# APEX - Complete User Guide

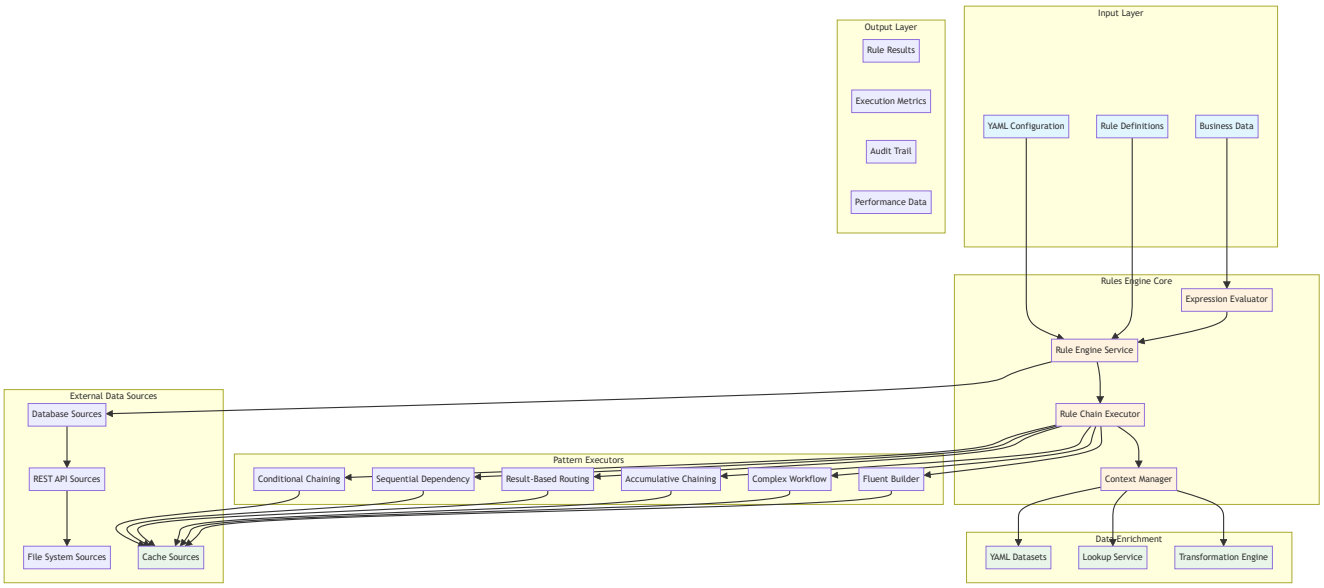
**Version:** 1.0 **Date:** 2025-07-30 **Author:** Mark Andrew Ray-Smith Cityline Ltd

# Overview

APEX (Advanced Processing Engine for eXpressions) is a comprehensive rule evaluation system built on Spring Expression Language (SpEL) with enterprise-grade external data source integration. It provides a progressive API design that scales from simple rule evaluation to complex business rule management systems with seamless data access capabilities.

## Key Capabilities

- **External Data Source Integration**: Connect to databases, REST APIs, file systems, and caches
- **YAML Dataset Enrichment**: Embed reference data directly in configuration files
- **Progressive API Design**: Three-layer API from simple to advanced use cases
- **Enterprise Features**: Connection pooling, health monitoring, caching, failover
- **High Performance**: Optimized for production workloads with comprehensive monitoring




## Core Capabilities

### Rule Evaluation

- **Three-Layer API Design**: Simple one-liner evaluation → Structured rule sets → Advanced rule chains
- **SpEL Expression Support**: Full Spring Expression Language capabilities with custom functions
- **Multiple Data Types**: Support for primitives, objects, collections, and complex nested structures
- **Context Management**: Rich evaluation context with variable propagation and result tracking

### Configuration Management

- **YAML Configuration**: External rule and dataset management with hot-reloading support

- **Rule Groups**: Organize related rules with execution control and priority management
- **Rule Chains**: Advanced patterns for nested rules and complex business logic workflows
- **Data Service Configuration**: Programmatic setup of data sources for rule evaluation
- **Metadata Support**: Enterprise metadata including business ownership, effective dates, and custom properties

#### Data Enrichment

- **YAML Dataset Enrichment**: Embed lookup datasets directly in configuration files
- **Multiple Enrichment Types**: Lookup enrichment, transformation enrichment, and custom processors
- **Caching Support**: High-performance in-memory caching with configurable strategies
- **External Integration**: Support for database lookups, REST API calls, and custom data sources

#### Enterprise Features

- **Performance Monitoring**: Execution time tracking, rule performance analytics, and bottleneck identification
- **Error Handling**: Comprehensive error management with detailed logging and graceful degradation
- **Audit Trail**: Complete execution history with rule results and context tracking
- **Security**: Input validation, expression sandboxing, and access control integration

#### Financial Services Support

- **OTC Derivatives Validation**: Specialized rules for financial instrument validation
- **Regulatory Compliance**: Support for MiFID II, EMIR, and Dodd-Frank requirements
- **Risk Assessment**: Multi-criteria risk scoring with weighted components
- **Trade Processing**: Complex workflow patterns for trade lifecycle management

#### Advanced Rule Patterns

- **Conditional Chaining**: Execute expensive rules only when conditions are met
- **Sequential Dependency**: Build processing pipelines where each stage uses previous results
- **Result-Based Routing**: Route to different rule sets based on intermediate results
- **Accumulative Chaining**: Build up scores across multiple criteria with weighted components
- **Complex Financial Workflow**: Multi-stage processing with dependencies and conditional execution
- **Fluent Rule Builder**: Complex decision trees with conditional branching logic

## Architecture Benefits

#### Developer Experience

- **Progressive Complexity**: Start simple and add complexity as needed
- **Type Safety**: Strong typing support with compile-time validation
- **Testing Support**: Comprehensive testing utilities and mock frameworks

#### Operations

- **Hot Configuration Reload**: Update rules without application restart
- **Performance Monitoring**: Built-in metrics and monitoring capabilities
- **Scalability**: Designed for high-throughput, low-latency environments

#### Business User Friendly

- **YAML Configuration**: Human-readable configuration format
- **Business Metadata**: Rich metadata support for business context

- **Version Control**: Configuration stored in Git with full change history
- **Documentation**: Self-documenting rules with descriptions and examples

# Quick Start (5 Minutes)

Welcome to APEX! This section will get you up and running quickly with three progressively more powerful approaches. Don't worry if some concepts seem unfamiliar at first - we'll explain everything step by step.

## Understanding the Basics

Before we dive in, let's understand what APEX does: it evaluates business rules against your data. Think of it as asking questions like "Is this customer old enough?" or "Does this transaction meet our requirements?" APEX uses expressions (written in Spring Expression Language or SpEL) to define these questions.

The `#` symbol in expressions refers to data you provide. For example, `#age >= 18` means "check if the age value is 18 or greater."

## 1. One-Liner Rule Evaluation (Simplest Approach)

This is the easiest way to get started. You can evaluate a single rule with just one line of code:

```
import dev.mars.rulesengine.core.api.Rules;

// Check if someone is an adult (age 18 or older)
boolean isAdult = Rules.check("#age >= 18", Map.of("age", 25)); // returns true

// Check if account has sufficient balance
boolean hasBalance = Rules.check("#balance > 1000", Map.of("balance", 500)); // returns false

// Working with objects instead of simple values
Customer customer = new Customer("John", 25, "john@example.com");
boolean valid = Rules.check("#data.age >= 18 && #data.email != null", customer); // returns true
```

**What's happening here:**

- `Rules.check()` is a static method that evaluates one rule
- The first parameter is your rule expression (the question you're asking)
- The second parameter is your data (either a Map or an object)
- It returns true/false based on whether the rule passes

## 2. Template-Based Rules (Structured Approach)

When you need multiple related rules, templates provide a cleaner approach:

```
import dev.mars.rulesengine.core.api.RuleSet;

// Create a set of validation rules using pre-built templates
RulesEngine validation = RuleSet.validation()
    .ageCheck(18)           // Must be 18 or older
    .emailRequired()        // Must have an email address
    .balanceMinimum(1000)   // Must have at least $1000 balance
    .build();
```

```
// Validate your customer data against all rules at once
ValidationResult result = validation.validate(customer);

// Check the results
if (result.isValid()) {
    System.out.println("Customer passed all validations!");
} else {
    System.out.println("Validation failed: " + result.getFailureMessages());
}
```

**What's happening here:**

- `RuleSet.validation()` creates a builder for common validation scenarios
- Each method (like `ageCheck()`) adds a pre-configured rule
- `build()` creates the final rules engine
- `validate()` runs all rules and gives you a comprehensive result

## 3. YAML Configuration (Most Flexible Approach)

For complex scenarios or when non-developers need to modify rules, YAML configuration is ideal. This approach separates your business logic from your code.

First, create a `rules.yaml` file that defines your business rules:

```
# This section provides information about your rule set
metadata:
  name: "Customer Validation Rules"
  version: "1.0.0"

# Define your business rules here
rules:
  - id: "age-check"                                    # Unique identifier
    name: "Age Validation"                             # Human-readable name
    condition: "#data.age >= 18"                       # The actual rule logic
    message: "Customer must be at least 18 years old"  # Error message if rule fails

  - id: "email-check"
    name: "Email Validation"
    condition: "#data.email != null && #data.email.contains('@')"
    message: "Valid email address is required"

# Enrichments add extra data to your objects during rule evaluation
enrichments:
  - id: "status-enrichment"                    # Unique identifier for this enrichment
    type: "lookup-enrichment"                  # Type of enrichment (lookup from a dataset)
    condition: "['statusCode'] != null"        # Only enrich if statusCode exists
    lookup-config:
      lookup-dataset:                          # The data to look up from
        type: "inline"                         # Data is defined right here in the file
        key-field: "code"                      # Field to match against
        data:                                  # The actual lookup data
          - code: "A"
            name: "Active"
            description: "Active customer"
          - code: "I"
            name: "Inactive"
            description: "Inactive customer"
    field-mappings:                            # How to add the looked-up data to your object
      - source-field: "name"                   # Take the "name" from lookup data
        target-field: "statusName"             # Add it as "statusName" to your object
```

```
      - source-field: "description"
        target-field: "statusDescription"
```

Now load and use this configuration in your Java code:

```
// Load the YAML configuration file
RulesEngineConfiguration config = YamlConfigurationLoader.load("rules.yaml");
RulesEngine engine = new RulesEngine(config);

// Prepare your data for evaluation
Map<String, Object> data = Map.of(
    "age", 25,                    // Customer's age
    "email", "john@example.com",  // Customer's email
    "statusCode", "A"             // Customer's status code (will be enriched)
);

// Evaluate all rules and enrichments
RuleResult result = engine.evaluate(data);

// Check what happened
if (result.isSuccess()) {
    System.out.println("All rules passed!");

    // Access enriched data
    String statusName = (String) result.getEnrichedData().get("statusName");
    System.out.println("Customer status: " + statusName); // Prints "Active"
} else {
    System.out.println("Some rules failed: " + result.getFailureMessages());
}
```

**What's happening here:**

- The YAML file defines your business logic separately from your code
- `YamlConfigurationLoader.load()` reads and parses the YAML file
- The rules engine evaluates both rules and enrichments automatically
- Enrichments add extra information to your data (like looking up "Active" from status code "A")
- You get back a comprehensive result with both validation outcomes and enriched data

### Which Approach Should You Use?

- **One-liner**: Perfect for simple, one-off rule checks
- **Template-based**: Great for common validation scenarios with multiple related rules
- **YAML configuration**: Best for complex business logic, when rules change frequently, or when business users need to modify rules

You can start with the one-liner approach and gradually move to more sophisticated approaches as your needs grow!

## Core Concepts

Understanding these three core concepts will help you make the most of APEX. Think of them as the building blocks for creating intelligent business logic.

### Rules: Your Business Logic

Rules are the heart of APEX - they define the questions you want to ask about your data. Each rule is like a business requirement written in a way the computer can understand.

**Anatomy of a Rule:**

```yaml
rules:
  - id: "trade-amount-validation"                          # Unique identifier for this rule
    name: "Trade Amount Validation"                        # Human-readable name
    condition: "#amount > 0 && #amount <= 1000000"         # The actual business logic
    message: "Trade amount must be between 0 and 1,000,000"  # What to show if the rule fails
    severity: "ERROR"                                      # How serious is a failure?
    tags: ["financial", "validation"]                      # Categories for organization
```

**Breaking down the condition:**

- `#amount > 0` means "the amount must be greater than zero"
- `&&` means "AND" (both conditions must be true)
- `#amount <= 1000000` means "the amount must be less than or equal to 1,000,000"
- Together: "The amount must be positive and not exceed 1 million"

**Common rule patterns:**

- Validation: `#age >= 18` (must be 18 or older)
- Range checking: `#score >= 0 && #score <= 100` (score between 0-100)
- Required fields: `#email != null` (email must exist)
- Pattern matching: `#email.contains('@')` (email must contain @)

## Enrichments: Adding Smart Data

Enrichments automatically add related information to your data during rule evaluation. Think of them as smart lookups that happen behind the scenes.

**Why use enrichments?** Instead of just having a currency code like "USD", enrichments can automatically add the full name "US Dollar" and region "North America" to your data.

```yaml
enrichments:
  - id: "currency-enrichment"                  # Unique identifier
    type: "lookup-enrichment"                  # Type of enrichment (lookup from data)
    condition: "['currency'] != null"          # Only enrich if currency code exists
    lookup-config:
      lookup-dataset:                          # Where to find the lookup data
        type: "inline"                         # Data is defined right here
        key-field: "code"                      # Field to match against (currency code)
        cache-enabled: true                    # Cache for better performance
        data:                                  # The actual lookup data
          - code: "USD"
            name: "US Dollar"
            region: "North America"
          - code: "EUR"
            name: "Euro"
            region: "Europe"
    field-mappings:                            # How to add the data to your object
      - source-field: "name"                   # Take "name" from lookup data
        target-field: "currencyName"           # Add as "currencyName" to your object
      - source-field: "region"
```

```
      target-field: "currencyRegion"
```

**What happens during enrichment:**

1. Your data has `currency: "USD"`
2. APEX looks up "USD" in the dataset
3. It finds the matching record with name "US Dollar" and region "North America"
4. It adds `currencyName: "US Dollar"` and `currencyRegion: "North America"` to your data
5. Your rules can now use these enriched fields

## Datasets: Your Reference Data

Datasets are collections of reference data that enrichments use for lookups. They're like lookup tables that contain additional information about codes, IDs, or other identifiers in your data.

**Two ways to organize your datasets:**

**Inline Datasets (Best for small, unique data)**

Use inline datasets when you have small amounts of data that are specific to one configuration file:

```
lookup-dataset:
  type: "inline"                    # Data is defined right here in this file
  key-field: "code"                 # Field to match against when looking up
  cache-enabled: true               # Keep data in memory for faster lookups
  cache-ttl-seconds: 3600          # Cache for 1 hour (3600 seconds)
  data:                            # The actual data
    - code: "USD"                  # This is the key field
      name: "US Dollar"            # Additional information
    - code: "EUR"
      name: "Euro"
```

**When to use inline datasets:**

- Small datasets (less than 50 records)
- Data that's unique to this specific configuration
- Simple lookup tables that won't be reused elsewhere

**External Dataset Files (Best for larger, reusable data)**

Use external files when you have larger datasets or data that multiple configurations might use:

```
lookup-dataset:
  type: "yaml-file"                          # Data comes from an external file
  file-path: "datasets/currencies.yaml"      # Path to the data file
  key-field: "code"                          # Field to match against
  cache-enabled: true                        # Cache for performance
```

Then create `datasets/currencies.yaml` :

```
data:
  - code: "USD"
```

```
    name: "US Dollar"
    symbol: "$"
    region: "North America"
  - code: "EUR"
    name: "Euro"
    symbol: "€"
    region: "Europe"
  - code: "GBP"
    name: "British Pound"
    symbol: "£"
    region: "Europe"
```

**When to use external dataset files:**

- Larger datasets (50+ records)
- Data that multiple configurations need to share
- Data that changes independently of your rule configurations
- When you want to keep your main configuration file clean and focused

# YAML Configuration Guide

YAML (Yet Another Markup Language) is a human-readable format for configuration files. Don't worry if you're new to YAML - it's designed to be easy to read and write. Think of it as a structured way to organize information, similar to how you might organize information in an outline.

## Understanding YAML Basics

YAML uses indentation (spaces) to show relationships between items. Here are the key concepts:

- **Indentation matters**: Use spaces (not tabs) to show hierarchy
- **Lists**: Items that start with a dash ( `-` )
- **Key-value pairs**: `key: value`
- **Nested structures**: Indent to show items belong together

## Configuration Structure

Every APEX configuration file follows this basic structure:

```yaml
# Metadata section: Information about this configuration file
metadata:
  name: "Configuration Name"              # What this configuration does
  version: "1.0.0"                       # Version for tracking changes
  description: "Configuration description" # Detailed explanation
  author: "Team Name"                     # Who created/maintains this
  created: "2024-01-15"                  # When it was created
  last-modified: "2024-07-26"           # Last update date
  tags: ["tag1", "tag2"]                # Categories for organization

# Rules section: Your business logic
rules:
  # Individual rule definitions go here
  # Each rule defines a condition to check

# Enrichments section: Data enhancement
enrichments:
```

```
  # Enrichment definitions go here
  # Each enrichment adds data to your objects

# Rule groups section: Organized rule collections
rule-groups:
  # Rule group definitions go here
  # Groups let you organize related rules together
```

**Why organize it this way?**

- **Metadata**: Helps you track and document your configurations
- **Rules**: Contains your business logic and validation requirements
- **Enrichments**: Automatically adds useful information to your data
- **Rule Groups**: Organizes related rules for better management

## Rule Configuration

Rules are where you define your business logic. Each rule is like a question you're asking about your data. Here's how to configure them:

```
rules:
  - id: "unique-rule-id"                    # Required: Unique identifier (like a name tag)
    name: "Human Readable Name"             # Required: What this rule does in plain English
    condition: "#data.field > 100"          # Required: The actual business logic to check
    message: "Validation message"           # Optional: What to show if the rule fails
    severity: "ERROR"                       # Optional: How serious is a failure? (ERROR, WARNING, INFO)
    enabled: true                           # Optional: Turn this rule on/off (default: true)
    tags: ["validation", "business"]        # Optional: Categories for organization
    metadata:                               # Optional: Additional information for governance
      owner: "Business Team"               # Who owns/maintains this rule
      domain: "Finance"                    # What business area it belongs to
      purpose: "Regulatory compliance"     # Why this rule exists
```

**Understanding each part:**

- **id**: A unique name for this rule (like "customer-age-check"). Use descriptive names that make sense to your team.
- **name**: A human-friendly description that anyone can understand (like "Customer Age Validation").
- **condition**: The actual business logic using SpEL expressions. Common patterns:
  - `#age >= 18` (age must be 18 or older)
  - `#amount > 0 && #amount <= 1000` (amount between 0 and 1000)
  - `#email != null && #email.contains('@')` (email must exist and contain @)
- **message**: What users see when the rule fails. Make it helpful and actionable.
- **severity**: How important is this rule?
  - `ERROR` : Critical - must be fixed
  - `WARNING` : Important - should be reviewed
  - `INFO` : Informational - good to know
- **enabled**: Allows you to temporarily turn rules on/off without deleting them.
- **tags**: Help organize and filter rules. Use consistent tags across your organization.
- **metadata**: Additional information for governance, documentation, and audit trails.

## Enrichment Configuration

Enrichments automatically add related information to your data. Think of them as smart lookups that happen automatically during rule evaluation.

```yaml
enrichments:
  - id: "enrichment-id"                    # Required: Unique identifier for this enrichment
    type: "lookup-enrichment"              # Required: Type of enrichment (lookup is most common)
    condition: "['field'] != null"         # Optional: Only enrich if this condition is true
    enabled: true                          # Optional: Turn this enrichment on/off (default: true)
    lookup-config:                         # Configuration for the lookup process
      lookup-dataset:                      # Where to find the lookup data
        type: "inline"                     # Data source type: "inline" or "yaml-file"
        key-field: "lookupKey"             # Field to match against in your data
        cache-enabled: true                # Keep lookup data in memory for speed
        cache-ttl-seconds: 3600           # How long to cache (1 hour = 3600 seconds)
        default-values:                    # What to use if no match is found
          defaultField: "defaultValue"
        data:                              # The actual lookup data (for inline type)
          - lookupKey: "key1"              # This is what we match against
            field1: "value1"               # Additional data to add
            field2: "value2"
    field-mappings:                        # How to add the looked-up data to your object
      - source-field: "field1"             # Take this field from the lookup data
        target-field: "enrichedField1"     # Add it to your object with this name
      - source-field: "field2"
        target-field: "enrichedField2"
```

**Understanding enrichment flow:**

1. **Check condition**: If specified, only enrich when the condition is true
2. **Find matching data**: Look up the key value in your dataset
3. **Map fields**: Copy specified fields from the lookup data to your object
4. **Use defaults**: If no match found, use default values (if configured)

**Example in action:**

- Your data has: `{statusCode: "A"}`
- Lookup dataset has: `{code: "A", name: "Active", description: "Customer is active"}`
- Field mappings copy `name` to `statusName` and `description` to `statusDescription`
- Result: Your data now has: `{statusCode: "A", statusName: "Active", statusDescription: "Customer is active"}`

**Common use cases:**

- Convert codes to human-readable names (status codes, country codes, etc.)
- Add regional information based on location codes
- Enrich product data with category information
- Add calculated fields based on lookup tables

# Dataset Enrichment

Dataset enrichment is one of APEX's most powerful features. It automatically adds related information to your data during rule evaluation, transforming simple codes into rich, meaningful data.

## Understanding Dataset Enrichment

Imagine you have customer data with just a status code like "A". Dataset enrichment can automatically add the full status name "Active" and description "Customer account is active and in good standing" to your data. This happens transparently during rule evaluation, so your rules can work with both the original code and the enriched information.

## When to Use Dataset Enrichment

Dataset enrichment works best for reference data - information that helps explain or categorize your main data.

**Perfect candidates for dataset enrichment:**

- **Currency codes and names**: "USD" → "US Dollar", "EUR" → "Euro"
- **Country codes and regions**: "US" → "United States", "North America"
- **Status codes and descriptions**: "A" → "Active", "Customer account is active"
- **Product categories**: "ELEC" → "Electronics", "Consumer electronics category"
- **Reference data that changes infrequently**: Data that's stable over time
- **Small to medium datasets**: Less than 1000 records for optimal performance

**Not suitable for dataset enrichment:**

- **Large datasets**: More than 1000 records (use external data sources instead)
- **Frequently changing data**: Data that updates multiple times per day
- **Data requiring complex business logic**: Calculations or complex transformations
- **Real-time data from external systems**: Live data that needs fresh API calls

**Why these limitations?** Dataset enrichment loads all data into memory for fast lookups. This works great for small, stable reference data but isn't efficient for large or frequently changing datasets.

## Dataset Types

### 1. Inline Datasets

Best for small, unique datasets:

```yaml
lookup-dataset:
  type: "inline"
  key-field: "code"
  data:
    - code: "A"
      name: "Active"
    - code: "I"
      name: "Inactive"
```

### 2. External YAML Files

Best for reusable datasets:

Create `datasets/statuses.yaml` :

```yaml
data:
  - code: "A"
    name: "Active"
    description: "Active status"
  - code: "I"
    name: "Inactive"
```

```
    description: "Inactive status"
```

Reference in configuration:

```
lookup-dataset:
  type: "yaml-file"
  file-path: "datasets/statuses.yaml"
  key-field: "code"
```

### Performance Optimization

```
lookup-dataset:
  type: "inline"
  key-field: "code"
  cache-enabled: true
  cache-ttl-seconds: 3600
  preload-enabled: true
  data:
    # Dataset entries
```

# External Data Source Integration

## Overview

The SpEL Rules Engine provides comprehensive external data source integration, enabling seamless access to databases, REST APIs, file systems, and caches through a unified interface. This enterprise-grade integration supports advanced features like connection pooling, health monitoring, caching, and automatic failover.

## Supported Data Source Types

#### 1. Database Sources

Connect to relational databases with full connection pooling support:

```
dataSources:
  - name: "user-database"
    type: "database"
    sourceType: "postgresql"
    enabled: true

    connection:
      host: "localhost"
      port: 5432
      database: "myapp"
      username: "app_user"
      password: "${DB_PASSWORD}"
      maxPoolSize: 20
      minPoolSize: 5

    queries:
      getUserById: "SELECT * FROM users WHERE id = :id"
      getAllUsers: "SELECT * FROM users ORDER BY created_at DESC"
```

```
    parameterNames:
      - "id"

    cache:
      enabled: true
      ttlSeconds: 300
      maxSize: 1000
```

**Supported Databases**: PostgreSQL, MySQL, Oracle, SQL Server, H2

#### 2. REST API Sources

Integrate with HTTP/HTTPS APIs with various authentication methods:

```
dataSources:
  - name: "external-api"
    type: "rest-api"
    enabled: true

    connection:
      baseUrl: "https://api.example.com/v1"
      timeout: 10000
      retryAttempts: 3

    authentication:
      type: "bearer"
      token: "${API_TOKEN}"

    endpoints:
      getUser: "/users/{userId}"
      searchUsers: "/users/search?q={query}"

    parameterNames:
      - "userId"
      - "query"

    circuitBreaker:
      enabled: true
      failureThreshold: 5
      recoveryTimeout: 30000
```

**Authentication Types**: Bearer tokens, API keys, Basic auth, OAuth2

#### 3. File System Sources

Process various file formats with automatic parsing:

```
dataSources:
  - name: "data-files"
    type: "file-system"
    enabled: true

    connection:
      basePath: "/data/files"
      filePattern: "*.csv"
      watchForChanges: true
      encoding: "UTF-8"

    fileFormat:
```

```
      type: "csv"
      hasHeaderRow: true
      delimiter: ","

      columnMappings:
        "customer_id": "id"
        "customer_name": "name"

    parameterNames:
      - "filename"
```

**Supported Formats**: CSV, JSON, XML, fixed-width, plain text

#### 4. Cache Sources

High-performance in-memory caching:

```
dataSources:
  - name: "app-cache"
    type: "cache"
    sourceType: "memory"
    enabled: true

    cache:
      enabled: true
      maxSize: 10000
      ttlSeconds: 1800
      evictionPolicy: "LRU"
```

## Using External Data Sources

### Basic Usage

```
// Initialize configuration service
DataSourceConfigurationService configService = DataSourceConfigurationService.getInstance();
YamlRuleConfiguration yamlConfig = loadConfiguration("data-sources.yaml");
configService.initialize(yamlConfig);

// Get data source
ExternalDataSource userDb = configService.getDataSource("user-database");

// Execute queries
Map<String, Object> parameters = Map.of("id", 123);
List<Object> results = userDb.query("getUserById", parameters);

// Get single result
Object user = userDb.queryForObject("getUserById", parameters);
```

### Advanced Usage with Load Balancing

```
// Get manager for advanced operations
DataSourceManager manager = configService.getDataSourceManager();

// Load balancing across multiple sources
ExternalDataSource source = manager.getDataSourceWithLoadBalancing(DataSourceType.DATABASE);

// Failover query across healthy sources
```

```
List<Object> results = manager.queryWithFailover(DataSourceType.DATABASE, "getAllUsers", Collections.emptyMap());

// Async operations
CompletableFuture<List<Object>> future = manager.queryAsync("user-database", "getAllUsers", Collections.emptyMap());
List<Object> users = future.get(10, TimeUnit.SECONDS);
```

## Enterprise Features

### Health Monitoring

```
healthCheck:
  enabled: true
  intervalSeconds: 30
  timeoutSeconds: 5
  failureThreshold: 3
  query: "SELECT 1"
```

### Environment-Specific Configuration

```
environments:
  development:
    dataSources:
      - name: "user-database"
        connection:
          host: "localhost"
          maxPoolSize: 5

  production:
    dataSources:
      - name: "user-database"
        connection:
          host: "prod-db.example.com"
          maxPoolSize: 50
```

### Monitoring and Statistics

```
// Get performance metrics
DataSourceMetrics metrics = dataSource.getMetrics();
System.out.println("Success rate: " + metrics.getSuccessRate());
System.out.println("Average response time: " + metrics.getAverageResponseTime());

// Registry statistics
RegistryStatistics stats = registry.getStatistics();
System.out.println("Health percentage: " + stats.getHealthPercentage());
```

## Integration with Rules

External data sources integrate seamlessly with the rules engine:

```
# Use data sources in rule conditions
rules:
  - id: "user-validation"
    condition: "dataSource('user-database').queryForObject('getUserById', {'id': #userId}) != null"
    message: "User exists in database"
```

```yaml
# Use in enrichments
enrichments:
  - id: "user-enrichment"
    type: "data-source-enrichment"
    data-source: "user-database"
    query: "getUserById"
    parameters:
      id: "#userId"
    field-mappings:
      - source-field: "name"
        target-field: "userName"
      - source-field: "email"
        target-field: "userEmail"
```

## Best Practices

### Configuration Management

- Use environment variables for sensitive data
- Implement environment-specific overrides
- Validate configurations before deployment
- Use meaningful, descriptive names

### Performance Optimization

- Configure appropriate connection pool sizes
- Enable caching for frequently accessed data
- Use circuit breakers for external APIs
- Monitor performance metrics regularly

### Security

- Always use SSL/TLS in production
- Implement proper access controls
- Use strong authentication methods
- Encrypt sensitive configuration data

### Error Handling

- Configure health checks appropriately
- Implement retry logic with exponential backoff
- Use graceful degradation strategies
- Monitor and alert on failures

For detailed configuration guides, see:

- Database Configuration Guide
- REST API Configuration Guide
- File System Configuration Guide
- Best Practices Guide

# Rule Groups Configuration

## Overview

Rule Groups provide a way to organize related rules and control their execution as a logical unit. Rules within a group can be combined using AND or OR operators, allowing for complex validation scenarios where multiple conditions must be met (AND) or where any one of several conditions is sufficient (OR).

## Key Features

- **Logical Operators**: Combine rules with AND or OR operators
- **Priority Management**: Control execution order within groups
- **Category Support**: Organize groups by business domain
- **Sequence Control**: Define rule execution order within groups
- **Metadata Support**: Rich metadata for governance and audit trails

## Programmatic Rule Group Creation

### Using RuleGroupBuilder

```
// Create a rule group with AND operator
RuleGroup validationGroup = new RuleGroupBuilder()
    .withId("validation-group")
    .withName("Customer Validation Group")
    .withDescription("Complete customer validation checks")
    .withCategory("customer-validation")
    .withPriority(10)
    .withAndOperator() // All rules must pass
    .build();

// Create a rule group with OR operator
RuleGroup eligibilityGroup = new RuleGroupBuilder()
    .withId("eligibility-group")
    .withName("Customer Eligibility Group")
    .withDescription("Customer eligibility checks")
    .withCategory("customer-eligibility")
    .withPriority(20)
    .withOrOperator() // Any rule can pass
    .build();
```

### Using RulesEngineConfiguration

```
RulesEngineConfiguration config = new RulesEngineConfiguration();

// Create rule group with AND operator
RuleGroup andGroup = config.createRuleGroupWithAnd(
    "RG001",                          // Group ID
    new Category("validation", 10),   // Category
    "Validation Checks",              // Name
    "All validation rules must pass", // Description
    10                                // Priority
);

// Create rule group with OR operator
RuleGroup orGroup = config.createRuleGroupWithOr(
    "RG002",                          // Group ID
    new Category("eligibility", 20),  // Category
    "Eligibility Checks",             // Name
    "Any eligibility rule can pass",  // Description
```

```
    20                              // Priority
);

// Create multi-category rule group
Set<String> categories = Set.of("validation", "compliance");
RuleGroup multiCategoryGroup = config.createRuleGroupWithAnd(
    "RG003",                        // Group ID
    categories,                     // Multiple categories
    "Compliance Validation",        // Name
    "Compliance and validation checks", // Description
    30                              // Priority
);
```

### Adding Rules to Groups

```
// Create individual rules
Rule ageRule = config.rule("age-check")
    .withName("Age Validation")
    .withCondition("#age >= 18")
    .withMessage("Customer must be at least 18")
    .build();

Rule emailRule = config.rule("email-check")
    .withName("Email Validation")
    .withCondition("#email != null && #email.contains('@')")
    .withMessage("Valid email required")
    .build();

Rule incomeRule = config.rule("income-check")
    .withName("Income Validation")
    .withCondition("#income >= 25000")
    .withMessage("Minimum income requirement")
    .build();

// Add rules to AND group (all must pass)
andGroup.addRule(ageRule, 1);    // Execute first
andGroup.addRule(emailRule, 2);  // Execute second
andGroup.addRule(incomeRule, 3); // Execute third

// Add rules to OR group (any can pass)
orGroup.addRule(ageRule, 1);
orGroup.addRule(incomeRule, 2);

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

## YAML Rule Group Configuration

### Basic Rule Group Configuration

```
metadata:
  name: "Customer Processing Rules"
  version: "1.0.0"

rules:
  - id: "age-validation"
    name: "Age Check"
    condition: "#age >= 18"
    message: "Customer must be at least 18"
```

```yaml
  - id: "email-validation"
    name: "Email Check"
    condition: "#email != null && #email.contains('@')"
    message: "Valid email address required"

  - id: "income-validation"
    name: "Income Check"
    condition: "#income >= 25000"
    message: "Minimum income of $25,000 required"

rule-groups:
  - id: "customer-validation"
    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"
      purpose: "Customer data validation"
```

### Advanced Rule Group Configuration

```yaml
rule-groups:
  # AND group - all rules must pass
  - id: "strict-validation"
    name: "Strict Validation Group"
    description: "All validation rules must pass"
    category: "validation"
    categories: ["validation", "compliance"]  # Multiple categories
    priority: 10
    enabled: true
    stop-on-first-failure: true  # Stop on first failure for efficiency
    parallel-execution: false    # Sequential execution
    rule-ids:
      - "age-validation"
      - "email-validation"
      - "income-validation"
    tags: ["strict", "validation", "required"]
    metadata:
      owner: "Compliance Team"
      business-domain: "Customer Onboarding"
      created-by: "compliance.admin@company.com"
    execution-config:
      timeout-ms: 5000
      retry-count: 3
      circuit-breaker: true

  # OR group - any rule can pass with advanced rule references
  - id: "eligibility-check"
    name: "Customer Eligibility Check"
    description: "Customer meets at least one eligibility criteria"
    category: "eligibility"
    priority: 20
    enabled: true
```

```yaml
    stop-on-first-failure: false # Continue even if one rule fails
    parallel-execution: true     # Parallel execution for performance
    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
      - rule-id: "high-value-customer"
        sequence: 3
        enabled: true
        override-priority: 15
    metadata:
      owner: "Business Team"
      purpose: "Customer eligibility determination"
```

### Complete YAML Configuration Reference

```yaml
rule-groups:
  - id: "complete-example"                    # Required: Unique identifier
    name: "Complete Rule Group Example"       # Required: Human-readable name
    description: "Shows all available options" # Optional: Description

    # Category Configuration
    category: "validation"                    # Single category
    categories: ["validation", "compliance"] # Multiple categories (alternative to category)

    # Execution Configuration
    priority: 10                              # Optional: Execution priority (default: 100)
    enabled: true                             # Optional: Enable/disable group (default: true)
    stop-on-first-failure: false             # Optional: Stop on first rule failure (default: false)
    parallel-execution: false                # Optional: Execute rules in parallel (default: false)

    # Rule References - Option 1: Simple rule IDs
    rule-ids:
      - "rule-1"
      - "rule-2"
      - "rule-3"

    # Rule References - Option 2: Advanced rule references with control
    rule-references:
      - rule-id: "advanced-rule-1"
        sequence: 1                           # Optional: Execution sequence
        enabled: true                         # Optional: Enable/disable this rule (default: true)
        override-priority: 5                  # Optional: Override rule's default priority
      - rule-id: "advanced-rule-2"
        sequence: 2
        enabled: false                        # Disabled rule
        override-priority: 10

    # Metadata and Tags
    tags: ["validation", "customer", "strict"] # Optional: Tags for categorization
    metadata:                                 # Optional: Custom metadata
      owner: "Team Name"
      business-domain: "Domain"
      created-by: "user@company.com"
      purpose: "Business purpose"
      custom-field: "custom-value"

    # Advanced Execution Configuration
```

```yaml
  execution-config:                    # Optional: Advanced execution settings
    timeout-ms: 5000                   # Optional: Timeout in milliseconds
    retry-count: 3                     # Optional: Number of retries on failure
    circuit-breaker: true              # Optional: Enable circuit breaker pattern
```

### YAML Configuration Properties Reference

| Property | Type | Required | Default | Description |
|---|---|---|---|---|
| `id` | String | Yes | - | Unique identifier for the rule group |
| `name` | String | Yes | - | Human-readable name for the rule group |
| `description` | String | No | "" | Description of what the rule group does |
| `category` | String | No | - | Single category for the rule group |
| `categories` | List | No | - | Multiple categories (alternative to `category`) |
| `priority` | Integer | No | 100 | Execution priority (lower = higher priority) |
| `enabled` | Boolean | No | true | Whether the rule group is enabled |
| `stop-on-first-failure` | Boolean | No | false | Stop execution on first rule failure (AND logic) |
| `parallel-execution` | Boolean | No | false | Execute rules in parallel for performance |
| `rule-ids` | List | No | - | Simple list of rule IDs to include |
| `rule-references` | List | No | - | Advanced rule references with control options |
| `tags` | List | No | - | Tags for categorization and filtering |
| `metadata` | Map<String, Object> | No | - | Custom metadata for governance |
| `execution-config` | ExecutionConfig | No | - | Advanced execution configuration |

### Rule Reference Properties

| Property | Type | Required | Default | Description |
|---|---|---|---|---|
| `rule-id` | String | Yes | - | ID of the rule to reference |
| `sequence` | Integer | No | - | Execution sequence within the group |
| `enabled` | Boolean | No | true | Whether this rule is enabled in the group |
| `override-priority` | Integer | No | - | Override the rule's default priority |

### Execution Config Properties

| Property | Type | Required | Default | Description |
|---|---|---|---|---|
| `timeout-ms` | Long | No | - | Timeout in milliseconds for rule group execution |
| `retry-count` | Integer | No | - | Number of retries on execution failure |
| `circuit-breaker` | Boolean | No | false | Enable circuit breaker pattern for resilience |

## Rule Group Execution Behavior

### AND Groups (All Rules Must Pass)

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

// Execution logic:
// - All rules must evaluate to true
// - If any rule fails, the entire group fails
// - Execution can stop on first failure (configurable)
// - Result is true only if ALL rules pass

Map<String, Object> data = Map.of("age", 25, "email", "test@example.com", "income", 30000);
RuleResult result = engine.executeRuleGroup(andGroup, data);
// Returns true only if age >= 18 AND email is valid AND income >= 25000
```

### OR Groups (Any Rule Can Pass)

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

// Execution logic:
// - Any rule can evaluate to true for group to pass
// - If one rule passes, the entire group passes
// - Execution can stop on first success (configurable)
// - Result is true if ANY rule passes

Map<String, Object> data = Map.of("age", 16, "email", null, "income", 30000);
RuleResult result = engine.executeRuleGroup(orGroup, data);
// Returns true if income >= 25000 (even though age and email fail)
```

## Integration with Rules Engine

```
// Create rules engine with rule groups
RulesEngineConfiguration config = new RulesEngineConfiguration();

// Add individual rules and rule groups
config.rule("basic-rule").withCondition("#value > 0").build();
config.createRuleGroupWithAnd("validation-group", category, "Validation", "All validations", 10);

RulesEngine engine = new RulesEngine(config);

// Execute all rules and groups for a category
List<RuleResult> results = engine.executeRulesForCategory(category, data);

// Execute specific rule group
RuleGroup group = config.getRuleGroupById("validation-group");
```

```
RuleResult groupResult = engine.executeRuleGroup(group, data);
```

### Best Practices

#### Rule Group Organization

- Use AND groups for validation scenarios where all conditions must be met
- Use OR groups for eligibility scenarios where any condition is sufficient
- Group related rules by business domain or functional area
- Use meaningful IDs and names for easy identification

#### Performance Optimization

- Enable `stop-on-first-failure` for AND groups to improve performance
- Enable `stop-on-first-success` for OR groups when appropriate
- Use `parallel-execution` for independent rules that can run concurrently
- Order rules by execution cost (fastest first) for optimal performance

#### Governance and Maintenance

- Include comprehensive metadata for audit trails
- Use tags for categorization and filtering
- Document rule group purpose and business logic
- Version control rule group configurations

## Data Service Configuration

### Overview

Data service configuration provides a programmatic way to set up data sources that rules can reference for lookups, enrichments, and data-driven rule evaluation. This approach is particularly useful for testing, demonstrations, and scenarios where you need to configure mock or custom data sources.

### DataServiceManager

The `DataServiceManager` serves as the central orchestration point for all data operations:

```
// Initialize with mock data sources
DataServiceManager dataManager = new DataServiceManager();
dataManager.initializeWithMockData();

// Load custom data sources
dataManager.loadDataSource(new CustomDataSource("ProductsSource", "products"));

// Request data for rule evaluation
List<Product> products = dataManager.requestData("products");
Customer customer = dataManager.requestData("customer");
```

### DemoDataServiceManager

For demonstration and testing purposes, the `DemoDataServiceManager` extends the base manager with pre-configured mock data:

```
public class DemoDataServiceManager extends DataServiceManager {

    @Override
    public DataServiceManager initializeWithMockData() {
        // Create and load mock data sources for various data types
        loadDataSource(new MockDataSource("ProductsDataSource", "products"));
        loadDataSource(new MockDataSource("InventoryDataSource", "inventory"));
        loadDataSource(new MockDataSource("CustomerDataSource", "customer"));
        loadDataSource(new MockDataSource("TemplateCustomerDataSource", "templateCustomer"));
        loadDataSource(new MockDataSource("LookupServicesDataSource", "lookupServices"));
        loadDataSource(new MockDataSource("SourceRecordsDataSource", "sourceRecords"));

        // Add data sources for dynamic matching
        loadDataSource(new MockDataSource("MatchingRecordsDataSource", "matchingRecords"));
        loadDataSource(new MockDataSource("NonMatchingRecordsDataSource", "nonMatchingRecords"));

        return this;
    }
}
```

## MockDataSource Implementation

The `MockDataSource` provides pre-populated test data for various business scenarios:

```
public class MockDataSource implements DataSource {
    private final String name;
    private final String dataType;
    private final Map<String, Object> dataStore = new HashMap<>();

    public MockDataSource(String name, String dataType) {
        this.name = name;
        this.dataType = dataType;
        initializeData(); // Populate with test data
    }

    @Override
    public <T> T getData(String dataType, Object... parameters) {
        // Handle special cases like dynamic matching
        if ("matchingRecords".equals(dataType) || "nonMatchingRecords".equals(dataType)) {
            // Dynamic data processing based on parameters
            return processMatchingLogic(parameters);
        }

        return (T) dataStore.get(dataType);
    }
}
```

## Integration with Rules

Data services integrate seamlessly with rule evaluation:

```
// Set up data service manager
DemoDataServiceManager dataManager = new DemoDataServiceManager();
dataManager.initializeWithMockData();
```

```
// Create rules engine with data context
RulesEngineConfiguration config = new RulesEngineConfiguration();
RulesEngine engine = new RulesEngine(config);

// Get data for rule evaluation
List<Product> products = dataManager.requestData("products");
Customer customer = dataManager.requestData("customer");

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

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

## Custom Data Sources

You can create custom data sources for specific business needs:

```
public class CustomDataSource implements DataSource {
    private final String name;
    private final String dataType;

    public CustomDataSource(String name, String dataType) {
        this.name = name;
        this.dataType = dataType;
    }

    @Override
    public <T> T getData(String dataType, Object... parameters) {
        // Implement custom data retrieval logic
        // Could connect to databases, APIs, files, etc.
        return retrieveCustomData(dataType, parameters);
    }

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

// Usage
DataServiceManager manager = new DataServiceManager();
manager.loadDataSources(
    new CustomDataSource("CustomProductsSource", "customProducts"),
    new CustomDataSource("CustomCustomerSource", "customCustomer"),
    new CustomDataSource("CustomTradesSource", "customTrades")
);
```

## Best Practices

### Data Service Organization

- Use meaningful names for data sources and data types
- Group related data sources logically
- Implement proper error handling for data retrieval failures
- Cache frequently accessed data for performance

#### Testing and Development

- Use `DemoDataServiceManager` for demonstrations and testing
- Create environment-specific data service configurations
- Mock external dependencies during development
- Validate data integrity before rule evaluation

#### Production Considerations

- Replace mock data sources with production implementations
- Implement proper connection pooling and resource management
- Add monitoring and health checks for data sources
- Use appropriate caching strategies for performance

# Migration from External Services

## Step-by-Step Migration Process

#### Step 1: Identify Migration Candidates

Analyze your existing lookup services for:

- Small, static datasets (< 100 records)
- Infrequently changing reference data
- Simple key-value lookups

#### Step 2: Extract Data

Export data from your existing service:

```
// Before: External service
@Service
public class CurrencyLookupService implements LookupService {
    public Currency lookup(String code) {
        // Database or API call
    }
}
```

#### Step 3: Create YAML Dataset

Convert to YAML format:

```
enrichments:
  - id: "currency-enrichment"
    type: "lookup-enrichment"
    lookup-config:
      lookup-dataset:
        type: "inline"
        key-field: "code"
        data:
          - code: "USD"
            name: "US Dollar"
            region: "North America"
          - code: "EUR"
```

```
            name: "Euro"
            region: "Europe"
```

**Step 4: Update Configuration**

Replace service calls with enrichment:

```
// After: YAML dataset enrichment
RulesEngineConfiguration config = YamlConfigurationLoader.load("config.yaml");
RulesEngine engine = new RulesEngine(config);
RuleResult result = engine.evaluate(data); // Enrichment happens automatically
```

**Step 5: Test and Validate**

Ensure the migration works correctly:

```
@Test
public void testCurrencyEnrichment() {
    Map<String, Object> data = Map.of("currency", "USD");
    RuleResult result = engine.evaluate(data);

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

# Best Practices

Following these best practices will help you build maintainable, performant, and reliable rule-based systems with APEX. These recommendations come from real-world experience and will save you time and effort in the long run.

## Configuration Organization

Good organization makes your rules easier to understand, maintain, and debug.

**File Organization:**

- **Use external dataset files for reusable data**: If multiple configurations need the same lookup data, put it in a separate file that can be shared
- **Keep inline datasets small**: Limit inline datasets to less than 50 records to keep configuration files readable
- **Use meaningful IDs and names**: Choose descriptive identifiers like "customer-age-validation" instead of "rule1"
- **Include metadata for documentation**: Add owner, purpose, and creation date information to help future maintainers

**Naming Conventions:**

- Use consistent naming patterns across your organization
- Include the business domain in rule IDs (e.g., "finance-trade-validation", "customer-eligibility-check")
- Use descriptive messages that help users understand what went wrong

## Performance Optimization

APEX is designed for high performance, but following these practices will ensure optimal speed.

**Caching Strategy:**

- **Enable caching for frequently accessed datasets**: Turn on caching for lookup data that's used often
- **Use appropriate cache TTL values**: Set cache expiration times based on how often your data changes
  - Static data (countries, currencies): 24 hours or more
  - Semi-static data (product categories): 1-4 hours
  - Dynamic data: 5-30 minutes
- **Monitor performance metrics**: Use APEX's built-in monitoring to identify slow rules or enrichments
- **Preload datasets when possible**: Load reference data at startup rather than on first use

**Rule Optimization:**

- Order rules by execution cost (fastest first) when using rule groups
- Use specific conditions to avoid unnecessary rule evaluations
- Consider using rule chains for complex multi-step logic

### Maintenance and Governance

Proper maintenance practices prevent technical debt and ensure long-term success.

**Version Control:**

- **Version control all configuration files**: Treat YAML configurations like code - use Git or similar
- **Use environment-specific configurations**: Have separate configurations for development, testing, and production
- **Document dataset sources and update procedures**: Record where data comes from and how to update it
- **Regular review and cleanup of unused datasets**: Remove obsolete rules and datasets to keep configurations clean

**Change Management:**

- Test configuration changes in non-production environments first
- Use meaningful commit messages when updating configurations
- Consider the impact of rule changes on existing processes
- Maintain a changelog for significant rule modifications

# Advanced Rule Patterns

For complex business scenarios requiring rule dependencies and chaining, the Rules Engine supports sophisticated patterns where rules depend on results of previous rules. These patterns are essential for multi-stage workflows and decision trees.

### Available Patterns

1. **Conditional Chaining** - Execute Rule B only if Rule A triggers
2. **Sequential Dependency** - Each rule builds upon results from the previous rule
3. **Result-Based Routing** - Route to different rule sets based on previous results
4. **Accumulative Chaining** - Build up a score/result across multiple rules
5. **Complex Financial Workflow** - Real-world nested rule scenarios with multi-stage processing
6. **Fluent Rule Builder** - Compose rules with conditional execution paths using fluent API

### Quick Example: Conditional Chaining

```
// 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));

// Conditional execution of Rule B based on Rule A result
if (resultsA.get(0).isTriggered()) {
    Rule ruleB = new Rule(
        "EnhancedDueDiligenceCheck",
        "#accountAge >= 3",
        "Enhanced due diligence check passed"
    );
    // Execute enhanced validation only when needed
}
```

## Quick Example: Accumulative Chaining

```
rule-chains:
  - id: "credit-scoring"
    pattern: "accumulative-chaining"
    configuration:
      accumulator-variable: "totalScore"
      initial-value: 0
      accumulation-rules:
        - id: "credit-score-component"
          condition: "#creditScore >= 700 ? 25 : (#creditScore >= 650 ? 15 : 10)"
          message: "Credit score component"
          weight: 1.0
        - id: "income-component"
          condition: "#annualIncome >= 80000 ? 20 : 15"
          message: "Income component"
          weight: 1.0
      final-decision-rule:
        id: "loan-decision"
        condition: "#totalScore >= 60 ? 'APPROVED' : 'DENIED'"
        message: "Final loan decision"
```

## When to Use Advanced Patterns

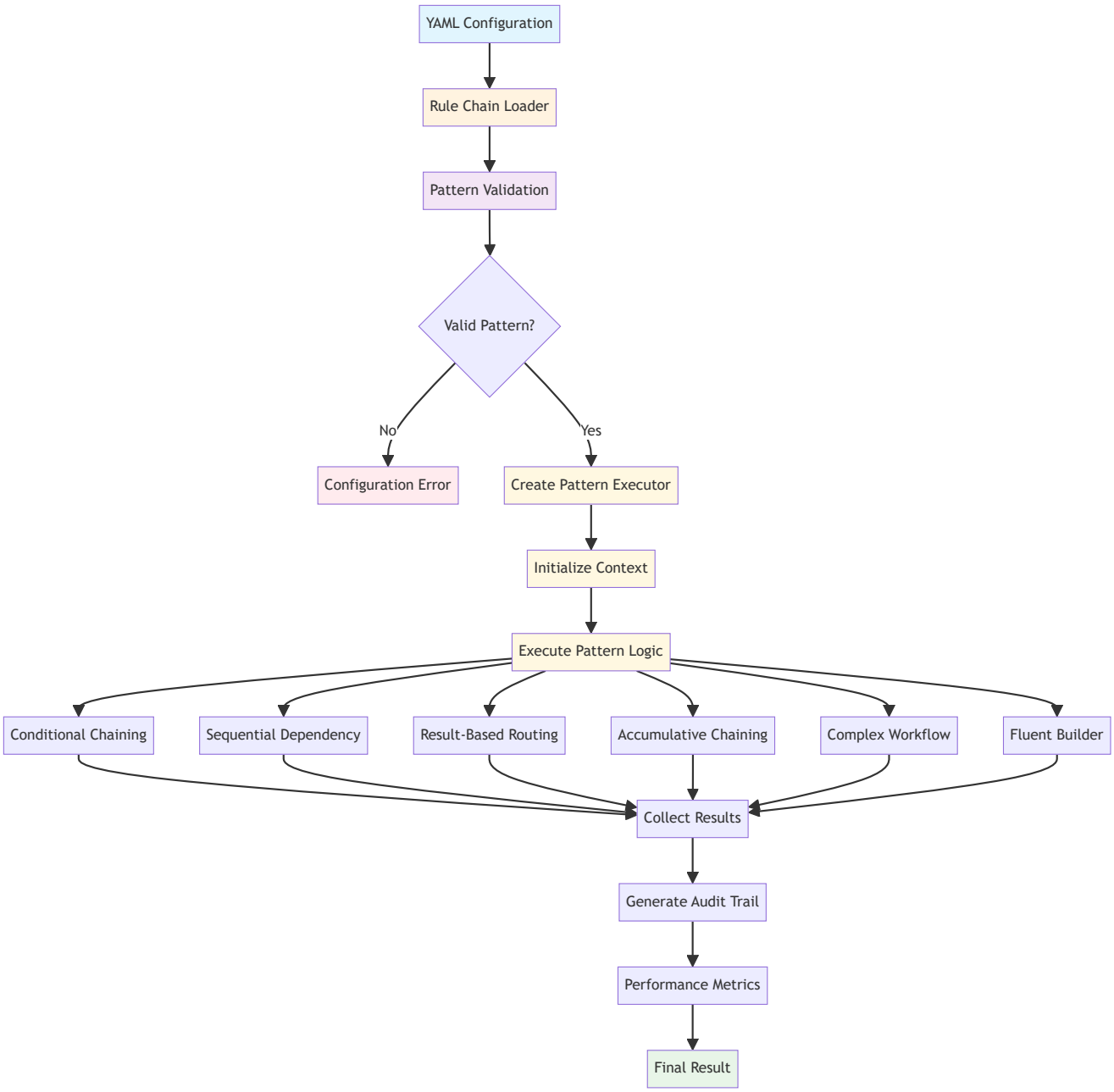
**Use advanced rule patterns when you need:**

- Multi-stage approval processes
- Risk-based processing with different validation paths
- Complex financial workflows with dependencies
- Decision trees with conditional branching
- Accumulative scoring systems
- Performance optimization through conditional execution

## Documentation Reference

## Rule Chains Configuration

Rule chains are configured in the `rule-chains` section of your YAML configuration file. Each rule chain specifies a pattern and its configuration:




```yaml
rule-chains:
  - id: "my-rule-chain"
    name: "My Rule Chain"
    description: "Description of what this chain does"
    pattern: "conditional-chaining"  # One of the 6 supported patterns
    enabled: true
    priority: 10
    category: "business-logic"
    configuration:
      # Pattern-specific configuration goes here
```

## Pattern-Specific Configuration Examples

### Pattern 1: Conditional Chaining

Execute expensive or specialized rules only when certain conditions are met:

```yaml
rule-chains:
  - id: "high-value-processing"
    pattern: "conditional-chaining"
    configuration:
      trigger-rule:
        condition: "#customerType == 'PREMIUM' && #transactionAmount > 100000"
        message: "High-value customer transaction detected"
      conditional-rules:
        on-trigger:
          - condition: "#accountAge >= 3"
            message: "Enhanced due diligence check"
        on-no-trigger:
          - condition: "true"
            message: "Standard processing applied"
```

### Pattern 2: Sequential Dependency

Build processing pipelines where each stage uses results from previous stages:

```yaml
rule-chains:
  - id: "discount-pipeline"
    pattern: "sequential-dependency"
    configuration:
      stages:
        - stage: 1
          name: "Base Discount"
          rule:
            condition: "#customerTier == 'GOLD' ? 0.15 : 0.05"
            message: "Base discount calculated"
          output-variable: "baseDiscount"
        - stage: 2
          name: "Regional Multiplier"
          rule:
            condition: "#region == 'US' ? #baseDiscount * 1.2 : #baseDiscount"
            message: "Regional multiplier applied"
          output-variable: "finalDiscount"
```

### Pattern 3: Result-Based Routing

Route to different rule sets based on intermediate results:

```yaml
rule-chains:
  - id: "risk-routing"
    pattern: "result-based-routing"
    configuration:
      router-rule:
        condition: "#riskScore > 70 ? 'HIGH_RISK' : 'LOW_RISK'"
        message: "Risk level determined"
      routes:
        HIGH_RISK:
          rules:
            - condition: "#transactionAmount > 100000"
```

```
            message: "Manager approval required"
        LOW_RISK:
          rules:
            - condition: "#transactionAmount > 0"
              message: "Basic validation"
```

**Pattern 4: Accumulative Chaining with Weight-Based Rule Selection**

Build up scores across multiple criteria with weighted components and intelligent rule selection:

```
rule-chains:
  - id: "advanced-credit-scoring"
    pattern: "accumulative-chaining"
    configuration:
      accumulator-variable: "totalScore"
      initial-value: 0

      # NEW: Rule Selection Strategy
      rule-selection:
        strategy: "weight-threshold"  # Execute only high-importance rules
        weight-threshold: 0.7

      accumulation-rules:
        - id: "credit-score-component"
          condition: "#creditScore >= 700 ? 25 : (#creditScore >= 650 ? 15 : 10)"
          message: "Credit score component"
          weight: 0.9      # High importance - will be executed
          priority: "HIGH"
        - id: "income-component"
          condition: "#annualIncome >= 80000 ? 20 : (#annualIncome >= 60000 ? 15 : 10)"
          message: "Income component"
          weight: 0.8      # High importance - will be executed
          priority: "HIGH"
        - id: "employment-component"
          condition: "#employmentYears >= 5 ? 15 : (#employmentYears >= 2 ? 10 : 5)"
          message: "Employment component"
          weight: 0.6      # Below threshold - will be skipped
          priority: "MEDIUM"
        - id: "debt-ratio-component"
          condition: "(#existingDebt / #annualIncome) < 0.2 ? 10 : 0"
          message: "Debt-to-income ratio component"
          weight: 0.5      # Below threshold - will be skipped
          priority: "LOW"
      final-decision-rule:
        condition: "#totalScore >= 40 ? 'APPROVED' : (#totalScore >= 25 ? 'CONDITIONAL' : 'DENIED')"
        message: "Final loan decision"
```

**Rule Selection Strategies:**

1. **Weight Threshold**: Execute only rules above a weight threshold

```
rule-selection:
  strategy: "weight-threshold"
  weight-threshold: 0.7  # Only rules with weight >= 0.7
```

2. **Top Weighted**: Execute the N highest-weighted rules

```yaml
rule-selection:
  strategy: "top-weighted"
  max-rules: 3  # Execute top 3 rules by weight
```

3. **Priority Based**: Execute rules based on priority levels

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

4. **Dynamic Threshold**: Calculate threshold based on context

```yaml
rule-selection:
  strategy: "dynamic-threshold"
  threshold-expression: "#riskLevel == 'HIGH' ? 0.8 : 0.6"
```

**Advanced Accumulative Features:**

1. **Weighted Components**: Different importance levels for components

```yaml
- id: "critical-component"
  condition: "#value > 100 ? 20 : 10"
  weight: 2.0  # This component has double impact
  priority: "HIGH"
```

2. **Negative Scoring**: Penalties that reduce the total score

```yaml
- id: "risk-penalty"
  condition: "#riskFactors > 3 ? -15 : 0"  # Subtract points for high risk
  weight: 1.0
  priority: "MEDIUM"
```

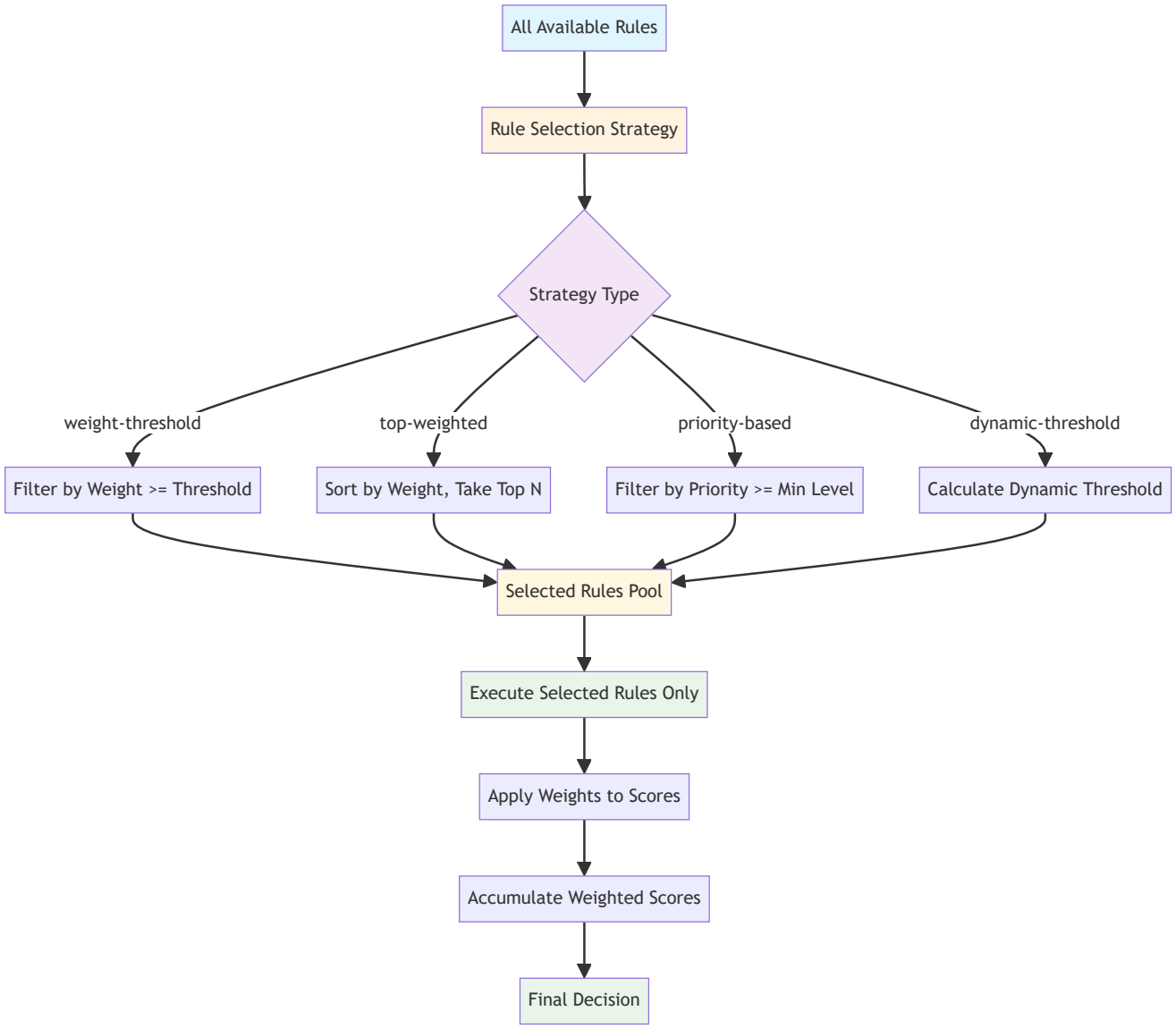
3. **Flexible Initial Values**: Start with a base score

```yaml
configuration:
  initial-value: 50  # Start with 50 points instead of 0
```

4. **Complex Conditional Logic**: Multi-tier scoring within rules

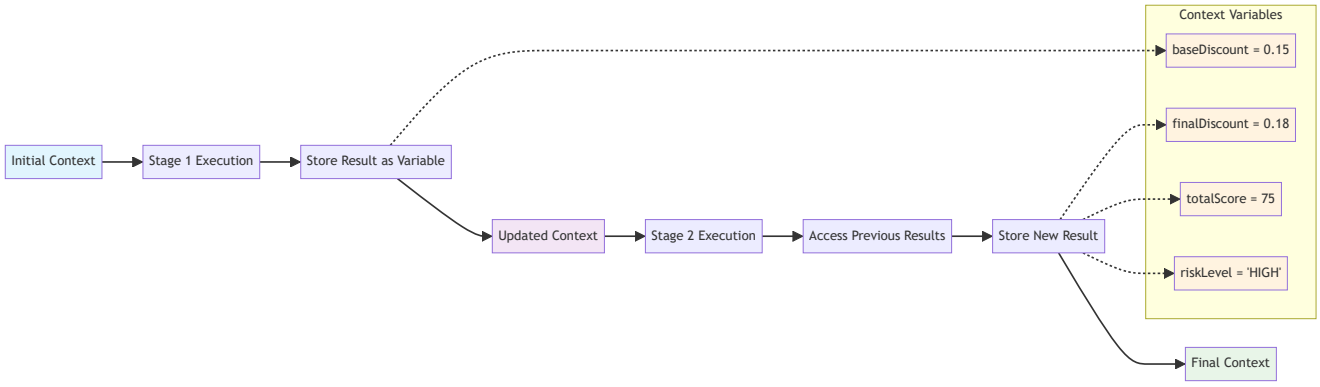
```yaml
- id: "tiered-scoring"
  condition: "#value >= 100 ? 30 : (#value >= 75 ? 20 : (#value >= 50 ? 10 : 0))"
  weight: 0.9
  priority: "HIGH"
```

**Weight-Based Rule Selection Flow:**




## Execution Context and Variable Propagation

Rule chains maintain an execution context that allows results from one stage to be used in subsequent stages:




- **Stage Results**: Each stage can store results using `output-variable`

- **Accumulator Variables**: Accumulative chains maintain running totals
- **Context Variables**: All results are available as SpEL variables in later rules

Example of variable usage:

```yaml
# Stage 1 stores result as 'baseDiscount'
- stage: 1
  rule:
    condition: "#customerTier == 'GOLD' ? 0.15 : 0.05"
  output-variable: "baseDiscount"

# Stage 2 uses 'baseDiscount' from Stage 1
- stage: 2
  rule:
    condition: "#baseDiscount * 1.2"  # Uses result from Stage 1
  output-variable: "finalDiscount"
```

## Execution Results and Monitoring

Rule chains provide comprehensive execution results and monitoring capabilities:

### Accessing Execution Results

```java
// Execute a rule chain
RuleChainResult result = ruleChainExecutor.executeRuleChain(ruleChain, context);

// Check overall success
if (result.isSuccessful()) {
    System.out.println("Chain executed successfully: " + result.getFinalOutcome());
} else {
    System.out.println("Chain failed: " + result.getErrorMessage());
}

// Access execution path
List<String> executionPath = result.getExecutionPath();
System.out.println("Rules executed: " + String.join(" → ", executionPath));

// Access stage results (for sequential dependency and accumulative chaining)
Object stageResult = result.getStageResult("baseDiscount");
System.out.println("Base discount calculated: " + stageResult);

// Access performance metrics
long executionTime = result.getExecutionTimeMillis();
int rulesExecuted = result.getExecutedRulesCount();
int rulesTriggered = result.getTriggeredRulesCount();
```

### Pattern-Specific Result Access

#### Sequential Dependency Results:

```java
// Access results from each stage
Double baseDiscount = (Double) result.getStageResult("baseDiscount");
Double finalDiscount = (Double) result.getStageResult("finalDiscount");
BigDecimal finalAmount = (BigDecimal) result.getStageResult("finalAmount");
```

#### Accumulative Chaining Results:

```
// Access component scores
Double creditComponent = (Double) result.getStageResult("component_1_credit-score-component_score");
Double incomeComponent = (Double) result.getStageResult("component_2_income-component_score");

// Access weighted scores
Double creditWeighted = (Double) result.getStageResult("component_1_credit-score-component_weighted");

// Access accumulator progression
Double initialScore = (Double) result.getStageResult("totalScore_initial");
Double finalScore = (Double) result.getStageResult("totalScore_final");
```

#### Result-Based Routing Results:

```
// Access routing decision
String routeKey = (String) result.getStageResult("routeKey");
String routeExecutionResult = (String) result.getStageResult("routeExecutionResult");
Integer routeExecutedRules = (Integer) result.getStageResult("routeExecutedRules");
```

## Error Handling and Validation

Rule chains include comprehensive validation and error handling:

- **Configuration Validation**: Pattern-specific validation ensures required fields are present
- **Runtime Error Handling**: Individual rule failures don't stop the entire chain
- **Execution Tracking**: Full audit trail of which rules executed and their results
- **Performance Monitoring**: Execution time tracking for optimization
- **Detailed Error Messages**: Specific error information for troubleshooting
- **Graceful Degradation**: Chains continue executing even when individual rules fail

## Practical Examples

#### Credit Application Processing

Combine multiple patterns for comprehensive credit application processing:

```
rule-chains:
  # First, determine application complexity
  - id: "application-complexity-routing"
    pattern: "result-based-routing"
    configuration:
      router-rule:
        condition: "#loanAmount > 500000 || #applicantType == 'BUSINESS' ? 'COMPLEX' : 'SIMPLE'"
      routes:
        COMPLEX:
          rules:
            - condition: "#documentationComplete == true"
              message: "Complex application documentation check"
        SIMPLE:
          rules:
            - condition: "#basicInfoComplete == true"
              message: "Simple application basic info check"

  # Then, calculate credit score
  - id: "credit-score-calculation"
    pattern: "accumulative-chaining"
    configuration:
```

```
      accumulator-variable: "creditScore"
      initial-value: 0
      accumulation-rules:
        - condition: "#creditHistory >= 700 ? 30 : 15"
          weight: 1.0
        - condition: "#incomeStability >= 0.8 ? 25 : 10"
          weight: 1.0
        - condition: "#debtToIncomeRatio < 0.3 ? 20 : 0"
          weight: 1.0
      final-decision-rule:
        condition: "#creditScore >= 60 ? 'APPROVED' : 'DENIED'"

  # Finally, apply conditional enhanced processing for high-value loans
  - id: "enhanced-processing"
    pattern: "conditional-chaining"
    configuration:
      trigger-rule:
        condition: "#loanAmount > 1000000 && #creditScore >= 60"
      conditional-rules:
        on-trigger:
          - condition: "#manualReviewRequired = true"
            message: "High-value loan requires manual review"
```

#### Performance-Based Pricing

Use sequential dependency for complex pricing calculations:

```
rule-chains:
  - id: "performance-pricing"
    pattern: "sequential-dependency"
    configuration:
      stages:
        - stage: 1
          name: "Base Rate Calculation"
          rule:
            condition: "#customerTier == 'PLATINUM' ? 0.02 : (#customerTier == 'GOLD' ? 0.025 : 0.03)"
          output-variable: "baseRate"
        - stage: 2
          name: "Volume Discount"
          rule:
            condition: "#tradingVolume > 10000000 ? #baseRate * 0.8 : (#tradingVolume > 5000000 ? #baseRate * 0.9 : #base
          output-variable: "discountedRate"
        - stage: 3
          name: "Relationship Bonus"
          rule:
            condition: "#relationshipYears > 5 ? #discountedRate * 0.95 : #discountedRate"
          output-variable: "finalRate"
```

#### Comprehensive Financial Services Example

Combining all 4 implemented patterns for complete loan processing:

```
rule-chains:
  # Step 1: Route based on loan type and amount
  - id: "loan-application-routing"
    pattern: "result-based-routing"
    priority: 10
    configuration:
      router-rule:
        condition: "#loanType == 'MORTGAGE' && #loanAmount > 500000 ? 'COMPLEX_MORTGAGE' : (#loanType == 'PERSONAL' ? 'PE
```

```yaml
      output-variable: "loanCategory"
    routes:
      COMPLEX_MORTGAGE:
        rules:
          - condition: "#propertyAppraisal != null && #incomeVerification == true"
            message: "Complex mortgage documentation verified"
      PERSONAL_LOAN:
        rules:
          - condition: "#loanAmount <= 50000"
            message: "Personal loan within limits"
      STANDARD_LOAN:
        rules:
          - condition: "#basicDocumentation == true"
            message: "Standard loan documentation complete"

# Step 2: Calculate comprehensive credit score
- id: "comprehensive-credit-scoring"
  pattern: "accumulative-chaining"
  priority: 20
  configuration:
    accumulator-variable: "creditScore"
    initial-value: 0
    accumulation-rules:
      - id: "credit-history"
        condition: "#creditRating >= 750 ? 35 : (#creditRating >= 700 ? 30 : (#creditRating >= 650 ? 20 : 10))"
        weight: 1.5  # Credit history is most important
      - id: "income-stability"
        condition: "#annualIncome >= 100000 ? 25 : (#annualIncome >= 75000 ? 20 : 15)"
        weight: 1.2
      - id: "employment-history"
        condition: "#employmentYears >= 5 ? 20 : (#employmentYears >= 2 ? 15 : 10)"
        weight: 1.0
      - id: "debt-burden"
        condition: "(#totalDebt / #annualIncome) < 0.2 ? 15 : ((#totalDebt / #annualIncome) < 0.4 ? 5 : -10)"
        weight: 1.0
      - id: "assets"
        condition: "#liquidAssets >= 50000 ? 10 : (#liquidAssets >= 25000 ? 5 : 0)"
        weight: 0.8
    final-decision-rule:
      condition: "#creditScore >= 80 ? 'EXCELLENT' : (#creditScore >= 60 ? 'GOOD' : (#creditScore >= 40 ? 'FAIR' : 'POO

# Step 3: Apply conditional enhanced processing for high-value loans
- id: "enhanced-processing"
  pattern: "conditional-chaining"
  priority: 30
  configuration:
    trigger-rule:
      condition: "#loanAmount > 1000000 && #creditScore >= 60"
      message: "High-value loan with acceptable credit"
    conditional-rules:
      on-trigger:
        - condition: "#manualUnderwritingRequired = true"
          message: "Manual underwriting required for high-value loan"
        - condition: "#seniorApprovalRequired = true"
          message: "Senior management approval required"
      on-no-trigger:
        - condition: "#standardProcessing = true"
          message: "Standard automated processing"

# Step 4: Calculate final terms based on all previous results
- id: "loan-terms-calculation"
  pattern: "sequential-dependency"
  priority: 40
  configuration:
    stages:
      - stage: 1
```

```
      name: "Base Interest Rate"
      rule:
        condition: "#creditScore >= 80 ? 0.035 : (#creditScore >= 60 ? 0.045 : (#creditScore >= 40 ? 0.055 : 0.065))"
      output-variable: "baseRate"
    - stage: 2
      name: "Loan Amount Adjustment"
      rule:
        condition: "#loanAmount > 1000000 ? #baseRate + 0.005 : #baseRate"
      output-variable: "adjustedRate"
    - stage: 3
      name: "Final Rate with Fees"
      rule:
        condition: "#adjustedRate + (#loanAmount * 0.001 / #loanAmount)"
      output-variable: "finalRate"
```

**Complex Financial Workflow Example**

Pattern 5 for multi-stage processing with dependencies:

```
rule-chains:
  - id: "trade-settlement-workflow"
    pattern: "complex-workflow"
    configuration:
      stages:
        - stage: "trade-validation"
          name: "Trade Data Validation"
          rules:
            - condition: "#tradeType != null && #notionalAmount != null && #counterparty != null"
              message: "Basic trade data validation"
          failure-action: "terminate"
        - stage: "risk-assessment"
          name: "Risk Assessment"
          depends-on: ["trade-validation"]
          rules:
            - condition: "#notionalAmount > 1000000 && #marketVolatility > 0.2 ? 'HIGH' : 'MEDIUM'"
              message: "Risk level assessment"
          output-variable: "riskLevel"
        - stage: "approval-workflow"
          name: "Approval Workflow"
          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"
        - stage: "settlement-calculation"
          name: "Settlement Processing"
          depends-on: ["approval-workflow"]
          rules:
            - condition: "#tradeType == 'DERIVATIVE' ? 5 : (#tradeType == 'EQUITY' ? 3 : 2)"
              message: "Settlement days calculated"
          output-variable: "settlementDays"
```

**Fluent Decision Tree Example**

Pattern 6 for complex decision trees:

```yaml
rule-chains:
  - id: "customer-onboarding-tree"
    pattern: "fluent-builder"
    configuration:
      root-rule:
        id: "customer-tier-check"
        condition: "#customerType == 'VIP' || #customerType == 'PREMIUM'"
        message: "High-tier customer detected"
        on-success:
          rule:
            id: "account-value-check"
            condition: "#initialDeposit > 100000"
            message: "High-value account detected"
            on-success:
              rule:
                id: "expedited-onboarding"
                condition: "true"
                message: "Expedited onboarding approved"
            on-failure:
              rule:
                id: "standard-premium-onboarding"
                condition: "true"
                message: "Standard premium onboarding"
        on-failure:
          rule:
            id: "standard-customer-check"
            condition: "#initialDeposit > 1000"
            message: "Standard customer validation"
            on-success:
              rule:
                id: "standard-onboarding"
                condition: "true"
                message: "Standard onboarding approved"
            on-failure:
              rule:
                id: "minimum-deposit-required"
                condition: "true"
                message: "Minimum deposit requirement not met"
```

This comprehensive example demonstrates how all 6 implemented patterns work together to create sophisticated business processing systems with routing, scoring, conditional processing, sequential calculations, complex workflows, and decision trees.

## Integration with Existing Rules

Rule chains work alongside traditional rules and rule groups:

```yaml
# Traditional rules continue to work
rules:
  - id: "basic-rule"
    condition: "#amount > 0"
    message: "Basic validation"

# Rule groups continue to work
rule-groups:
  - id: "validation-group"
    rule-ids: ["basic-rule"]

# Rule chains add advanced patterns
rule-chains:
  - id: "advanced-processing"
    pattern: "sequential-dependency"
```

```
configuration:
  # Advanced configuration here
```

For complete implementation details, examples, and architecture information, see the **Technical Reference Guide** section on "Nested Rules and Rule Chaining Patterns".

# Getting Help

## Common Issues

- **Configuration not loading**: Check YAML syntax and file paths
- **Enrichment not working**: Verify condition expressions and field mappings
- **Performance issues**: Enable caching and monitor metrics
- **Data not found**: Check key field matching and default values

## Documentation Resources

- Technical Implementation Guide for architecture details
- Financial Services Guide for domain-specific examples
- Configuration examples and templates

## Support

- Create GitHub issues for bugs or feature requests
- Check existing documentation for common solutions
- Review configuration examples for similar use cases