**: No configuration options, no performance monitoring, no complex rule relationships

2. Template-Based API Pattern (For Common Scenarios)

When to use: Validation scenarios, common business patterns, structured rule sets **Complexity**: Low to Medium **Setup time**: Minutes

This pattern provides pre-built templates for common use cases:

```
// Fluent builder for common validation scenarios
public class RuleSet {
    public static ValidationBuilder validation() {
        return new ValidationBuilder();
    }

    public static class ValidationBuilder {
        // Pre-built validation patterns
        public ValidationBuilder ageCheck(int minimumAge) {
            return addRule("#data.age >= " + minimumAge);
        }

        public ValidationBuilder emailRequired() {
            return addRule("#data.email != null && #data.email.contains('@')");
        }

        public ValidationBuilder balanceMinimum(BigDecimal minimum) {
            return addRule("#data.balance >= " + minimum);
        }

        public RulesEngine build() {
            return new RulesEngine(configuration);
        }
    }
}

// Usage example:
RulesEngine validator = RuleSet.validation()
    .ageCheck(18)
    .emailRequired()
    .balanceMinimum(new BigDecimal("1000"))
    .build();
```

Line-by-line explanation:

- **Line 2**: Main RuleSet class that provides factory methods for creating different types of rule builders
- **Line 3**: Static factory method that returns a new ValidationBuilder instance for creating validation rule sets
- **Line 7**: Inner class that implements the fluent builder pattern for validation rules
- **Line 9**: Method that adds an age validation rule with a configurable minimum age requirement
- **Line 10**: Create a SpEL expression that checks if the data's age field is greater than or equal to the minimum
- **Line 13**: Method that adds an email validation rule requiring a non-null email with an @ symbol
- **Line 14**: Create a SpEL expression that validates email presence and basic format
- **Line 17**: Method that adds a balance validation rule with a configurable minimum balance
- **Line 18**: Create a SpEL expression that checks if the balance meets the minimum requirement
- **Line 21**: Build method that creates and returns a configured RulesEngine instance
- **Line 22**: Construct the rules engine with the accumulated configuration from the builder
- **Line 27**: Example usage showing the fluent API in action
- **Line 28**: Add an age check requiring customers to be at least 18 years old
- **Line 29**: Add an email validation requirement
- **Line 30**: Add a minimum balance requirement of 1000 (currency units)
- **Line 31**: Build the final rules engine with all validation rules configured

Benefits: Pre-built patterns, readable code, good for common scenarios **Limitations**: Limited to built-in templates, less flexibility than advanced configuration

3. Advanced Configuration Pattern (For Complex Systems)

When to use: Enterprise applications, complex rule relationships, performance-critical systems **Complexity**: Medium to High **Setup time**: Hours to days (depending on complexity)

This pattern gives you full control over every aspect of the rules engine:

```
// Full configuration-based approach with all enterprise features
RulesEngineConfiguration config = new RulesEngineConfiguration.Builder()
    .withPerformanceMonitoring(true)           // Track rule execution performance
    .withErrorRecovery(ErrorRecoveryStrategy.CONTINUE_ON_ERROR)  // Handle errors gracefully
    .withCaching(true)                         // Cache rule results for performance
    .withMaxExecutionTime(Duration.ofSeconds(30))  // Prevent runaway rules
    .addRule(rule1)                           // Add individual rules
    .addRule(rule2)
    .addRuleGroup(validationGroup)            // Add rule groups
    .addEnrichment(enrichment1)               // Add data enrichments
    .addDataSource(databaseSource)            // Add external data sources
    .build();

RulesEngine engine = new RulesEngine(config);
```

Line-by-line explanation:

- **Line 2**: Create a new configuration builder using the Builder pattern for complex configuration setup
- **Line 3**: Enable performance monitoring to track execution times, memory usage, and other metrics
- **Line 4**: Set error recovery strategy to continue processing other rules even if one rule fails
- **Line 5**: Enable caching to store rule results and improve performance for repeated evaluations
- **Line 6**: Set a maximum execution time of 30 seconds to prevent infinite loops or runaway rules
- **Line 7**: Add an individual rule (rule1) to the configuration
- **Line 8**: Add another individual rule (rule2) to the configuration
- **Line 9**: Add a rule group that contains multiple related rules with AND/OR logic
- **Line 10**: Add a data enrichment that will enhance input data with additional information
- **Line 11**: Add an external data source (like a database) for rules to access external data
- **Line 12**: Build the final configuration object with all the specified settings
- **Line 14**: Create a new rules engine instance using the comprehensive configuration

Benefits: Full control, enterprise features, maximum flexibility, performance optimization **Limitations**: More complex setup, requires deeper understanding of APEX concepts

Choosing the Right Pattern

Pattern	Use When	Setup Time	Flexibility	Enterprise Features
Ultra-Simple	Quick checks, prototypes	Seconds	Low	None
Template-Based	Common scenarios, validation	Minutes	Medium	Basic
Advanced	Enterprise systems, complex rules	Hours+	High	Full

Recommendation: Start with the Ultra-Simple pattern for learning and prototyping, move to Template-Based for structured applications, and use Advanced configuration for production enterprise systems.

Rule Groups Architecture

Overview

Rule Groups are one of APEX's most powerful organizational features. Think of them as smart containers that hold related rules and execute them according to business logic patterns.

What are Rule Groups? Instead of executing rules individually, Rule Groups let you organize related rules and control how they work together. For example, you might have a "Customer Validation" group where ALL rules must pass, or an "Eligibility Check" group where ANY rule can pass.

Why use Rule Groups?

- **Logical organization**: Group related rules together (like all validation rules, all eligibility rules)
- **Business logic control**: Use AND logic (all must pass) or OR logic (any can pass)
- **Performance optimization**: Stop execution early when the outcome is determined
- **Better maintainability**: Organize rules by business domain or functional area
- **Clearer results**: Get group-level results in addition to individual rule results

Two main patterns:

- **AND Groups**: All rules must pass for the group to succeed (like validation scenarios)
- **OR Groups**: Any rule can pass for the group to succeed (like eligibility scenarios)

Core Components

RuleGroup Class

```
public class RuleGroup implements RuleBase {
    private final String id;
    private final Set<Category> categories;
    private final String name;
    private final String description;
    private final int priority;
    private final Map<Integer, Rule> rulesBySequence;
    private final boolean isAndOperator;

    // Constructor for single category
    public RuleGroup(String id, String category, String name, String description,
                     int priority, boolean isAndOperator)

    // Constructor for multiple categories
    public RuleGroup(String id, Set<Category> categories, String name, String description,
                     int priority, boolean isAndOperator)

    // Add rule with sequence number
    public void addRule(Rule rule, int sequenceNumber)

    // Execution method
    public RuleResult evaluate(Map<String, Object> facts)
}
```

Line-by-line explanation:

- **Line 1**: Class declaration implementing RuleBase interface, making rule groups compatible with individual rules
- **Line 2**: Unique identifier for this rule group (e.g., "customer-validation-group")
- **Line 3**: Set of categories this rule group belongs to (e.g., "validation", "compliance")
- **Line 4**: Human-readable name for the rule group (e.g., "Customer Validation Rules")
- **Line 5**: Detailed description explaining the purpose of this rule group
- **Line 6**: Priority level for execution order when multiple rule groups exist
- **Line 7**: Map storing rules by their sequence number for ordered execution
- **Line 8**: Boolean flag determining if this is an AND group (all must pass) or OR group (any can pass)
- **Line 11-12**: Constructor for creating a rule group with a single category
- **Line 15-16**: Constructor for creating a rule group with multiple categories
- **Line 19**: Method to add a rule to the group with a specific sequence number for execution order
- **Line 22**: Main execution method that evaluates all rules in the group according to the AND/OR logic

RuleGroupBuilder

```
public class RuleGroupBuilder {
    private String id;
    private Set<Category> categories = new HashSet<>();
    private String name;
    private String description;
    private int priority = 100;
    private boolean isAndOperator = true;

    public RuleGroupBuilder withId(String id)
    public RuleGroupBuilder withName(String name)
    public RuleGroupBuilder withDescription(String description)
    public RuleGroupBuilder withCategory(Category category)
    public RuleGroupBuilder withCategoryNames(Set<String> categoryNames)
    public RuleGroupBuilder withPriority(int priority)
    public RuleGroupBuilder withAndOperator()
    public RuleGroupBuilder withOrOperator()
    public RuleGroup build()
}
```

Line-by-line explanation:

- **Line 1**: Builder class that implements the Builder pattern for creating RuleGroup instances
- **Line 2**: Private field to store the unique identifier for the rule group being built
- **Line 3**: Private field to store the set of categories, initialized as an empty HashSet
- **Line 4**: Private field to store the human-readable name of the rule group
- **Line 5**: Private field to store the detailed description of the rule group's purpose
- **Line 6**: Private field to store the priority level, defaulting to 100 (medium priority)
- **Line 7**: Private field to store the operator type, defaulting to true (AND operator)
- **Line 9**: Fluent method to set the rule group ID and return the builder for method chaining
- **Line 10**: Fluent method to set the rule group name and return the builder
- **Line 11**: Fluent method to set the rule group description and return the builder
- **Line 12**: Fluent method to add a single category to the rule group and return the builder
- **Line 13**: Fluent method to add multiple categories by name and return the builder
- **Line 14**: Fluent method to set the priority level and return the builder
- **Line 15**: Fluent method to configure the group as an AND group (all rules must pass)
- **Line 16**: Fluent method to configure the group as an OR group (any rule can pass)
- **Line 17**: Final method that creates and returns the configured RuleGroup instance

Configuration Methods

Programmatic Configuration

```
// Method 1: Using RuleGroupBuilder
RuleGroup group = new RuleGroupBuilder()
    .withId("validation-group")
    .withName("Customer Validation")
    .withDescription("Complete customer validation checks")
    .withCategory("validation")
    .withPriority(10)
    .withAndOperator()
    .build();

// Method 2: Using RulesEngineConfiguration
RulesEngineConfiguration config = new RulesEngineConfiguration();

// AND group
RuleGroup andGroup = config.createRuleGroupWithAnd(
    "and-group", category, "AND Group", "All must pass", 10);

// OR group
RuleGroup orGroup = config.createRuleGroupWithOr(
    "or-group", category, "OR Group", "Any can pass", 20);

// Multi-category group
Set<String> categories = Set.of("validation", "compliance");
RuleGroup multiGroup = config.createRuleGroupWithAnd(
    "multi-group", categories, "Multi Group", "Multiple categories", 30);

// Register groups
config.registerRuleGroup(andGroup);
config.registerRuleGroup(orGroup);
```

Line-by-line explanation:

- **Line 2**: Create a new RuleGroupBuilder instance using the fluent builder pattern
- **Line 3**: Set the unique identifier for this rule group to "validation-group"
- **Line 4**: Set the human-readable name to "Customer Validation"
- **Line 5**: Set a detailed description explaining the purpose of this rule group
- **Line 6**: Assign this rule group to the "validation" category for organization
- **Line 7**: Set the priority to 10 (high priority - will execute before lower priority groups)
- **Line 8**: Configure this as an AND group, meaning all rules must pass for the group to succeed
- **Line 9**: Build and return the configured RuleGroup instance
- **Line 12**: Create a new RulesEngineConfiguration instance for alternative configuration approach
- **Line 15-16**: Create an AND rule group using the configuration's factory method
- **Line 19-20**: Create an OR rule group using the configuration's factory method
- **Line 23**: Create a set of category names for a multi-category rule group
- **Line 24-25**: Create a rule group that belongs to multiple categories (validation and compliance)
- **Line 28**: Register the AND group with the configuration so it will be included in rule execution
- **Line 29**: Register the OR group with the configuration

YAML Configuration

```
rule-groups:
  - id: "customer-validation"
```

```yaml
    name: "Customer Validation Rules"
    description: "Complete customer validation rule set"
    category: "validation"
    priority: 10
    enabled: true
    stop-on-first-failure: false
    parallel-execution: false
    rule-ids:
      - "age-validation"
      - "email-validation"
      - "income-validation"
    metadata:
      owner: "Customer Team"
      domain: "Customer Management"

  - id: "eligibility-check"
    name: "Customer Eligibility Check"
    description: "Customer meets eligibility criteria"
    category: "eligibility"
    categories: ["eligibility", "customer"]  # Multiple categories
    priority: 20
    enabled: true
    stop-on-first-failure: false
    parallel-execution: true
    rule-references:
      - rule-id: "premium-customer"
        sequence: 1
        enabled: true
        override-priority: 5
      - rule-id: "long-term-customer"
        sequence: 2
        enabled: true
        override-priority: 10
    tags: ["eligibility", "customer"]
    metadata:
      owner: "Business Team"
      purpose: "Customer eligibility determination"
    execution-config:
      timeout-ms: 3000
      retry-count: 2
      circuit-breaker: false
```

Line-by-line explanation:

- **Line 1**: Root element defining a collection of rule groups
- **Line 2**: Start of the first rule group definition
- **Line 3**: Unique identifier for the rule group, used for referencing and management
- **Line 4**: Human-readable name displayed in logs and management interfaces
- **Line 5**: Detailed description explaining what this rule group validates
- **Line 6**: Single category assignment for organizational purposes
- **Line 7**: Priority level (10 = high priority, executes before lower priority groups)
- **Line 8**: Boolean flag to enable/disable this rule group without removing it
- **Line 9**: Whether to stop executing remaining rules when the first rule fails (false = continue)
- **Line 10**: Whether rules in this group can execute in parallel (false = sequential execution)
- **Line 11**: Start of the list of rule IDs that belong to this group
- **Line 12**: Reference to the age validation rule
- **Line 13**: Reference to the email validation rule
- **Line 14**: Reference to the income validation rule
- **Line 15**: Start of metadata section for governance and tracking
- **Line 16**: Team or person responsible for maintaining this rule group

- **Line 17**: Business domain this rule group belongs to
- **Line 19**: Start of the second rule group definition
- **Line 20**: Unique identifier for the eligibility check rule group
- **Line 21**: Human-readable name for the eligibility rule group
- **Line 22**: Description of the eligibility checking purpose
- **Line 23**: Primary category for this rule group
- **Line 24**: Multiple categories this rule group belongs to (eligibility and customer)
- **Line 25**: Priority level (20 = lower priority than the validation group)
- **Line 26**: This rule group is enabled and will execute
- **Line 27**: Continue executing all rules even if one fails
- **Line 28**: Rules in this group can execute in parallel for better performance
- **Line 29**: Start of rule references with additional configuration
- **Line 30**: Reference to the premium customer rule with specific settings
- **Line 31**: Execution sequence number (1 = execute first)
- **Line 32**: This specific rule reference is enabled
- **Line 33**: Override the rule's default priority with value 5
- **Line 34**: Reference to the long-term customer rule
- **Line 35**: Execution sequence number (2 = execute second)
- **Line 36**: This rule reference is enabled
- **Line 37**: Override priority for this rule reference
- **Line 38**: Tags for categorization and searching
- **Line 39**: Start of metadata for this rule group
- **Line 40**: Business team responsible for this rule group
- **Line 41**: Purpose statement for governance
- **Line 42**: Start of execution configuration section
- **Line 43**: Maximum execution time in milliseconds (3 seconds)
- **Line 44**: Number of retry attempts if execution fails
- **Line 45**: Whether to use circuit breaker pattern for fault tolerance

Execution Patterns

AND Group Execution

```
// AND group: ALL rules must pass
public RuleResult evaluateAndGroup(RuleGroup group, Map<String, Object> facts) {
    boolean allPassed = true;
    List<RuleResult> individualResults = new ArrayList<>();

    // Execute rules in sequence order
    for (Map.Entry<Integer, Rule> entry : group.getRulesBySequence().entrySet()) {
        Rule rule = entry.getValue();
        RuleResult result = rule.evaluate(facts);
        individualResults.add(result);

        if (!result.isTriggered()) {
            allPassed = false;
            if (group.isStopOnFirstFailure()) {
                break; // Early termination for performance
            }
        }
    }

    return new RuleResult(group.getName(),
                          allPassed ? "All rules passed" : "One or more rules failed",
                          allPassed, individualResults);
```

```
}
```

OR Group Execution

```
// OR group: ANY rule can pass
public RuleResult evaluateOrGroup(RuleGroup group, Map<String, Object> facts) {
    boolean anyPassed = false;
    List<RuleResult> individualResults = new ArrayList<>();

    // Execute rules in sequence order
    for (Map.Entry<Integer, Rule> entry : group.getRulesBySequence().entrySet()) {
        Rule rule = entry.getValue();
        RuleResult result = rule.evaluate(facts);
        individualResults.add(result);

        if (result.isTriggered()) {
            anyPassed = true;
            if (group.isStopOnFirstSuccess()) {
                break; // Early termination for performance
            }
        }
    }

    return new RuleResult(group.getName(),
                          anyPassed ? "At least one rule passed" : "No rules passed",
                          anyPassed, individualResults);
}
```

Nested Rules and Rule Chaining Patterns

When simple rules aren't enough, APEX provides sophisticated rule chaining patterns that let you build complex business workflows. These patterns handle scenarios where rules depend on the results of other rules, creating intelligent decision trees and multi-stage processes.

What is Rule Chaining? Rule chaining is like creating a flowchart in code. Instead of evaluating all rules independently, you can create sequences where:

- Rule B only runs if Rule A succeeds
- Rule C uses the result from Rule B as input
- Different rules run based on previous results
- Scores accumulate across multiple rules

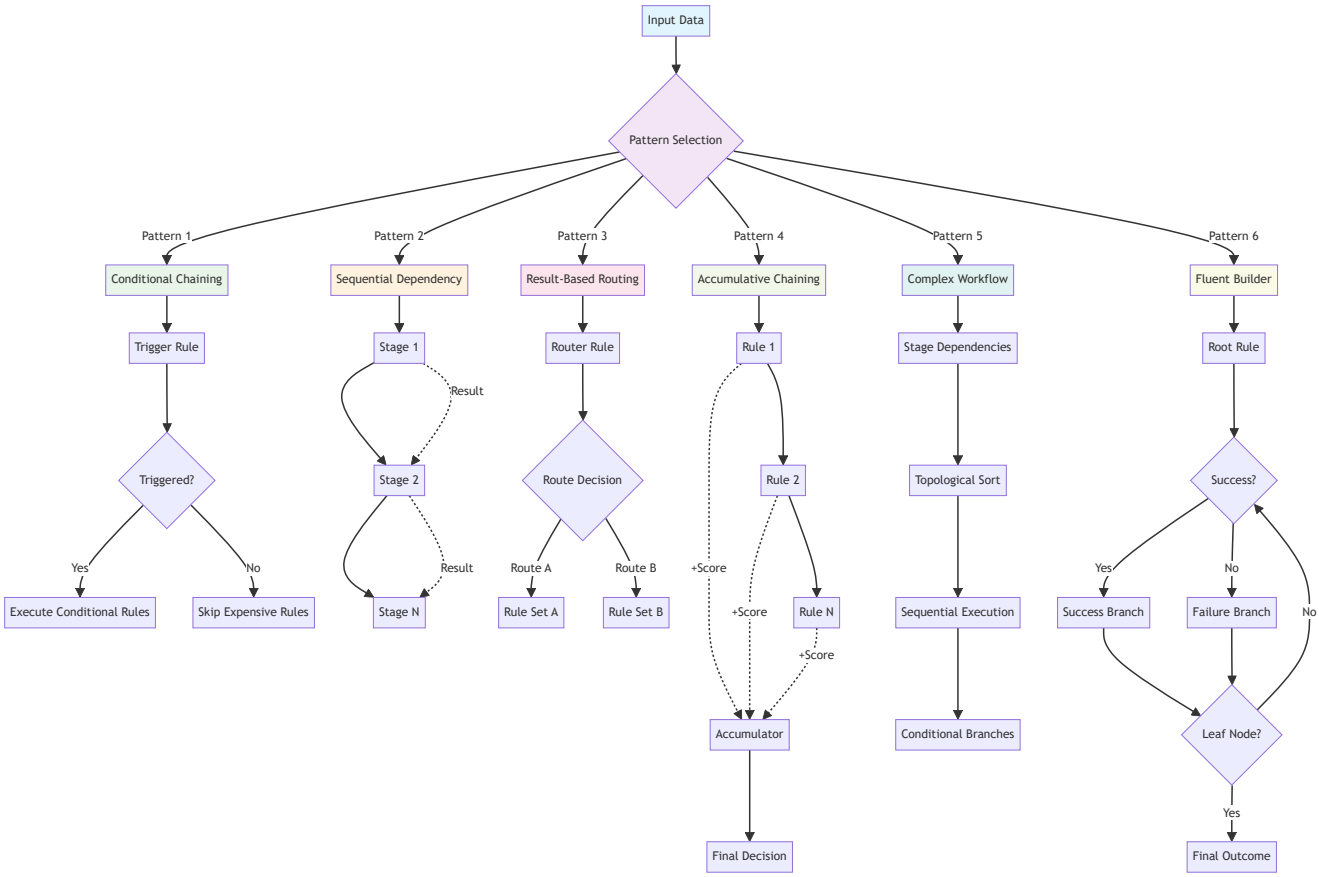
Why use Rule Chaining?

- **Performance**: Skip expensive operations when conditions aren't met
- **Complex logic**: Model real-world business processes with dependencies
- **Dynamic routing**: Send different data through different validation paths
- **Accumulative scoring**: Build up scores across multiple criteria
- **Decision trees**: Create branching logic based on intermediate results

Six powerful patterns available:

1. **Conditional Chaining**: Execute expensive rules only when needed
2. **Sequential Dependency**: Build processing pipelines

3. **Result-Based Routing**: Route to different rule sets based on results
4. **Accumulative Chaining**: Build up scores with intelligent rule selection
5. **Complex Workflow**: Multi-stage processes with dependencies
6. **Fluent Builder**: Decision trees with conditional branching



Pattern 1: Conditional Chaining

The Problem: You have expensive or specialized rules that should only run when certain conditions are met. For example, enhanced due diligence checks that only apply to high-value transactions.

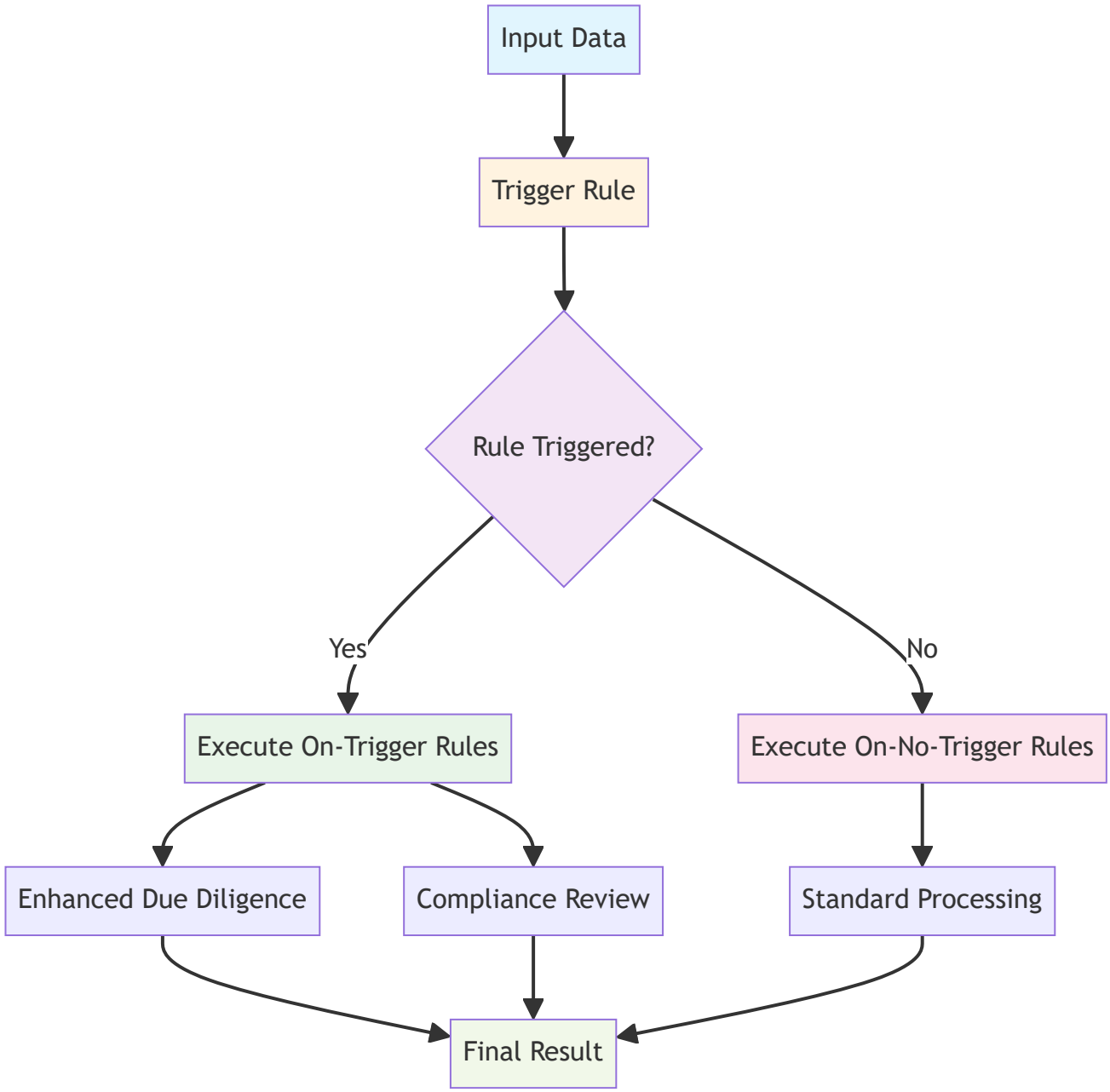
The Solution: Conditional Chaining executes a "trigger rule" first, then runs different rule sets based on whether the trigger succeeded or failed.

Real-world example:

- Trigger: "Is this a high-value customer transaction?"
- If YES: Run enhanced compliance checks, manager approval, detailed audit
- If NO: Run standard processing with basic validation

Benefits:

- **Performance**: Skip expensive operations when they're not needed
- **Resource efficiency**: Don't waste time on unnecessary processing
- **Clear logic**: Explicitly model conditional business processes
- **Cost optimization**: Avoid expensive API calls or database queries when possible




```
// Rule A: Check if customer qualifies for high-value processing
Rule ruleA = new Rule(
    "HighValueCustomerCheck",
    "#customerType == 'PREMIUM' && #transactionAmount > 100000",
    "Customer qualifies for high-value processing"
);

// Execute Rule A first
List<RuleResult> resultsA = ruleEngineService.evaluateRules(
    Arrays.asList(ruleA), createEvaluationContext(context));
RuleResult resultA = resultsA.get(0);

// Conditional execution of Rule B based on Rule A result
if (resultA.isTriggered()) {
    Rule ruleB = new Rule(
        "EnhancedDueDiligenceCheck",
        "#accountAge >= 3",
        "Enhanced due diligence check passed"
```

```
        );

        List<RuleResult> resultsB = ruleEngineService.evaluateRules(
            Arrays.asList(ruleB), createEvaluationContext(context));

        if (resultsB.get(0).isTriggered()) {
            System.out.println("APPROVED: High-value transaction approved with enhanced checks");
        } else {
            System.out.println("REJECTED: Enhanced due diligence failed");
        }
    } else {
        System.out.println("APPROVED: Standard processing applied");
    }
```

Line-by-line explanation:

- **Line 2**: Create a new Rule instance that will serve as the trigger rule for conditional processing
- **Line 3**: Set the rule ID to "HighValueCustomerCheck" for identification and logging
- **Line 4**: Define the condition using SpEL - checks if customer is PREMIUM and transaction exceeds 100,000
- **Line 5**: Set a descriptive message explaining what this rule validates
- **Line 9**: Execute the trigger rule using the rule engine service with the current context
- **Line 10**: Create an evaluation context from the current data context for rule execution
- **Line 11**: Extract the first (and only) result from the results list
- **Line 13**: Check if the trigger rule was triggered (condition evaluated to true)
- **Line 14**: If triggered, create the enhanced due diligence rule that will run expensive checks
- **Line 15**: Create a new Rule for enhanced due diligence with specific requirements
- **Line 16**: Set the rule ID for the enhanced check
- **Line 17**: Define condition requiring account age of at least 3 years
- **Line 18**: Set descriptive message for the enhanced due diligence rule
- **Line 21**: Execute the enhanced due diligence rule with the current context
- **Line 22**: Create evaluation context for the enhanced rule execution
- **Line 24**: Check if the enhanced due diligence rule passed
- **Line 25**: If passed, approve the transaction with enhanced checks message
- **Line 27**: If enhanced due diligence failed, reject the transaction
- **Line 29**: If trigger rule didn't fire, approve with standard processing (no enhanced checks needed)

Use Cases:

- Financial approval workflows with escalating requirements
- Multi-stage validation where expensive checks are conditional
- Risk-based processing with different validation paths

Pattern 2: Sequential Dependency

The Problem: You need to build up a result through multiple steps, where each step uses the output from the previous step. Like calculating a final price through multiple discount stages.

The Solution: Sequential Dependency creates a processing pipeline where each rule adds to or modifies the result from the previous rule.

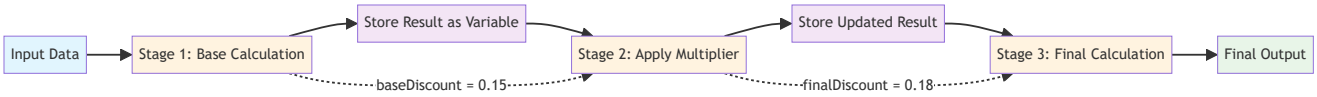
Real-world example:

1. **Stage 1**: Calculate base discount based on customer tier (15% for Gold customers)
2. **Stage 2**: Apply regional multiplier (20% bonus for US customers = 15% × 1.2 = 18%)

3. **Stage 3**: Calculate final amount using the accumulated discount

Benefits:

- **Clear progression**: Each step builds logically on the previous one
- **Maintainable**: Easy to add, remove, or modify individual stages
- **Debuggable**: Can inspect intermediate results at each stage
- **Reusable**: Individual stages can be reused in different pipelines




```
Map<String, Object> context = new HashMap<>();
context.put("baseAmount", new BigDecimal("100000"));
context.put("customerTier", "GOLD");
context.put("region", "US");

// Rule 1: Calculate base discount
Rule rule1 = new Rule(
    "BaseDiscountCalculation",
    "#customerTier == 'GOLD' ? 0.15 : (#customerTier == 'SILVER' ? 0.10 : 0.05)",
    "Calculate base discount based on customer tier"
);

StandardEvaluationContext evalContext = createEvaluationContext(context);
Double baseDiscount = evaluatorService.evaluate(rule1.getCondition(), evalContext, Double.class);

// Add result to context for next rule
context.put("baseDiscount", baseDiscount);

// Rule 2: Apply regional multiplier (depends on Rule 1 result)
Rule rule2 = new Rule(
    "RegionalMultiplierCalculation",
    "#region == 'US' ? #baseDiscount * 1.2 : #baseDiscount",
    "Apply regional multiplier to base discount"
);

evalContext = createEvaluationContext(context);
Double finalDiscount = evaluatorService.evaluate(rule2.getCondition(), evalContext, Double.class);
context.put("finalDiscount", finalDiscount);

// Rule 3: Calculate final amount (depends on Rule 2 result)
Rule rule3 = new Rule(
    "FinalAmountCalculation",
    "#baseAmount * (1 - #finalDiscount)",
    "Calculate final amount after all discounts"
);

evalContext = createEvaluationContext(context);
BigDecimal finalAmount = evaluatorService.evaluate(rule3.getCondition(), evalContext, BigDecimal.class);
```

Line-by-line explanation:

- **Line 1**: Create a HashMap to store all data that will be available to the rules
- **Line 2**: Set the base amount to 100,000 (could be order value, loan amount, etc.)
- **Line 3**: Set the customer tier to "GOLD" which will determine the base discount rate

- **Line 4**: Set the region to "US" which will affect the regional multiplier
- **Line 7**: Create the first rule that calculates the base discount based on customer tier
- **Line 8**: Set the rule ID for identification and logging purposes
- **Line 9**: Define a conditional expression that assigns different discount rates based on customer tier
- **Line 10**: Set a descriptive message explaining what this rule calculates
- **Line 13**: Create an evaluation context from the current data for rule execution
- **Line 14**: Execute the rule condition and get the calculated base discount as a Double
- **Line 17**: Add the calculated base discount back to the context so subsequent rules can use it
- **Line 20**: Create the second rule that applies a regional multiplier to the base discount
- **Line 21**: Set the rule ID for the regional multiplier calculation
- **Line 22**: Define condition that multiplies base discount by 1.2 for US customers, otherwise uses base discount
- **Line 23**: Set descriptive message for the regional multiplier rule
- **Line 26**: Create a new evaluation context with the updated data (including baseDiscount)
- **Line 27**: Execute the regional multiplier rule and get the final discount rate
- **Line 28**: Add the final discount rate to the context for use by the next rule
- **Line 31**: Create the third rule that calculates the final amount after applying all discounts
- **Line 32**: Set the rule ID for the final amount calculation
- **Line 33**: Define expression that calculates final amount: baseAmount × (1 - finalDiscount)
- **Line 34**: Set descriptive message for the final calculation rule
- **Line 37**: Create evaluation context with all accumulated data
- **Line 38**: Execute the final calculation rule and get the result as a BigDecimal

Use Cases:

- Pricing calculations with multiple factors
- Multi-step data transformations
- Sequential validation with cumulative results

Pattern 3: Result-Based Routing

The Problem: Different types of data need different processing rules. A high-risk transaction needs different validation than a low-risk one.

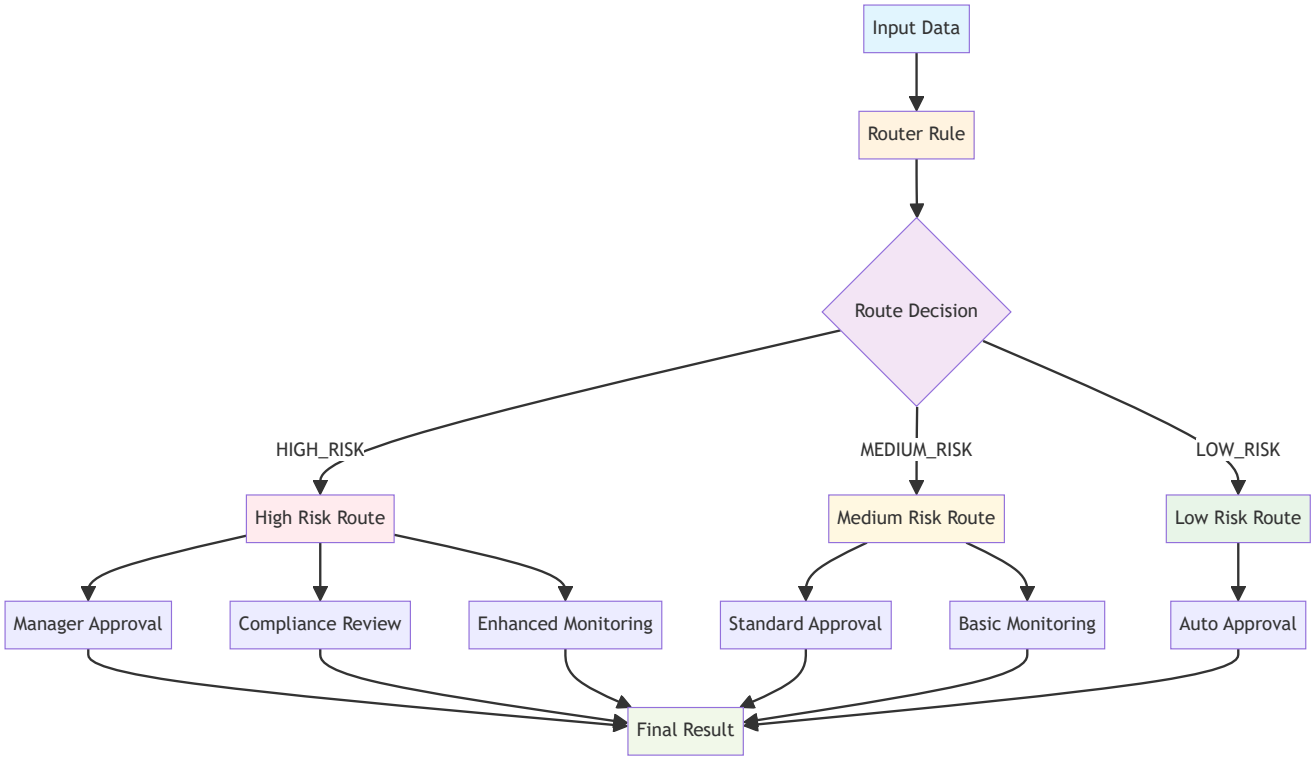
The Solution: Result-Based Routing uses a "router rule" to determine which path to take, then executes the appropriate rule set for that path.

Real-world example:

- **Router**: Calculate risk score and determine if it's HIGH_RISK, MEDIUM_RISK, or LOW_RISK
- **HIGH_RISK path**: Manager approval + compliance review + enhanced monitoring
- **MEDIUM_RISK path**: Standard approval + basic monitoring
- **LOW_RISK path**: Auto-approval only

Benefits:

- **Efficiency**: Only run the rules that are relevant to each case
- **Flexibility**: Easy to add new routing paths or modify existing ones
- **Clarity**: Business logic clearly shows different treatment for different scenarios
- **Scalability**: Can handle many different routing scenarios without complexity




```
// Router Rule: Determine processing path
Rule routerRule = new Rule(
    "ProcessingPathRouter",
    "#riskScore > 70 ? 'HIGH_RISK' : (#riskScore > 30 ? 'MEDIUM_RISK' : 'LOW_RISK')",
    "Determine processing path based on risk score"
);

StandardEvaluationContext evalContext = createEvaluationContext(context);
String processingPath = evaluatorService.evaluate(routerRule.getCondition(), evalContext, String.class);

// Route to appropriate rule set based on result
switch (processingPath) {
    case "HIGH_RISK":
        // Execute high-risk rules requiring multiple approvals
        Rule rule1 = new Rule("ManagerApprovalRequired", "#transactionAmount > 100000", "Manager approval required");
        Rule rule2 = new Rule("ComplianceReviewRequired", "#riskScore > 80", "Compliance review required");
        Rule rule3 = new Rule("CustomerHistoryCheck", "#customerHistory == 'EXCELLENT'", "Customer history must be excell

        List<Rule> highRiskRules = Arrays.asList(rule1, rule2, rule3);
        List<RuleResult> results = ruleEngineService.evaluateRules(highRiskRules, createEvaluationContext(context));

        boolean allPassed = results.stream().allMatch(RuleResult::isTriggered);
        System.out.println("HIGH RISK RESULT: " + (allPassed ? "APPROVED with multiple approvals" : "REQUIRES MANUAL REVI
        break;

    case "MEDIUM_RISK":
        // Execute medium-risk rules with standard approvals
        executeStandardApprovalRules(context);
        break;

    case "LOW_RISK":
        // Execute basic validation only
        executeBasicValidation(context);
        break;
```

```
}
```

Line-by-line explanation:

- **Line 2**: Create a router rule that will determine which processing path to take
- **Line 3**: Set the rule ID to "ProcessingPathRouter" for identification
- **Line 4**: Define a nested conditional expression that returns HIGH_RISK, MEDIUM_RISK, or LOW_RISK based on risk score
- **Line 5**: Set descriptive message explaining the router rule's purpose
- **Line 8**: Create an evaluation context from the current data context
- **Line 9**: Execute the router rule and get the processing path as a String result
- **Line 12**: Use a switch statement to route to different rule sets based on the determined path
- **Line 13**: Handle the HIGH_RISK case with the most stringent requirements
- **Line 15**: Create a rule requiring manager approval for transactions over 100,000
- **Line 16**: Create a rule requiring compliance review for very high risk scores (>80)
- **Line 17**: Create a rule requiring excellent customer history for high-risk processing
- **Line 19**: Combine all high-risk rules into a single list for batch execution
- **Line 20**: Execute all high-risk rules together and collect the results
- **Line 22**: Check if all high-risk rules passed using Java streams
- **Line 23**: Output the result - either approved with multiple approvals or requires manual review
- **Line 24**: Break out of the switch statement for HIGH_RISK case
- **Line 26**: Handle the MEDIUM_RISK case with standard approval processes
- **Line 28**: Call a method to execute standard approval rules (implementation not shown)
- **Line 29**: Break out of the switch statement for MEDIUM_RISK case
- **Line 31**: Handle the LOW_RISK case with minimal validation
- **Line 33**: Call a method to execute basic validation only (implementation not shown)
- **Line 34**: Break out of the switch statement for LOW_RISK case

Use Cases:

- Risk-based processing with different validation requirements
- Customer tier-based workflows
- Regulatory compliance with varying requirements

Pattern 4: Accumulative Chaining

The Problem: You need to build up a score from multiple criteria, like credit scoring where you evaluate income, credit history, employment, and debt ratio to get a final approval decision.

The Solution: Accumulative Chaining runs multiple rules that each contribute points to a running total, then makes a final decision based on the accumulated score.

Advanced Feature: **Weight-Based Rule Selection** - intelligently choose which rules to execute based on their importance, priority, or dynamic conditions. This saves processing time by focusing on the most critical rules.

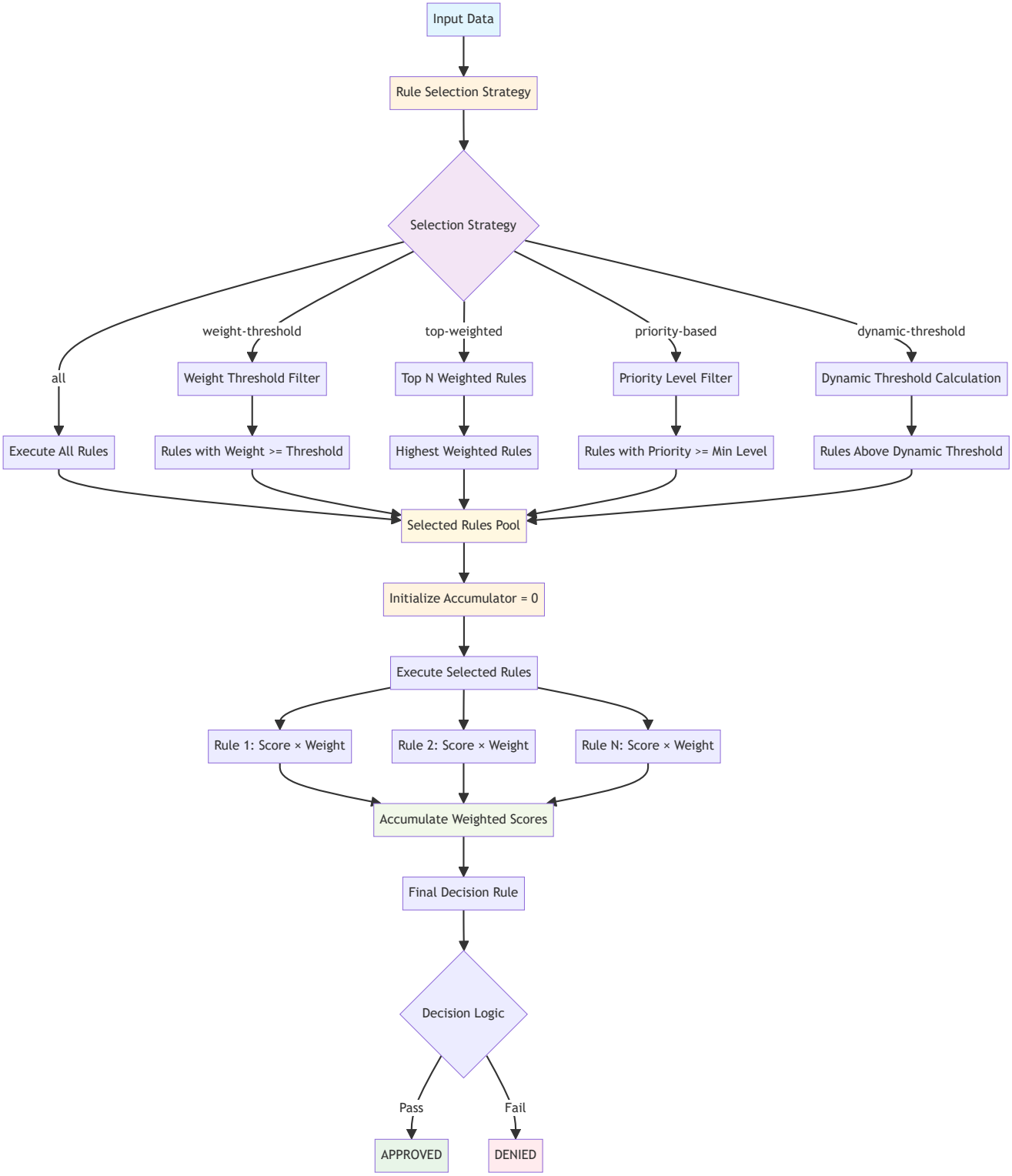
Real-world example:

- **Credit Score Rule**: 25 points if credit score ≥ 700 (weight: 0.9, high priority)
- **Income Rule**: 20 points if income ≥ $80,000 (weight: 0.8, high priority)
- **Employment Rule**: 15 points if employed ≥ 5 years (weight: 0.6, medium priority)
- **Debt Ratio Rule**: 10 points if debt ratio < 20% (weight: 0.5, low priority)
- **Final Decision**: APPROVED if total ≥ 60 points

Smart Rule Selection: With weight threshold 0.7, only the first two rules run (saving processing time), but still providing enough data for a good decision.

Benefits:

- **Intelligent processing**: Focus on the most important criteria
- **Performance optimization**: Skip low-importance rules when appropriate
- **Flexible scoring**: Different weights for different importance levels
- **Dynamic adaptation**: Adjust rule selection based on context (like risk level)



```
Map<String, Object> context = new HashMap<>();
context.put("creditScore", 750);
context.put("annualIncome", new BigDecimal("85000"));
context.put("employmentYears", 8);
context.put("existingDebt", new BigDecimal("25000"));
context.put("totalScore", 0); // Initialize accumulator

// Rule 1: Credit Score Component
Rule rule1 = new Rule(
    "CreditScoreEvaluation",
    "#creditScore >= 700 ? 25 : (#creditScore >= 650 ? 15 : (#creditScore >= 600 ? 10 : 0))",
    "Credit score component"
);

StandardEvaluationContext evalContext = createEvaluationContext(context);
Integer creditScorePoints = evaluatorService.evaluate(rule1.getCondition(), evalContext, Integer.class);
context.put("totalScore", (Integer)context.get("totalScore") + creditScorePoints);

// Rule 2: Income Component (adds to accumulated score)
Rule rule2 = new Rule(
    "IncomeEvaluation",
    "#annualIncome >= 80000 ? 20 : (#annualIncome >= 60000 ? 15 : (#annualIncome >= 40000 ? 10 : 0))",
    "Income component"
);

evalContext = createEvaluationContext(context);
Integer incomePoints = evaluatorService.evaluate(rule2.getCondition(), evalContext, Integer.class);
context.put("totalScore", (Integer)context.get("totalScore") + incomePoints);

// Rule 3: Employment Stability Component
Rule rule3 = new Rule(
    "EmploymentStabilityEvaluation",
    "#employmentYears >= 5 ? 15 : (#employmentYears >= 2 ? 10 : 5)",
    "Employment stability component"
);

evalContext = createEvaluationContext(context);
Integer employmentPoints = evaluatorService.evaluate(rule3.getCondition(), evalContext, Integer.class);
context.put("totalScore", (Integer)context.get("totalScore") + employmentPoints);

// Final Decision Rule (depends on accumulated score)
Rule finalRule = new Rule(
    "LoanDecision",
    "#totalScore >= 60 ? 'APPROVED' : (#totalScore >= 40 ? 'CONDITIONAL' : 'DENIED')",
    "Final loan decision based on total score"
);

evalContext = createEvaluationContext(context);
String decision = evaluatorService.evaluate(finalRule.getCondition(), evalContext, String.class);
```

Line-by-line explanation:

- **Line 1**: Create a HashMap to store all data and the accumulating score
- **Line 2**: Set the credit score to 750 (excellent credit)
- **Line 3**: Set the annual income to $85,000
- **Line 4**: Set employment years to 8 (stable employment)
- **Line 5**: Set existing debt to $25,000
- **Line 6**: Initialize the totalScore accumulator to 0 - this will collect points from all rules
- **Line 9**: Create the first rule that evaluates credit score and assigns points

- **Line 10**: Set rule ID for credit score evaluation
- **Line 11**: Define tiered scoring: 25 points for 700+, 15 for 650+, 10 for 600+, 0 otherwise
- **Line 12**: Set descriptive message for the credit score component
- **Line 15**: Create evaluation context with current data
- **Line 16**: Execute the credit score rule and get the points as an Integer
- **Line 17**: Add the credit score points to the running total (accumulation step)
- **Line 20**: Create the second rule that evaluates income and assigns points
- **Line 21**: Set rule ID for income evaluation
- **Line 22**: Define tiered scoring: 20 points for $80K+, 15 for$60K+, 10 for $40K+, 0 otherwise
- **Line 23**: Set descriptive message for the income component
- **Line 26**: Create fresh evaluation context with updated data (including current totalScore)
- **Line 27**: Execute the income rule and get the points
- **Line 28**: Add the income points to the running total (second accumulation)
- **Line 31**: Create the third rule that evaluates employment stability
- **Line 32**: Set rule ID for employment stability evaluation
- **Line 33**: Define tiered scoring: 15 points for 5+ years, 10 for 2+ years, 5 otherwise
- **Line 34**: Set descriptive message for employment stability component
- **Line 37**: Create evaluation context with all accumulated data
- **Line 38**: Execute the employment stability rule and get the points
- **Line 39**: Add employment points to the running total (final accumulation)
- **Line 42**: Create the final decision rule that uses the accumulated score
- **Line 43**: Set rule ID for the final loan decision
- **Line 44**: Define decision logic: APPROVED for 60+, CONDITIONAL for 40+, DENIED otherwise
- **Line 45**: Set descriptive message for the final decision rule
- **Line 48**: Create evaluation context with the final accumulated score
- **Line 49**: Execute the final decision rule and get the decision as a String

Weight-Based Rule Selection Strategies

One of the most powerful features of Accumulative Chaining is intelligent rule selection. Instead of always running every rule, APEX can automatically choose which rules to execute based on their importance, priority, or current context.

Why is this useful?

- **Performance**: Skip less important rules to save processing time
- **Resource efficiency**: Focus computational resources on the most critical evaluations
- **Dynamic adaptation**: Adjust rule selection based on risk level, customer tier, or other factors
- **Cost optimization**: Avoid expensive API calls or database queries for low-priority rules

Four selection strategies available:

Strategy 1: Weight Threshold Selection

Best for: Fixed importance levels where you want to skip rules below a certain importance

Execute only rules with weight above a specified threshold. Think of this as setting a "minimum importance level" - only rules that are important enough will run.

```
rule-selection:
  strategy: "weight-threshold"
```

```
    weight-threshold: 0.7  # Only execute rules with weight >= 0.7
```

Line-by-line explanation:

- **Line 1**: Configuration section for rule selection strategy
- **Line 2**: Set the selection strategy to "weight-threshold" - only rules above a certain weight will execute
- **Line 3**: Set the weight threshold to 0.7 - only rules with weight 0.7 or higher will be executed

Example: In credit scoring, you might set threshold 0.7 to focus only on the most critical factors (credit score, income) and skip less important ones (employment length, debt ratio).

Strategy 2: Top-Weighted Selection

Best for: Resource-constrained environments where you can only afford to run a limited number of rules

Execute the top N rules by weight, regardless of their actual weight values. This guarantees you'll run exactly the number of rules you specify.

```
  rule-selection:
    strategy: "top-weighted"
    max-rules: 3  # Execute only the 3 highest-weighted rules
```

Line-by-line explanation:

- **Line 1**: Configuration section for rule selection strategy
- **Line 2**: Set strategy to "top-weighted" - execute only the highest-weighted rules regardless of their actual weight values
- **Line 3**: Limit execution to the top 3 rules by weight - useful for resource-constrained environments

Example: In a high-volume trading system, you might only have time to run the 3 most important validation rules per transaction.

Strategy 3: Priority-Based Selection

Best for: Business scenarios where rules are organized by priority levels (HIGH/MEDIUM/LOW)

Execute rules based on priority levels, which is more business-friendly than numeric weights.

```
  rule-selection:
    strategy: "priority-based"
    min-priority: "MEDIUM"  # Execute HIGH and MEDIUM priority rules only
```

Line-by-line explanation:

- **Line 1**: Configuration section for rule selection strategy
- **Line 2**: Set strategy to "priority-based" - select rules based on business priority levels rather than numeric weights
- **Line 3**: Set minimum priority to "MEDIUM" - will execute HIGH and MEDIUM priority rules, skipping LOW priority rules

Example: During system maintenance, you might only run HIGH priority rules to reduce load, or during normal operations run HIGH and MEDIUM priority rules.

Strategy 4: Dynamic Threshold Selection

Best for: Context-aware systems where rule importance changes based on current conditions

Calculate the weight threshold dynamically based on the current context. This is the most flexible approach.

```yaml
rule-selection:
  strategy: "dynamic-threshold"
  threshold-expression: "#riskLevel == 'HIGH' ? 0.8 : 0.6"  # Higher threshold for high-risk scenarios
```

Line-by-line explanation:

- **Line 1**: Configuration section for rule selection strategy
- **Line 2**: Set strategy to "dynamic-threshold" - calculate the weight threshold dynamically based on current context
- **Line 3**: Define SpEL expression that sets threshold to 0.8 for high-risk scenarios, 0.6 otherwise - adapts rule selection to context

Example: For high-risk customers, use a higher threshold (0.8) to run only the most critical rules. For low-risk customers, use a lower threshold (0.6) to run more comprehensive checks.

Complete Example with Rule Selection

```yaml
rule-chains:
  - id: "selective-credit-scoring"
    pattern: "accumulative-chaining"
    configuration:
      accumulator-variable: "totalScore"
      initial-value: 0
      rule-selection:
        strategy: "weight-threshold"
        weight-threshold: 0.7
      accumulation-rules:
        - id: "credit-history"
          condition: "#creditScore >= 700 ? 30 : 15"
          weight: 0.9
          priority: "HIGH"
        - id: "income-verification"
          condition: "#annualIncome >= 80000 ? 25 : 10"
          weight: 0.8
          priority: "HIGH"
        - id: "employment-check"
          condition: "#employmentYears >= 5 ? 15 : 5"
          weight: 0.6  # Below threshold - will be skipped
          priority: "MEDIUM"
      final-decision-rule:
        condition: "#totalScore >= 40 ? 'APPROVED' : 'DENIED'"
```

Line-by-line explanation:

- **Line 1**: Root element defining a collection of rule chains
- **Line 2**: Start of a rule chain definition with unique identifier
- **Line 3**: Specify this chain uses the "accumulative-chaining" pattern for building up scores
- **Line 4**: Start of configuration section for this rule chain
- **Line 5**: Name of the variable that will accumulate the total score across all rules
- **Line 6**: Initial value for the accumulator (starts at 0)
- **Line 7**: Start of rule selection configuration section
- **Line 8**: Use "weight-threshold" strategy to select which rules to execute

- **Line 9**: Set weight threshold to 0.7 - only rules with weight >= 0.7 will execute
- **Line 10**: Start of the list of rules that can contribute to the accumulated score
- **Line 11**: First rule definition for credit history evaluation
- **Line 12**: SpEL condition that awards 30 points for credit score >= 700, otherwise 15 points
- **Line 13**: Weight of 0.9 (above threshold) - this rule will be selected for execution
- **Line 14**: Priority level set to HIGH for business importance
- **Line 15**: Second rule definition for income verification
- **Line 16**: SpEL condition that awards 25 points for income >= $80K, otherwise 10 points
- **Line 17**: Weight of 0.8 (above threshold) - this rule will be selected for execution
- **Line 18**: Priority level set to HIGH for business importance
- **Line 19**: Third rule definition for employment check
- **Line 20**: SpEL condition that awards 15 points for 5+ years employment, otherwise 5 points
- **Line 21**: Weight of 0.6 (below threshold) - this rule will be skipped due to weight threshold
- **Line 22**: Priority level set to MEDIUM (lower than the other rules)
- **Line 23**: Start of final decision rule configuration
- **Line 24**: SpEL condition that approves if total score >= 40, otherwise denies

Rule Selection Results:

- Total rules available: 3
- Rules selected for execution: 2 (credit-history and income-verification)
- Rules skipped: 1 (employment-check with weight 0.6 < 0.7)
- Final score calculation: Only selected rules contribute to the total

Configuration Reference:

Configuration Option	Type	Required	Default	Description
`rule-selection.strategy`	String	No	"all"	Selection strategy: "all", "weight-threshold", "top-weighted", "priority-based", "dynamic-threshold"
`rule-selection.weight-threshold`	Number	Conditional	-	Minimum weight for rule execution (required for "weight-threshold" strategy)
`rule-selection.max-rules`	Integer	Conditional	-	Maximum number of rules to execute (optional for "top-weighted" strategy)
`rule-selection.min-priority`	String	Conditional	"LOW"	Minimum priority level: "HIGH", "MEDIUM", "LOW" (for "priority-based" strategy)
`rule-selection.threshold-expression`	String	Conditional	-	SpEL expression for dynamic threshold calculation (required for "dynamic-threshold" strategy)

Rule Configuration Options:

Rule Option	Type	Required	Default	Description
`id`	String	No	Generated	Unique identifier for the rule
`condition`	String	Yes	-	SpEL expression that evaluates to a numeric score

Rule Option	Type	Required	Default	Description
`message`	String	No	Generated	Description of what this rule measures
`weight`	Number	No	1.0	Multiplier applied to the rule's score (also used for selection)
`priority`	String	No	"LOW"	Priority level: "HIGH", "MEDIUM", "LOW" (used for priority-based selection)

Selection Strategy Comparison:

Strategy	Best For	Performance	Flexibility	Use Case
all	Simple scenarios	Medium	Low	Traditional accumulative scoring
weight-threshold	Fixed importance levels	High	Medium	Skip low-importance rules
top-weighted	Resource constraints	High	Medium	Execute only most critical rules
priority-based	Hierarchical importance	High	Medium	Business priority-driven selection
dynamic-threshold	Context-aware selection	Medium	High	Market/risk-responsive rule execution

Use Cases:

- Credit scoring and loan approval systems with risk-based rule selection
- Performance evaluation with priority-weighted criteria
- Dynamic compliance checking based on regulatory requirements
- Resource-constrained environments where only critical rules should execute
- A/B testing scenarios with different rule selection strategies

Pattern 5: Complex Financial Workflow

The Problem: Real-world business processes often involve multiple stages with complex dependencies. For example, trade processing might require pre-validation, risk assessment, approval (which varies based on risk), and settlement calculation - all in a specific order with conditional logic.

The Solution: Complex Financial Workflow handles multi-stage processes where stages have dependencies on each other and can include conditional execution paths.

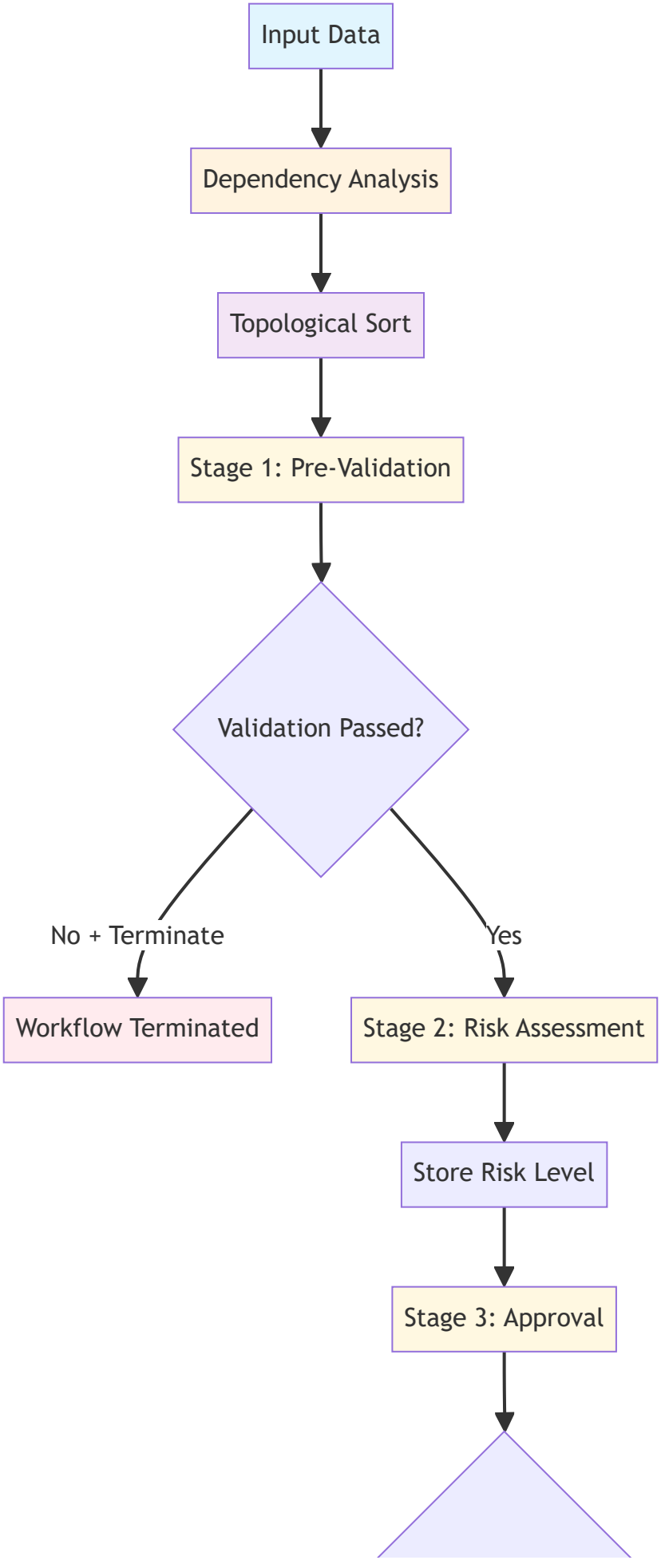
Real-world example - Trade Processing:

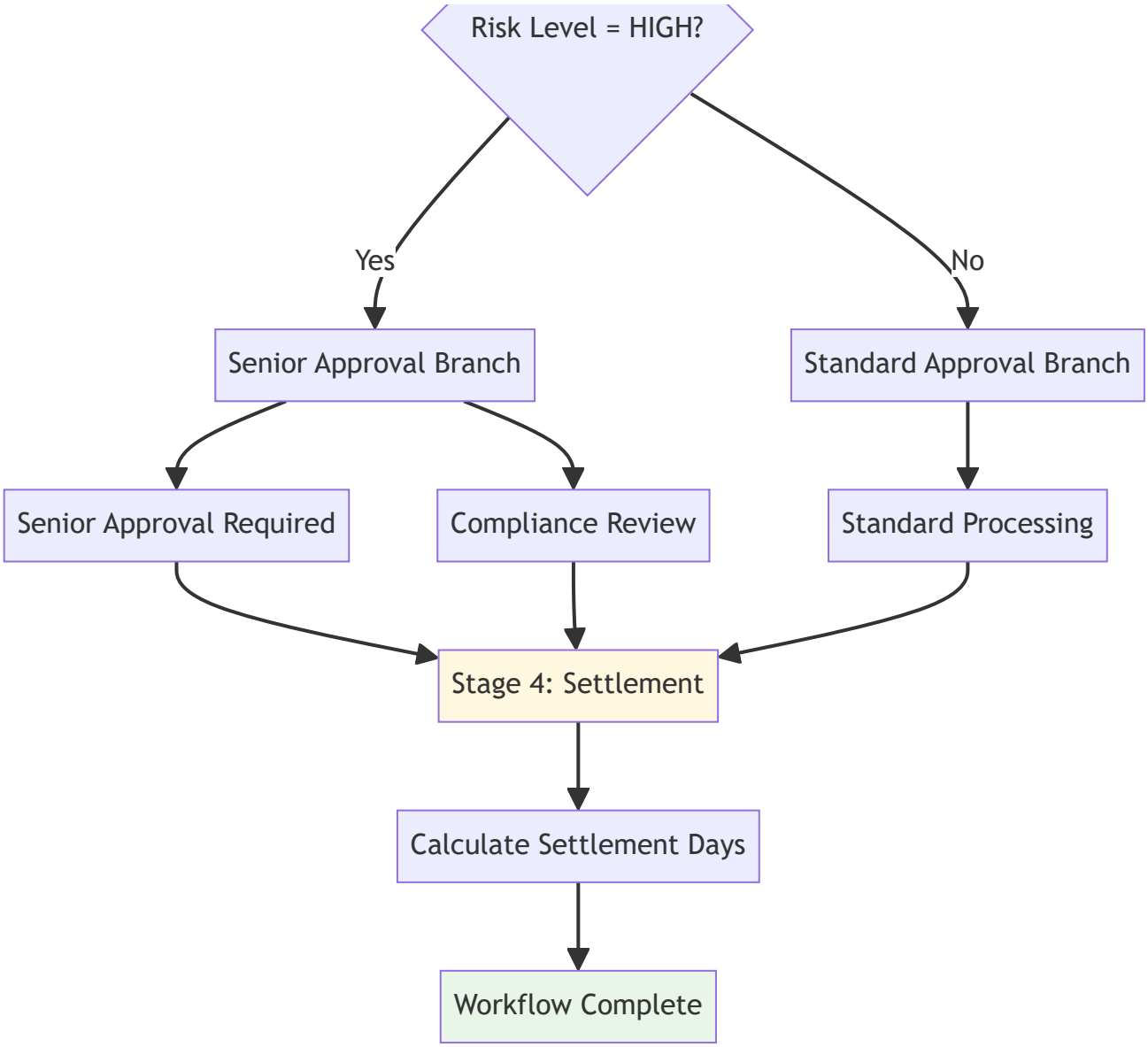
1. **Pre-validation**: Check basic trade data (if this fails, stop everything)
2. **Risk Assessment**: Calculate risk level (depends on pre-validation passing)
3. **Approval**: Different approval rules based on risk level (depends on risk assessment)
4. **Settlement**: Calculate settlement days (depends on approval)

Key Features:

- **Dependency management**: Stages execute in the correct order automatically
- **Conditional execution**: Different rules run based on previous results

- **Failure handling**: Configure whether failures stop the workflow or continue
- **Result passing**: Each stage can pass data to subsequent stages
- **Circular dependency detection**: Prevents infinite loops


Input Data
Dependency Analysis
Topological Sort
Stage 1: Pre-Validation
Validation Passed?
No + Terminate
Yes
Workflow Terminated
Stage 2: Risk Assessment
Store Risk Level
Stage 3: Approval




```
// Complex workflow execution example
Map<String, Object> tradeContext = new HashMap<>();
tradeContext.put("tradeType", "DERIVATIVE");
tradeContext.put("notionalAmount", new BigDecimal("2000000"));
tradeContext.put("counterparty", "BANK_A");
tradeContext.put("marketVolatility", 0.25);
tradeContext.put("seniorApprovalObtained", true);

ChainedEvaluationContext context = new ChainedEvaluationContext(tradeContext);
RuleChainResult result = ruleChainExecutor.executeRuleChain(complexWorkflowChain, context);

// Access stage results
String riskLevel = (String) result.getStageResult("riskLevel");
Integer settlementDays = (Integer) result.getStageResult("settlementDays");
String stageResult = (String) result.getStageResult("stage_pre-validation_result");

System.out.println("Workflow completed: " + result.getFinalOutcome());
System.out.println("Risk Level: " + riskLevel);
System.out.println("Settlement Days: " + settlementDays);
System.out.println("Execution Path: " + String.join(" → ", result.getExecutionPath()));
```

YAML Configuration:

```yaml
rule-chains:
  - id: "trade-processing-workflow"
    name: "Trade Processing Workflow"
    pattern: "complex-workflow"
    enabled: true
    configuration:
      stages:
        - stage: "pre-validation"
          name: "Pre-Validation Stage"
          rules:
            - condition: "#tradeType != null && #notionalAmount != null && #counterparty != null"
              message: "Basic trade data validation"
          failure-action: "terminate"
        - stage: "risk-assessment"
          name: "Risk Assessment Stage"
          depends-on: ["pre-validation"]
          rules:
            - condition: "#notionalAmount > 1000000 && #marketVolatility > 0.2 ? 'HIGH' : 'MEDIUM'"
              message: "Risk level assessment"
          output-variable: "riskLevel"
        - stage: "approval"
          name: "Approval Stage"
          depends-on: ["risk-assessment"]
          conditional-execution:
            condition: "#riskLevel == 'HIGH'"
            on-true:
              rules:
                - condition: "#seniorApprovalObtained == true"
                  message: "Senior approval required for high-risk trades"
            on-false:
              rules:
                - condition: "true"
                  message: "Standard approval applied"
```

Line-by-line explanation:

- **Line 1**: Root element defining a collection of rule chains
- **Line 2**: Unique identifier for this complex workflow rule chain
- **Line 3**: Human-readable name for the workflow
- **Line 4**: Specify this uses the "complex-workflow" pattern for multi-stage processing with dependencies
- **Line 5**: Boolean flag to enable this workflow (can be disabled without removing configuration)
- **Line 6**: Start of configuration section for the workflow
- **Line 7**: Start of stages definition - each stage represents a phase in the workflow
- **Line 8**: First stage identifier for pre-validation phase
- **Line 9**: Human-readable name for the pre-validation stage
- **Line 10**: Start of rules list for this stage
- **Line 11**: SpEL condition checking that required trade fields are not null
- **Line 12**: Descriptive message for the basic validation rule
- **Line 13**: Action to take if this stage fails - "terminate" stops the entire workflow
- **Line 14**: Second stage identifier for risk assessment phase
- **Line 15**: Human-readable name for the risk assessment stage
- **Line 16**: Dependency declaration - this stage depends on pre-validation completing successfully
- **Line 17**: Start of rules list for risk assessment stage
- **Line 18**: SpEL condition that returns 'HIGH' or 'MEDIUM' risk based on amount and volatility
- **Line 19**: Descriptive message for the risk assessment rule

- **Line 20**: Output variable name - the result of this stage will be stored as "riskLevel"
- **Line 21**: Third stage identifier for approval phase
- **Line 22**: Human-readable name for the approval stage
- **Line 23**: Dependency declaration - this stage depends on risk assessment completing
- **Line 24**: Start of conditional execution configuration
- **Line 25**: SpEL condition that determines which branch to execute based on risk level
- **Line 26**: Start of rules to execute when condition is true (high risk)
- **Line 27**: Start of rules list for high-risk branch
- **Line 28**: SpEL condition requiring senior approval for high-risk trades
- **Line 29**: Descriptive message for senior approval requirement
- **Line 30**: Start of rules to execute when condition is false (not high risk)
- **Line 31**: Start of rules list for standard risk branch
- **Line 32**: SpEL condition that always passes (true) for standard approval
- **Line 33**: Descriptive message for standard approval path

Key Features:

- **Dependency Management**: Stages execute in dependency order using topological sort
- **Conditional Execution**: Stages can have conditional logic with on-true/on-false branches
- **Failure Actions**: Configure whether stage failures terminate the workflow or continue
- **Output Variables**: Store stage results for use in subsequent stages
- **Circular Dependency Detection**: Prevents infinite loops in stage dependencies

Use Cases:

- Trade processing and settlement workflows
- Multi-stage approval processes
- Complex financial instrument validation
- Regulatory compliance workflows

Pattern 6: Fluent Rule Builder

The Problem: You need to create complex decision trees where the path taken depends on the results of previous decisions. Like a customer onboarding process where VIP customers get different treatment than standard customers.

The Solution: Fluent Rule Builder creates decision trees where each rule can have success and failure branches, leading to different subsequent rules.

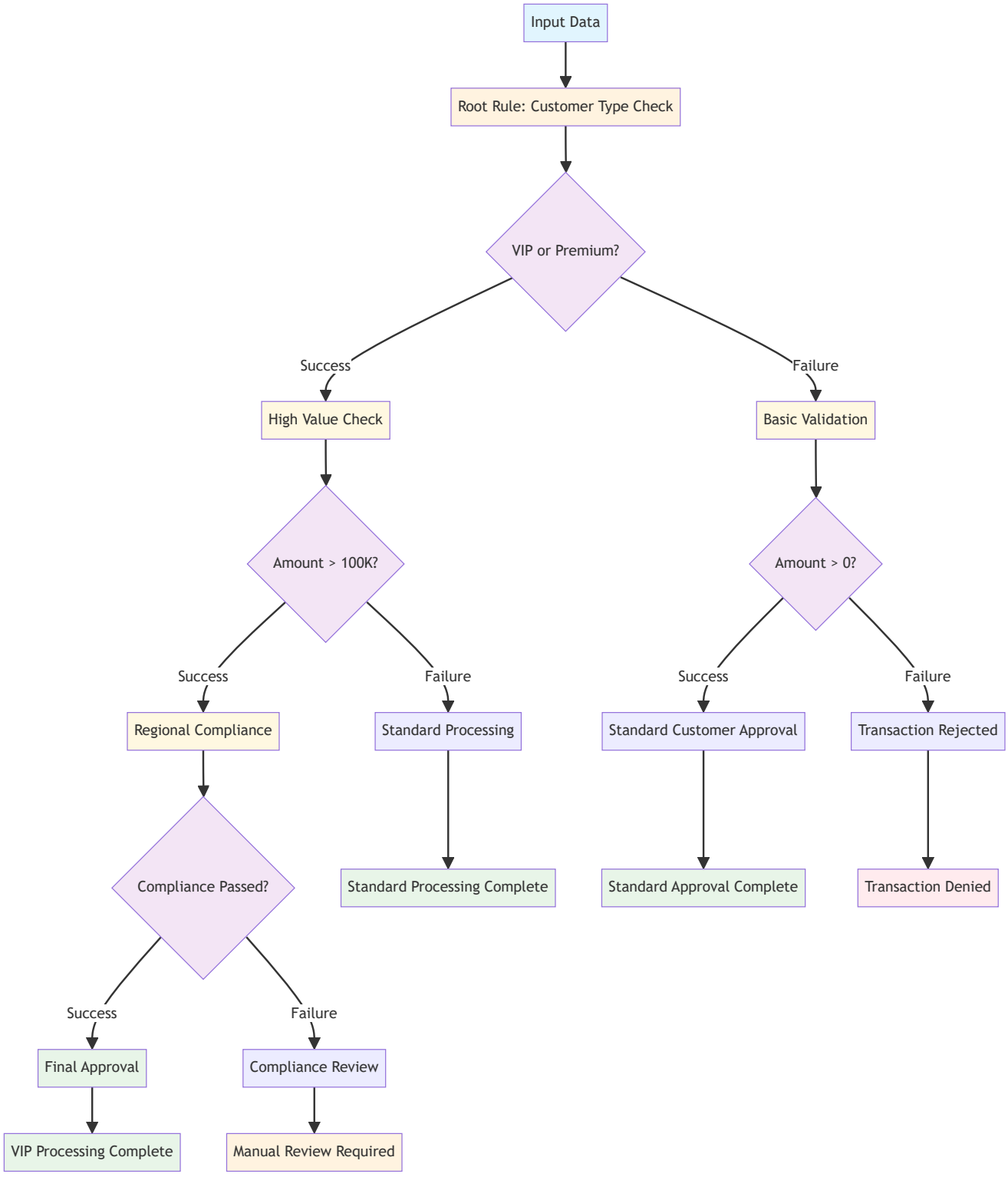
Real-world example - Customer Processing:

- **Root**: Is this a VIP or Premium customer?
  - **If YES**: Check if transaction > $100K
    - **If YES**: Run regional compliance checks → Final approval
    - **If NO**: Standard processing
  - **If NO**: Basic validation → Standard customer approval

Key Features:

- **Tree structure**: Rules form a decision tree with branching logic
- **Success/failure paths**: Each rule can define what happens next for both outcomes
- **Depth tracking**: Prevents infinite recursion with configurable limits
- **Execution tracking**: Full audit trail of which path was taken

- **Flexible branching**: Can create complex nested decision logic



YAML Configuration:

```yaml
rule-chains:
  - id: "customer-decision-tree"
    name: "Customer Processing Decision Tree"
    pattern: "fluent-builder"
    enabled: true
```

```yaml
    configuration:
      root-rule:
        id: "customer-type-check"
        condition: "#customerType == 'VIP' || #customerType == 'PREMIUM'"
        message: "High-tier customer detected"
        on-success:
          rule:
            id: "high-value-check"
            condition: "#transactionAmount > 100000"
            message: "High-value transaction detected"
            on-success:
              rule:
                id: "final-approval"
                condition: "true"
                message: "Final approval granted"
            on-failure:
              rule:
                id: "standard-processing"
                condition: "true"
                message: "Standard processing applied"
        on-failure:
          rule:
            id: "basic-validation"
            condition: "#transactionAmount > 0"
            message: "Basic validation check"
```

Java Execution Example:

```java
// Fluent builder execution example
Map<String, Object> customerContext = new HashMap<>();
customerContext.put("customerType", "VIP");
customerContext.put("transactionAmount", new BigDecimal("150000"));
customerContext.put("region", "US");
customerContext.put("accountAge", 6);

ChainedEvaluationContext context = new ChainedEvaluationContext(customerContext);
RuleChainResult result = ruleChainExecutor.executeRuleChain(fluentBuilderChain, context);

// Access individual rule results from the decision tree
Boolean customerTypeResult = (Boolean) result.getStageResult("fluent_rule_customer-type-check_result");
Boolean highValueResult = (Boolean) result.getStageResult("fluent_rule_high-value-check_result");
Boolean finalApprovalResult = (Boolean) result.getStageResult("fluent_rule_final-approval_result");

System.out.println("Decision tree completed: " + result.getFinalOutcome());
System.out.println("Customer Type Check: " + customerTypeResult);
System.out.println("High Value Check: " + highValueResult);
System.out.println("Final Approval: " + finalApprovalResult);
System.out.println("Execution Path: " + String.join(" → ", result.getExecutionPath()));

// Example with failure path
Map<String, Object> standardCustomerContext = new HashMap<>();
standardCustomerContext.put("customerType", "STANDARD");
standardCustomerContext.put("transactionAmount", new BigDecimal("15000"));

ChainedEvaluationContext failureContext = new ChainedEvaluationContext(standardCustomerContext);
RuleChainResult failureResult = ruleChainExecutor.executeRuleChain(fluentBuilderChain, failureContext);

// This will follow the on-failure branch to basic-validation
Boolean basicValidationResult = (Boolean) failureResult.getStageResult("fluent_rule_basic-validation_result");
System.out.println("Failure path result: " + basicValidationResult);
```

Key Features:

- **Tree Structure**: Rules form a decision tree with success/failure branches
- **Depth Tracking**: Prevents infinite recursion with configurable depth limits (max 20)
- **Execution Path Tracking**: Full audit trail of which branch was taken
- **Leaf Node Detection**: Automatically detects when decision tree execution completes
- **Flexible Branching**: Each rule can have both success and failure branches
- **Result Tracking**: Individual rule results stored with unique keys

Configuration Options:

Option	Type	Required	Description
`root-rule`	Object	Yes	The root rule that starts the decision tree
`root-rule.condition`	String	Yes	SpEL expression for the root rule
`root-rule.on-success`	Object	No	Configuration for success branch
`root-rule.on-failure`	Object	No	Configuration for failure branch
`rule.id`	String	No	Unique identifier for the rule
`rule.message`	String	No	Description of what the rule does

Use Cases:

- Complex decision trees with multiple conditional branches
- Business process automation with conditional logic
- Nested validation requirements
- Dynamic rule execution paths
- Customer onboarding workflows
- Approval decision trees
- **RuleChain**: Executes the composed rule chain
- **RuleChainResult**: Contains execution results and path information

Use Cases:

- Complex decision trees with multiple conditional branches
- Financial approval workflows with escalating requirements
- Risk assessment pipelines with dynamic routing
- Business process automation with conditional logic
- Compliance checking with nested validation requirements

Benefits:

- **Declarative Composition**: Rules are composed declaratively using a fluent API
- **Conditional Branching**: Clear success/failure paths with `onSuccess()` and `onFailure()` methods
- **Execution Tracking**: Full visibility into which rules were executed and the path taken
- **Reusable Chains**: Rule chains can be built once and executed multiple times with different contexts
- **Nested Complexity**: Supports deeply nested conditional logic with multiple branches

Helper Methods for All Patterns

```
/**
 * Helper method to create evaluation context from a map.
 */
private static StandardEvaluationContext createEvaluationContext(Map<String, Object> variables) {
    StandardEvaluationContext context = new StandardEvaluationContext();
    variables.forEach(context::setVariable);
    return context;
}

/**
 * Helper method for executing rule sets with error handling.
 */
private static List<RuleResult> executeRulesWithErrorHandling(List<Rule> rules,
                                                              StandardEvaluationContext context,
                                                              RuleEngineService ruleEngineService) {
    try {
        return ruleEngineService.evaluateRules(rules, context);
    } catch (Exception e) {
        System.err.println("Error executing rules: " + e.getMessage());
        return Collections.emptyList();
    }
}
```

Pattern Selection Guide

Pattern	Use When	Complexity	Performance	Status
Conditional Chaining	Need to execute expensive rules only when conditions are met	Low	High	✅ Implemented
Sequential Dependency	Each step builds upon the previous result	Medium	Medium	✅ Implemented
Result-Based Routing	Different processing paths based on intermediate results	Medium	Medium	✅ Implemented
Accumulative Chaining	Building scores or cumulative results across multiple criteria	Medium	Medium	✅ Implemented
Complex Financial Workflow	Multi-stage business processes with dependencies	High	Medium	✅ Implemented
Fluent Rule Builder	Complex decision trees with multiple branches	High	Medium	✅ Implemented

Performance Monitoring

Automatic Metrics Collection

The engine automatically collects comprehensive performance metrics with < 1% overhead:

```
// Enable performance monitoring
RulesEngineConfiguration config = new RulesEngineConfiguration.Builder()
    .withPerformanceMonitoring(true)
```

```
    .withMetricsCollectionInterval(Duration.ofSeconds(30))
    .build();

// Access performance data
PerformanceSnapshot snapshot = engine.getPerformanceMonitor().getSnapshot();
System.out.println("Average execution time: " + snapshot.getAverageExecutionTime());
System.out.println("Memory usage: " + snapshot.getMemoryUsage());
```

Line-by-line explanation:

- **Line 2**: Create a new configuration builder for setting up the rules engine
- **Line 3**: Enable performance monitoring to track execution times, memory usage, and other metrics
- **Line 4**: Set the metrics collection interval to 30 seconds for regular performance snapshots
- **Line 5**: Build the final configuration object with performance monitoring enabled
- **Line 8**: Get a performance snapshot from the engine's performance monitor
- **Line 9**: Print the average execution time across all rule evaluations
- **Line 10**: Print the current memory usage of the rules engine

Performance Metrics

Rule-Level Metrics

- Execution time (min, max, average, percentiles)
- Success/failure rates
- Memory usage per rule
- Cache hit/miss ratios

Engine-Level Metrics

- Total throughput (rules/second)
- Concurrent execution statistics
- Resource utilization
- Error rates and patterns

Performance Analysis

```
// Get detailed performance analysis
PerformanceAnalyzer analyzer = engine.getPerformanceAnalyzer();
List<PerformanceInsight> insights = analyzer.analyzePerformance();

for (PerformanceInsight insight : insights) {
    System.out.println("Issue: " + insight.getIssue());
    System.out.println("Recommendation: " + insight.getRecommendation());
    System.out.println("Impact: " + insight.getImpact());
}
```

Line-by-line explanation:

- **Line 2**: Get the performance analyzer from the rules engine for detailed analysis
- **Line 3**: Analyze current performance data and get a list of insights and recommendations
- **Line 5**: Iterate through each performance insight returned by the analyzer
- **Line 6**: Print the identified performance issue (e.g., "Slow rule execution detected")
- **Line 7**: Print the recommended action to address the issue (e.g., "Consider adding caching")

- **Line 8**: Print the potential impact of the issue (e.g., "High - affects all rule evaluations")

Rule Metadata System

Core Audit Dates

The **two most critical attributes** for any enterprise rule are:

1. `createdDate` - When the rule was first created (NEVER null)
2. `modifiedDate` - When the rule was last modified (NEVER null)

```
// These methods NEVER return null
Instant created = rule.getCreatedDate();     // ALWAYS available
Instant modified = rule.getModifiedDate();   // ALWAYS available

// Direct access from metadata
Instant created2 = rule.getMetadata().getCreatedDate();   // ALWAYS available
Instant modified2 = rule.getMetadata().getModifiedDate(); // ALWAYS available
```

Line-by-line explanation:

- **Line 2**: Get the creation date from the rule - this method is guaranteed to never return null
- **Line 3**: Get the last modified date from the rule - this method is guaranteed to never return null
- **Line 6**: Alternative way to access creation date directly from the rule's metadata object
- **Line 7**: Alternative way to access modified date directly from the rule's metadata object

Comprehensive Metadata

```
// Creating rules with full metadata
Rule rule = configuration.rule("TRADE-VAL-001")
    .withName("Trade Amount Validation")
    .withCondition("#amount > 0 && #amount <= 1000000")
    .withMessage("Trade amount is valid")
    .withMetadata(metadata -> metadata
        .withOwner("Trading Team")
        .withDomain("Finance")
        .withPurpose("Regulatory compliance")
        .withVersion("1.2.0")
        .withTags("trading", "validation", "regulatory")
        .withEffectiveDate(LocalDate.of(2024, 1, 1))
        .withExpirationDate(LocalDate.of(2024, 12, 31))
        .withCustomProperty("regulatoryReference", "MiFID-II-2024")
        .withCustomProperty("businessOwner", "john.doe@company.com")
    )
    .build();
```

Line-by-line explanation:

- **Line 2**: Start building a rule with the unique identifier "TRADE-VAL-001"
- **Line 3**: Set the human-readable name for this rule
- **Line 4**: Define the SpEL condition that validates trade amount is positive and not exceeding 1 million
- **Line 5**: Set the success message when the rule passes
- **Line 6**: Start configuring comprehensive metadata using a lambda expression

- **Line 7**: Set the team or person responsible for maintaining this rule
- **Line 8**: Set the business domain this rule belongs to (Finance)
- **Line 9**: Set the purpose statement explaining why this rule exists
- **Line 10**: Set the version number for tracking rule changes over time
- **Line 11**: Add tags for categorization and searching (trading, validation, regulatory)
- **Line 12**: Set the effective date when this rule becomes active (January 1, 2024)
- **Line 13**: Set the expiration date when this rule should be reviewed or retired
- **Line 14**: Add custom property linking to specific regulatory reference
- **Line 15**: Add custom property with business owner contact information
- **Line 17**: Build the final rule with all metadata configured

Metadata Queries

```
// Query rules by metadata
List<Rule> tradingRules = engine.getRules().stream()
    .filter(rule -> rule.getMetadata().getDomain().equals("Finance"))
    .filter(rule -> rule.getMetadata().getTags().contains("trading"))
    .collect(Collectors.toList());

// Find rules modified in the last 30 days
Instant thirtyDaysAgo = Instant.now().minus(30, ChronoUnit.DAYS);
List<Rule> recentlyModified = engine.getRules().stream()
    .filter(rule -> rule.getModifiedDate().isAfter(thirtyDaysAgo))
    .collect(Collectors.toList());
```

Configuration Examples

Basic Rule Configuration

```
metadata:
  name: "Customer Validation Rules"
  version: "1.0.0"
  description: "Basic customer validation"

rules:
  - id: "age-validation"
    name: "Age Check"
    condition: "#data.age >= 18"
    message: "Customer must be at least 18 years old"
    severity: "ERROR"
    metadata:
      owner: "Customer Team"
      domain: "Customer Management"
      tags: ["validation", "age"]
```

Line-by-line explanation:

- **Line 1**: Start of metadata section for the entire rule configuration file
- **Line 2**: Human-readable name for this collection of rules
- **Line 3**: Version number for tracking changes to this rule set
- **Line 4**: Description explaining the purpose of these rules
- **Line 6**: Start of the rules collection
- **Line 7**: First rule definition with unique identifier "age-validation"

- **Line 8**: Human-readable name for the age validation rule
- **Line 9**: SpEL condition that checks if the customer's age is at least 18
- **Line 10**: Error message displayed when the rule fails
- **Line 11**: Severity level set to "ERROR" indicating this is a critical validation
- **Line 12**: Start of metadata section for this specific rule
- **Line 13**: Team responsible for maintaining this rule
- **Line 14**: Business domain this rule belongs to
- **Line 15**: Tags for categorization and searching

Advanced Enrichment Configuration

```yaml
enrichments:
  - id: "comprehensive-currency-enrichment"
    type: "lookup-enrichment"
    condition: "['currency'] != null"
    enabled: true
    lookup-config:
      lookup-dataset:
        type: "yaml-file"
        file-path: "datasets/currencies.yaml"
        key-field: "code"
        cache-enabled: true
        cache-ttl-seconds: 3600
        preload-enabled: true
        default-values:
          region: "Unknown"
          isActive: false
          decimalPlaces: 2
    field-mappings:
      - source-field: "name"
        target-field: "currencyName"
      - source-field: "region"
        target-field: "currencyRegion"
      - source-field: "isActive"
        target-field: "currencyActive"
      - source-field: "decimalPlaces"
        target-field: "currencyDecimals"
    metadata:
      owner: "Finance Team"
      purpose: "Currency standardization"
      lastUpdated: "2024-07-26"
```

Line-by-line explanation:

- **Line 1**: Start of enrichments collection - enrichments add data to input before rule evaluation
- **Line 2**: Unique identifier for this currency enrichment
- **Line 3**: Type of enrichment - "lookup-enrichment" performs data lookups from external sources
- **Line 4**: SpEL condition that determines when this enrichment runs (only if currency field exists)
- **Line 5**: Boolean flag to enable/disable this enrichment
- **Line 6**: Start of lookup configuration section
- **Line 7**: Start of dataset configuration
- **Line 8**: Dataset type - "yaml-file" loads data from a YAML file
- **Line 9**: Path to the YAML file containing currency data
- **Line 10**: Field name in the dataset that serves as the lookup key
- **Line 11**: Enable caching to improve performance for repeated lookups
- **Line 12**: Cache time-to-live of 3600 seconds (1 hour)

- **Line 13**: Preload the entire dataset into memory at startup for faster access
- **Line 14**: Start of default values section for missing data
- **Line 15**: Default region value when not found in dataset
- **Line 16**: Default active status when not found in dataset
- **Line 17**: Default decimal places when not found in dataset
- **Line 18**: Start of field mappings section
- **Line 19**: Map the "name" field from dataset to "currencyName" in the enriched data
- **Line 20**: Target field name in the enriched data
- **Line 21**: Map the "region" field from dataset to "currencyRegion" in the enriched data
- **Line 22**: Target field name for currency region
- **Line 23**: Map the "isActive" field from dataset to "currencyActive" in the enriched data
- **Line 24**: Target field name for currency active status
- **Line 25**: Map the "decimalPlaces" field from dataset to "currencyDecimals" in the enriched data
- **Line 26**: Target field name for decimal places information
- **Line 27**: Start of metadata section for governance
- **Line 28**: Team responsible for maintaining this enrichment
- **Line 29**: Purpose statement explaining why this enrichment exists
- **Line 30**: Last updated date for tracking changes

Multi-Environment Configuration

```yaml
# config-dev.yaml
metadata:
  name: "Development Configuration"
  environment: "development"

enrichments:
  - id: "test-data-enrichment"
    lookup-config:
      lookup-dataset:
        type: "inline"
        data:
          - code: "TEST"
            name: "Test Data"

# config-prod.yaml
metadata:
  name: "Production Configuration"
  environment: "production"

enrichments:
  - id: "production-data-enrichment"
    lookup-config:
      lookup-dataset:
        type: "yaml-file"
        file-path: "datasets/production-data.yaml"
```

Performance-Optimized Configuration

```yaml
enrichments:
  - id: "high-performance-lookup"
    type: "lookup-enrichment"
    lookup-config:
      lookup-dataset:
        type: "yaml-file"
        file-path: "datasets/large-dataset.yaml"
```

```
        key-field: "id"
        # Performance optimizations
        cache-enabled: true
        cache-ttl-seconds: 7200
        preload-enabled: true
        cache-max-size: 10000
        # Monitoring
        performance-monitoring: true
        log-cache-stats: true
```

Integration Patterns

Spring Boot Integration

```
@Configuration
@EnableRulesEngine
public class RulesEngineConfig {

    @Bean
    @Primary
    public RulesEngine primaryRulesEngine() {
        return YamlConfigurationLoader.load("classpath:rules/primary-rules.yaml")
            .createEngine();
    }

    @Bean("validationEngine")
    public RulesEngine validationRulesEngine() {
        return YamlConfigurationLoader.load("classpath:rules/validation-rules.yaml")
            .createEngine();
    }
}

@Service
public class BusinessService {

    @Autowired
    private RulesEngine rulesEngine;

    @Autowired
    @Qualifier("validationEngine")
    private RulesEngine validationEngine;

    public void processData(Object data) {
        RuleResult validationResult = validationEngine.evaluate(data);
        if (validationResult.isSuccess()) {
            RuleResult businessResult = rulesEngine.evaluate(data);
            // Process results
        }
    }
}
```

Line-by-line explanation:

- **Line 1**: Spring configuration class annotation to define beans
- **Line 2**: Enable APEX rules engine integration with Spring Boot
- **Line 3**: Configuration class that sets up rules engine beans
- **Line 5**: Bean annotation to register this method as a Spring bean
- **Line 6**: Primary annotation makes this the default rules engine when multiple exist

- **Line 7**: Method that creates the primary rules engine bean
- **Line 8**: Load YAML configuration from classpath and create the rules engine
- **Line 9**: Return the configured rules engine instance
- **Line 12**: Bean annotation with specific name "validationEngine"
- **Line 13**: Method that creates a specialized validation rules engine
- **Line 14**: Load validation-specific YAML configuration from classpath
- **Line 15**: Return the validation rules engine instance
- **Line 19**: Service class annotation for Spring component scanning
- **Line 20**: Business service class that uses the rules engines
- **Line 22**: Autowired annotation to inject the primary rules engine
- **Line 23**: Private field to hold the primary rules engine reference
- **Line 25**: Autowired with qualifier to inject the specific validation engine
- **Line 26**: Qualifier annotation to specify which bean to inject by name
- **Line 27**: Private field to hold the validation rules engine reference
- **Line 29**: Business method that processes data using both rules engines
- **Line 30**: First evaluate data using the validation engine
- **Line 31**: Check if validation was successful before proceeding
- **Line 32**: If validation passed, evaluate using the primary business rules engine
- **Line 33**: Comment indicating where result processing logic would go

Microservices Integration

```
@RestController
@RequestMapping("/api/rules")
public class RulesController {

    @Autowired
    private RulesEngine rulesEngine;

    @PostMapping("/evaluate")
    public ResponseEntity<RuleResult> evaluateRules(@RequestBody Map<String, Object> data) {
        try {
            RuleResult result = rulesEngine.evaluate(data);
            return ResponseEntity.ok(result);
        } catch (RuleEngineException e) {
            return ResponseEntity.badRequest()
                .body(RuleResult.error("Rule evaluation failed: " + e.getMessage()));
        }
    }

    @GetMapping("/performance")
    public ResponseEntity<PerformanceSnapshot> getPerformanceMetrics() {
        PerformanceSnapshot snapshot = rulesEngine.getPerformanceMonitor().getSnapshot();
        return ResponseEntity.ok(snapshot);
    }
}
```

Batch Processing Integration

```
@Component
public class BatchRulesProcessor {

    @Autowired
    private RulesEngine rulesEngine;
```

```
    @EventListener
    public void processBatch(BatchProcessingEvent event) {
        List<Object> dataItems = event.getDataItems();

        // Parallel processing with rules engine
        List<RuleResult> results = dataItems.parallelStream()
            .map(rulesEngine::evaluate)
            .collect(Collectors.toList());

        // Process results
        results.forEach(this::handleResult);
    }

    private void handleResult(RuleResult result) {
        if (!result.isSuccess()) {
            // Handle validation failures
            log.warn("Rule validation failed: {}", result.getFailureReasons());
        }

        // Process enriched data
        Map<String, Object> enrichedData = result.getEnrichedData();
        // Continue processing...
    }
}
```

Error Handling and Recovery

Error Recovery Strategies

```
// Configure error recovery behavior
RulesEngineConfiguration config = new RulesEngineConfiguration.Builder()
    .withErrorRecovery(ErrorRecoveryStrategy.CONTINUE_ON_ERROR)
    .withMaxRetries(3)
    .withRetryDelay(Duration.ofMillis(100))
    .build();
```

Custom Error Handling

```
public class CustomErrorHandler implements RuleErrorHandler {

    @Override
    public void handleRuleError(Rule rule, Exception error, Object data) {
        log.error("Rule {} failed for data {}: {}",
            rule.getId(), data, error.getMessage());

        // Custom error handling logic
        if (error instanceof ValidationException) {
            // Handle validation errors
        } else if (error instanceof EnrichmentException) {
            // Handle enrichment errors
        }
    }

    @Override
    public boolean shouldContinueOnError(Rule rule, Exception error) {
        // Decide whether to continue processing other rules
        return !(error instanceof CriticalValidationException);
    }
```

```
}
```

Testing Strategies

Unit Testing Rules

```
@Test
public void testAgeValidationRule() {
    // Arrange
    Map<String, Object> data = Map.of("age", 25);

    // Act
    RuleResult result = rulesEngine.evaluate(data);

    // Assert
    assertTrue(result.isSuccess());
    assertFalse(result.hasFailures());
}

@Test
public void testEnrichmentFunctionality() {
    // Arrange
    Map<String, Object> data = Map.of("currency", "USD");

    // Act
    RuleResult result = rulesEngine.evaluate(data);

    // Assert
    assertEquals("US Dollar", result.getEnrichedData().get("currencyName"));
    assertEquals("North America", result.getEnrichedData().get("currencyRegion"));
}
```

Line-by-line explanation:

- **Line 1**: JUnit test annotation to mark this as a test method
- **Line 2**: Test method name that clearly describes what is being tested
- **Line 4**: Create test data with an age of 25 (above the minimum age requirement)
- **Line 7**: Execute the rules engine with the test data
- **Line 10**: Assert that the rule evaluation was successful (no failures)
- **Line 11**: Assert that there are no failure messages in the result
- **Line 14**: Second test method annotation
- **Line 15**: Test method for enrichment functionality
- **Line 17**: Create test data with a currency code "USD"
- **Line 20**: Execute the rules engine which should enrich the data with currency information
- **Line 23**: Assert that the enriched data contains the expected currency name
- **Line 24**: Assert that the enriched data contains the expected currency region

Integration Testing

```
@SpringBootTest
@TestPropertySource(properties = {
    "rules.config.path=classpath:test-rules.yaml"
})
public class RulesEngineIntegrationTest {
```

```
    @Autowired
    private RulesEngine rulesEngine;

    @Test
    public void testFullWorkflow() {
        // Test complete rule evaluation workflow
        Map<String, Object> testData = createTestData();
        RuleResult result = rulesEngine.evaluate(testData);

        // Verify results
        assertNotNull(result);
        assertTrue(result.isSuccess());
        assertFalse(result.getEnrichedData().isEmpty());
    }
}
```

Performance Optimization

Caching Strategies

```yaml
# Optimal caching configuration
enrichments:
  - id: "cached-lookup"
    lookup-config:
      lookup-dataset:
        cache-enabled: true
        cache-ttl-seconds: 3600
        cache-max-size: 1000
        preload-enabled: true
        cache-refresh-ahead: true
```

Line-by-line explanation:

- **Line 2**: Start of enrichments collection for performance-optimized configuration
- **Line 3**: Unique identifier for this cached lookup enrichment
- **Line 4**: Start of lookup configuration section
- **Line 5**: Start of dataset configuration with caching options
- **Line 6**: Enable caching to store lookup results in memory
- **Line 7**: Set cache time-to-live to 3600 seconds (1 hour) before data expires
- **Line 8**: Limit cache to maximum 1000 entries to control memory usage
- **Line 9**: Enable preloading to load all data into cache at startup
- **Line 10**: Enable refresh-ahead to refresh cache entries before they expire

Memory Management

```
// Configure memory-efficient processing
RulesEngineConfiguration config = new RulesEngineConfiguration.Builder()
    .withMemoryOptimization(true)
    .withMaxConcurrentEvaluations(100)
    .withResultCaching(false) // Disable for memory-constrained environments
    .build();
```

Monitoring and Alerting

```
// Set up performance monitoring
PerformanceMonitor monitor = rulesEngine.getPerformanceMonitor();
monitor.addThreshold("execution-time", Duration.ofMillis(100));
monitor.addThreshold("memory-usage", 50_000_000L); // 50MB

monitor.onThresholdExceeded((metric, value, threshold) -> {
    log.warn("Performance threshold exceeded: {} = {} (threshold: {})",
        metric, value, threshold);
    // Send alert
});
```

Comprehensive Configuration Examples

Financial Services Template

```
metadata:
  name: "Financial Services Rules"
  version: "2.0.0"
  description: "Comprehensive financial services validation and enrichment"
  domain: "Financial Services"
  tags: ["finance", "trading", "compliance"]

# Currency enrichment with comprehensive data
enrichments:
  - id: "currency-enrichment"
    type: "lookup-enrichment"
    condition: "['currency'] != null"
    lookup-config:
      lookup-dataset:
        type: "yaml-file"
        file-path: "datasets/currencies.yaml"
        key-field: "code"
        cache-enabled: true
        cache-ttl-seconds: 7200
        default-values:
          region: "Unknown"
          isActive: false
          decimalPlaces: 2
    field-mappings:
      - source-field: "name"
        target-field: "currencyName"
      - source-field: "region"
        target-field: "currencyRegion"
      - source-field: "isActive"
        target-field: "currencyActive"

  # Country enrichment for regulatory compliance
  - id: "country-enrichment"
    type: "lookup-enrichment"
    condition: "['countryCode'] != null"
    lookup-config:
      lookup-dataset:
        type: "yaml-file"
        file-path: "datasets/countries.yaml"
        key-field: "code"
        cache-enabled: true
    field-mappings:
```

```yaml
      - source-field: "name"
        target-field: "countryName"
      - source-field: "region"
        target-field: "region"
      - source-field: "regulatoryZone"
        target-field: "regulatoryZone"

rules:
  - id: "trade-amount-validation"
    name: "Trade Amount Validation"
    condition: "#amount != null && #amount > 0 && #amount <= 10000000"
    message: "Trade amount must be positive and not exceed 10M"
    severity: "ERROR"
    metadata:
      owner: "Trading Team"
      purpose: "Risk management"

  - id: "currency-validation"
    name: "Currency Code Validation"
    condition: "#currencyActive == true"
    message: "Currency must be active for trading"
    severity: "ERROR"
    depends-on: ["currency-enrichment"]
```

Line-by-line explanation:

- **Line 1**: Start of metadata section for the entire financial services rule set
- **Line 2**: Name of this comprehensive rule configuration
- **Line 3**: Version 2.0.0 indicating this is a mature, production-ready configuration
- **Line 4**: Description explaining this covers validation and enrichment for financial services
- **Line 5**: Business domain classification for organizational purposes
- **Line 6**: Tags for categorization and searching across multiple domains
- **Line 9**: Start of enrichments collection for data enhancement
- **Line 10**: Unique identifier for currency enrichment functionality
- **Line 11**: Type of enrichment - lookup-based data enhancement
- **Line 12**: Condition to run this enrichment only when currency field exists
- **Line 13**: Start of lookup configuration section
- **Line 14**: Start of dataset configuration
- **Line 15**: Dataset type - loads from YAML file
- **Line 16**: Path to the currencies dataset file
- **Line 17**: Field in the dataset that serves as the lookup key
- **Line 18**: Enable caching for performance optimization
- **Line 19**: Cache TTL of 7200 seconds (2 hours) for currency data
- **Line 20**: Start of default values for missing data
- **Line 21**: Default region when currency not found in dataset
- **Line 22**: Default active status when currency not found
- **Line 23**: Default decimal places when currency not found
- **Line 24**: Start of field mappings section
- **Line 25**: Map dataset "name" field to "currencyName" in enriched data
- **Line 26**: Target field name for currency name
- **Line 27**: Map dataset "region" field to "currencyRegion" in enriched data
- **Line 28**: Target field name for currency region
- **Line 29**: Map dataset "isActive" field to "currencyActive" in enriched data
- **Line 30**: Target field name for currency active status
- **Line 33**: Unique identifier for country enrichment functionality

- **Line 34**: Type of enrichment - lookup-based for country data
- **Line 35**: Condition to run this enrichment only when countryCode field exists
- **Line 36**: Start of lookup configuration for country data
- **Line 37**: Start of country dataset configuration
- **Line 38**: Dataset type - loads from YAML file
- **Line 39**: Path to the countries dataset file
- **Line 40**: Field in the dataset that serves as the lookup key
- **Line 41**: Enable caching for country lookup performance
- **Line 42**: Start of field mappings for country data
- **Line 43**: Map dataset "name" field to "countryName" in enriched data
- **Line 44**: Target field name for country name
- **Line 45**: Map dataset "region" field to "region" in enriched data
- **Line 46**: Target field name for country region
- **Line 47**: Map dataset "regulatoryZone" field to "regulatoryZone" in enriched data
- **Line 48**: Target field name for regulatory zone information
- **Line 50**: Start of rules collection for validation logic
- **Line 51**: Unique identifier for trade amount validation rule
- **Line 52**: Human-readable name for the trade amount rule
- **Line 53**: SpEL condition validating amount is not null, positive, and under 10 million
- **Line 54**: Error message when trade amount validation fails
- **Line 55**: Severity level set to ERROR for critical validation
- **Line 56**: Start of metadata for this rule
- **Line 57**: Team responsible for maintaining this rule
- **Line 58**: Purpose statement for risk management
- **Line 60**: Unique identifier for currency validation rule
- **Line 61**: Human-readable name for currency validation
- **Line 62**: SpEL condition checking that currency is active for trading
- **Line 63**: Error message when currency is not active
- **Line 64**: Severity level set to ERROR for critical validation
- **Line 65**: Dependency declaration - this rule depends on currency enrichment completing first

Multi-Dataset Enrichment Example

```yaml
# Complex scenario with multiple related datasets
enrichments:
  # Primary instrument enrichment
  - id: "instrument-enrichment"
    type: "lookup-enrichment"
    condition: "['instrumentId'] != null"
    lookup-config:
      lookup-dataset:
        type: "yaml-file"
        file-path: "datasets/instruments.yaml"
        key-field: "id"
        cache-enabled: true
    field-mappings:
      - source-field: "name"
        target-field: "instrumentName"
      - source-field: "type"
        target-field: "instrumentType"
      - source-field: "sector"
        target-field: "sector"

  # Sector-based risk enrichment (depends on instrument enrichment)
  - id: "sector-risk-enrichment"
```

```
    type: "lookup-enrichment"
    condition: "['sector'] != null"
    depends-on: ["instrument-enrichment"]
    lookup-config:
      lookup-dataset:
        type: "inline"
        key-field: "sector"
        data:
          - sector: "Technology"
            riskLevel: "HIGH"
            volatilityFactor: 1.5
          - sector: "Utilities"
            riskLevel: "LOW"
            volatilityFactor: 0.8
          - sector: "Healthcare"
            riskLevel: "MEDIUM"
            volatilityFactor: 1.2
    field-mappings:
      - source-field: "riskLevel"
        target-field: "sectorRisk"
      - source-field: "volatilityFactor"
        target-field: "volatilityFactor"

rules:
  - id: "high-risk-validation"
    name: "High Risk Instrument Check"
    condition: "#sectorRisk == 'HIGH' && #amount > 1000000"
    message: "High-risk instruments require additional approval for amounts > 1M"
    severity: "WARNING"
    depends-on: ["sector-risk-enrichment"]
```

Environment-Specific Configuration

```
# Development environment
metadata:
  name: "Development Rules"
  environment: "development"

enrichments:
  - id: "test-data-enrichment"
    type: "lookup-enrichment"
    lookup-config:
      lookup-dataset:
        type: "inline"
        key-field: "code"
        data:
          - code: "TEST001"
            name: "Test Instrument 1"
            type: "EQUITY"
          - code: "TEST002"
            name: "Test Instrument 2"
            type: "BOND"

---
# Production environment
metadata:
  name: "Production Rules"
  environment: "production"

enrichments:
  - id: "production-data-enrichment"
    type: "lookup-enrichment"
    lookup-config:
```

```
      lookup-dataset:
        type: "yaml-file"
        file-path: "datasets/production-instruments.yaml"
        key-field: "code"
        cache-enabled: true
        cache-ttl-seconds: 3600
        preload-enabled: true
```

Advanced Dataset Patterns

Hierarchical Dataset Structure

```
# datasets/product-hierarchy.yaml
data:
  - code: "EQUITY"
    name: "Equity Securities"
    category: "SECURITIES"
    subcategories:
      - code: "COMMON"
        name: "Common Stock"
        riskWeight: 1.0
      - code: "PREFERRED"
        name: "Preferred Stock"
        riskWeight: 0.8

  - code: "FIXED_INCOME"
    name: "Fixed Income Securities"
    category: "SECURITIES"
    subcategories:
      - code: "GOVT_BOND"
        name: "Government Bond"
        riskWeight: 0.2
      - code: "CORP_BOND"
        name: "Corporate Bond"
        riskWeight: 0.5
```

Time-Based Dataset Configuration

```
enrichments:
  - id: "time-sensitive-enrichment"
    type: "lookup-enrichment"
    lookup-config:
      lookup-dataset:
        type: "inline"
        key-field: "code"
        time-based: true
        effective-date-field: "effectiveDate"
        data:
          - code: "RATE001"
            rate: 0.05
            effectiveDate: "2024-01-01"
          - code: "RATE001"
            rate: 0.055
            effectiveDate: "2024-07-01"
```

Conditional Dataset Loading

```yaml
enrichments:
  - id: "conditional-enrichment"
    type: "lookup-enrichment"
    condition: "['region'] == 'US' && ['instrumentType'] == 'EQUITY'"
    lookup-config:
      lookup-dataset:
        type: "yaml-file"
        file-path: "datasets/us-equity-data.yaml"
        key-field: "symbol"

  - id: "alternative-enrichment"
    type: "lookup-enrichment"
    condition: "['region'] == 'EU' && ['instrumentType'] == 'EQUITY'"
    lookup-config:
      lookup-dataset:
        type: "yaml-file"
        file-path: "datasets/eu-equity-data.yaml"
        key-field: "isin"
```