APEX SpEL (Spring Expression Language) Guide

Version: 2.3 **Last Updated:** 2025-10-13 **Status:** Production Ready

Table of Contents

Overview

Spring Expression Language (SpEL) is a powerful expression language that supports querying and manipulating objects at runtime. APEX uses SpEL consistently across all features to provide a unified, expressive syntax for data access and manipulation.

Why SpEL in APEX?

- **Consistency:** Same syntax across conditions, transformations, calculations, lookup keys, and field mappings
- **Power:** Full access to nested fields, arrays, collections, and complex expressions
- **Safety:** Safe navigation operators prevent null pointer exceptions
- **Flexibility:** Dynamic property access and runtime evaluation
- **Performance:** Expression caching for optimal performance

SpEL in APEX Features

SpEL is now supported consistently across ALL APEX features:

Feature	SpEL Support	Example
Conditions	✅ Yes	`condition: '#data.currency != null'`
Transformations	✅ Yes	`transformation: '#data.currency'`

Feature	SpEL Support	Example
Lookup Keys	✅ Yes	`lookup-key: '#symbol'`
Calculations	✅ Yes	`expression: '#amount * 0.01'`
Field Mappings	✅ **NEW (v2.3)**	`source-field: '#data.currency'`

The `#` Prefix Convention

- **With `#` prefix:** Expression is evaluated as SpEL
- **Without `#` prefix:** Treated as simple field name (backward compatible)

```
# SpEL expression (evaluated)
source-field: "#data.currency"

# Simple field name (direct lookup)
source-field: "currency"
```

Field Mapping SpEL Support

New in Version 2.3: Field mappings now support SpEL expressions in both `source-field` and `target-field`.

Problem Solved

Before v2.3:

```
enrichments:
  - id: "field-enrichment-demo"
    condition: "#data.currency != null"  # ✅ Works
    field-mappings:
      - source-field: "currency"         # ❌ Fails - can't access nested
        target-field: "buy_currency"
```

After v2.3 (SOLVED!):

```
enrichments:
  - id: "field-enrichment-demo"
    condition: "#data.currency != null"  # ✅ Works
    field-mappings:
      - source-field: "#data.currency"   # ✅ NOW WORKS!
        target-field: "buy_currency"
```

Features Enabled in Field Mappings

1. Nested Field Access

```yaml
field-mappings:
  - source-field: "#data.currency"
    target-field: "buy_currency"
  - source-field: "#data.trade.counterparty"
    target-field: "counterparty_name"
```

2. Safe Navigation

```yaml
field-mappings:
  - source-field: "#data?.currency"
    target-field: "currency_code"
  - source-field: "#data?.trade?.amount"
    target-field: "trade_amount"
```

3. Array Indexing

```yaml
field-mappings:
  - source-field: "#items[0].price"
    target-field: "first_item_price"
```

4. Complex Expressions

```yaml
field-mappings:
  - source-field: "#status == 'ACTIVE' ? #activePrice : #inactivePrice"
    target-field: "current_price"
```

5. Method Calls

```yaml
field-mappings:
  - source-field: "#currency.toUpperCase()"
    target-field: "currency_code"
```

6. Combination with Transformations

```yaml
field-mappings:
  - source-field: "#data.amount"
    target-field: "adjusted_amount"
    transformation: "#value * 1.1"
```

Backward Compatibility

100% Backward Compatible - Existing configurations work unchanged:

```yaml
field-mappings:
  # Old style - still works
  - source-field: "currency"
    target-field: "currency_code"

  # New style - also works
```

```
  - source-field: "#data.currency"
    target-field: "buy_currency"
```

Basic Syntax

Property Access Notation

SpEL offers three ways to access properties and arrays:

Option 1: Pure Bracket Notation

```
condition: "#trade['otcTrade']['otcLeg'][0]['stbRuleName'] != null"
```

When to use:

- Dynamic property names from variables
- Property names with special characters
- Runtime-determined property names

Option 2: Mixed Notation (Recommended)

```
condition: "#trade.otcTrade.otcLeg[0]['stbRuleName'] != null"
```

When to use:

- Best balance of readability and flexibility
- Known structure with some dynamic parts
- Most common in real-world APEX rules

Option 3: Pure Dot Notation

```
condition: "#trade.otcTrade.otcLeg[0].stbRuleName != null"
```

When to use:

- Fixed structure with known property names
- Maximum readability
- No special characters in property names

Comparison Summary

Aspect	Bracket `['prop']`	Mixed `.prop['dynamic']`	Dot `.prop`
Readability	⚠️ Verbose	✅ **Best balance**	✅ Clean
Dynamic Properties	✅ Full support	✅ Partial support	❌ No support

Aspect	Bracket `['prop']`	Mixed `.prop['dynamic']`	Dot `.prop`
Special Characters	✅ Handles all	✅ In brackets only	❌ Limited
Performance	⚠️ Slightly slower	✅ **Optimal**	✅ Fastest
Maintenance	⚠️ More typing	✅ **Recommended**	✅ Simple

Safe Navigation

Always use safe navigation (`?.`) to prevent null pointer exceptions:

```
# Safe array access - prevents errors if any level is null
condition: "#trade?.otcTrade?.otcLeg?.[0]?.stbRuleName != null"

# Safe access with bracket notation
condition: "#trade?.['otcTrade']?.['otcLeg']?.[0]?.['stbRuleName'] != null"

# Mixed safe navigation
condition: "#portfolio?.positions?.[0]?.trades?.size() > 0"
```

Why Safe Navigation is Critical

```
# ❌ UNSAFE - can throw NullPointerException
condition: "#trade.otcTrade.otcLeg[0].stbRuleName != null"

# ✅ SAFE - handles nulls gracefully
condition: "#trade?.otcTrade?.otcLeg?.[0]?.stbRuleName != null"
```

Array and Collection Access

Basic Array Element Access

```
# Access first element
condition: "#positions[0].instrumentId != null"

# Access specific index
condition: "#trades[2].tradeId != null"

# Access last element (if size is known)
condition: "#items[#items.size() - 1].status == 'COMPLETE'"
```

Array Bounds Checking

Always check array bounds before accessing elements:

```
# Check array exists and has elements
condition: "#trade?.otcTrade?.otcLeg?.size() > 0 && #trade.otcTrade.otcLeg[0]?.stbRuleName != null"

# Check specific index exists
condition: "#trade?.otcTrade?.otcLeg?.size() > 2 && #trade.otcTrade.otcLeg[2]?.stbRuleName != null"

# Check minimum array size
condition: "#positions?.size() >= 3 && #positions[2].quantity > 0"
```

Dynamic Index Access

Pattern 1: Search-Based Access (Most Common)

When you need to find an array element by condition:

```
# Find first leg where legType equals 'FLOATING'
condition: "#trade.otcTrade.otcLeg.^[legType == 'FLOATING']?.stbRuleName != null"

# Find first position with specific instrument type
condition: "#portfolio.positions.^[instrumentType == 'BOND']?.quantity > 0"

# Find pay leg in swap trade
expression: "#trade.legs.^[payReceive == 'PAY']?.notionalAmount"
```

Why this is most common:

- Business logic driven
- Position independent
- Robust to data structure variations
- Self-documenting

Pattern 2: Variable Index Access

For truly dynamic array access where the index itself is variable:

```
# Using a variable index
condition: "#trade.otcTrade.otcLeg[#legIndex].stbRuleName != null"

# Dynamic index from another field
condition: "#trade.otcTrade.otcLeg[#trade.selectedLegIndex].stbRuleName != null"

# Safe dynamic index access
condition: "#trade?.otcTrade?.otcLeg?.size() > #legIndex && #trade.otcTrade.otcLeg[#legIndex]?.stbRuleName != null"
```

Search Operators

Operator	Description	Example
`.^[condition]`	Find **first** match	`otcLeg.^[legType == 'FLOATING']`

Operator	Description	Example
`.$[condition]`	Find **last** match	`otcLeg.$[status == 'ACTIVE']`
`.?[condition]`	Find **all** matches	`otcLeg.?[currency == 'USD']`

Best Practices

1. Prioritize Readability Over Cleverness

```
# ✅ PREFERRED - Clear, step-by-step logic
condition: "#trade?.structure == 'SIMPLE'"
condition: "#trade?.otcTrade?.otcLeg?.size() > 0"
condition: "#trade.otcTrade.otcLeg[0]?.stbRuleName != null"

# ❌ AVOID - Clever but hard to debug
condition: "#trade?.structure == 'SIMPLE' && #trade?.otcTrade?.otcLeg?.[0]?.stbRuleName != null"
```

2. Always Use Safe Navigation

```
# ✅ Good - safe navigation prevents NPE
condition: "#trade?.otcTrade?.otcLeg?.size() > 0"

# ❌ Bad - can throw NullPointerException
condition: "#trade.otcTrade.otcLeg.size() > 0"
```

3. Check Array Bounds

```
# ✅ Good - bounds checking
condition: "#items?.size() > 2 && #items[2]?.status == 'ACTIVE'"

# ❌ Bad - no bounds checking
condition: "#items[2].status == 'ACTIVE'"
```

4. Break Complex Logic Into Steps

```
# ✅ PREFERRED - Multiple simple rules
# Rule 1: Check high value
condition: "#trade?.notionalAmount > 1000000"
# Rule 2: Check counterparty rating
condition: "#trade?.counterparty?.rating in {'AAA', 'AA+', 'AA'}"
# Rule 3: Extract trade ID
expression: "#trade.tradeId"

# ❌ AVOID - One complex expression
expression: "#trades?.?[notionalAmount > 1000000 && counterparty?.rating in {'AAA', 'AA+', 'AA'}]?.![tradeId]"
```

5. Use Collection Operations for Filtering

```
# ✅ Good - use collection operations
condition: "#trades?.?[status == 'PENDING'].size() > 0"

# ❌ Less efficient - would require manual iteration
```

6. Validate Data Types

```
# ✅ Good - type validation
condition: "#data.items instanceof T(java.util.List) && #data.items.size() > 0"

# ❌ Risky - assumes type without checking
condition: "#data.items.size() > 0"
```

7. Use Meaningful Variable Names

```
# ✅ Good - clear variable names
condition: "#currentLegIndex < #trade.otcTrade.otcLeg.size()"

# ❌ Less clear - generic names
condition: "#i < #trade.otcTrade.otcLeg.size()"
```

8. Avoid Repeated Expensive Operations

```
# ❌ Inefficient - repeated expensive operations
condition: "#expensiveCalculation()[0] != null && #expensiveCalculation()[0].value > 100"

# ✅ Efficient - calculate once, store in variable
condition: "#result = #expensiveCalculation(); #result?.size() > 0 && #result[0]?.value > 100"
```

Common Pitfalls

1. Syntax Errors

```
# ❌ Wrong - incorrect bracket syntax
condition: "#trade.['otcTrade'].['otcLeg'][0].['stbRuleName'] != null"

# ✅ Correct - proper bracket syntax
condition: "#trade['otcTrade']['otcLeg'][0]['stbRuleName'] != null"
```

2. Null Pointer Exceptions

```
# ❌ Wrong - can cause NullPointerException
condition: "#trade.otcTrade.otcLeg[0].stbRuleName != null"

# ✅ Correct - safe navigation
```

```
condition: "#trade?.otcTrade?.otcLeg?.[0]?.stbRuleName != null"
```

3. Array Bounds Errors

```
# ❌ Wrong - no bounds checking
condition: "#trade.otcTrade.otcLeg[5].stbRuleName != null"

# ✅ Correct - bounds checking
condition: "#trade?.otcTrade?.otcLeg?.size() > 5 && #trade.otcTrade.otcLeg[5]?.stbRuleName != null"
```

4. Type Assumptions

```
# ❌ Wrong - assumes array type
condition: "#data.items[0].name != null"

# ✅ Correct - validates type first
condition: "#data.items instanceof T(java.util.List) && #data.items.size() > 0 && #data.items[0]?.name != null"
```

5. Missing `#` Prefix in Field Mappings

```
# ❌ Wrong - trying to access nested field without SpEL
field-mappings:
  - source-field: "data.currency"  # Looks for field literally named "data.currency"
    target-field: "currency_code"

# ✅ Correct - use # prefix for SpEL
field-mappings:
  - source-field: "#data.currency"  # Evaluates as SpEL expression
    target-field: "currency_code"
```

Real-World Examples

Trade Processing

```
# Validate a multi-leg derivative trade
enrichments:
  - id: "multi-leg-validation"
    type: "field-enrichment"
    condition: "#trade?.legs?.size() > 1"
    field-mappings:
      # Extract first leg currency
      - source-field: "#trade.legs[0].currency"
        target-field: "leg1_currency"

      # Extract second leg currency
      - source-field: "#trade.legs[1].currency"
        target-field: "leg2_currency"

      # Calculate total notional
      - source-field: "#trade.legs.![notionalAmount].sum()"
```

```
        target-field: "total_notional"
```

Risk Management

```
# Check if portfolio exceeds risk limits
rules:
  - id: "risk-limit-check"
    name: "Portfolio Risk Limit Validation"
    condition: "#portfolio?.positions?.size() > 0"
    expression: "#portfolio.positions.![notionalAmount * riskWeight].sum()"
    severity: "ERROR"
    message: "Total risk-weighted exposure: ${#totalExposure}"

  - id: "risk-limit-breach"
    name: "Risk Limit Breach"
    condition: "#totalExposure > #riskLimits?.maxExposure"
    severity: "ERROR"
    message: "Portfolio exceeds maximum risk exposure"
```

Regulatory Reporting

```
# Extract required fields for regulatory report
enrichments:
  - id: "regulatory-extract"
    type: "field-enrichment"
    condition: "#trade?.reportingRequired == true"
    field-mappings:
      # Extract counterparty LEI
      - source-field: "#trade.counterparty?.lei"
        target-field: "counterparty_lei"

      # Extract all leg currencies
      - source-field: "#trade.legs?.![currency]"
        target-field: "leg_currencies"

      # Calculate total notional
      - source-field: "#trade.legs?.![notionalAmount].sum()"
        target-field: "total_notional"

      # Extract first leg maturity date
      - source-field: "#trade.legs.^[maturityDate != null]?.maturityDate"
        target-field: "maturity_date"
```

Lookup Enrichment with Nested Results

```
# Lookup instrument details and extract nested fields
enrichments:
  - id: "instrument-lookup"
    type: "lookup-enrichment"
    condition: "#symbol != null"
    lookup-config:
      lookup-key: "#symbol"
      lookup-dataset:
        type: "inline"
        key-field: "symbol"
        data:
          - symbol: "AAPL"
```

```
            data:
              instrument:
                name: "Apple Inc."
                type: "EQUITY"
              pricing:
                bid: 150.25
                ask: 150.30
      field-mappings:
        # Access nested fields in lookup result with SpEL
        - source-field: "#data.instrument.name"
          target-field: "instrument_name"
        - source-field: "#data.instrument.type"
          target-field: "instrument_type"
        - source-field: "#data.pricing.bid"
          target-field: "bid_price"
```

OTC Trade Leg Validation

```
rules:
  # Check if any leg has a specific rule
  - id: "otc-leg-rule-check"
    name: "OTC Leg Rule Validation"
    condition: "#trade?.otcTrade?.otcLeg?.?[stbRuleName == 'MARGIN_RULE'].size() > 0"
    message: "At least one leg must have margin rule"
    severity: "ERROR"

  # Validate all legs have required fields
  - id: "all-legs-complete"
    name: "All Legs Complete Validation"
    condition: "#trade?.otcTrade?.otcLeg?.?[stbRuleName == null || stbRuleName.trim().isEmpty()].size() == 0"
    message: "All legs must have stbRuleName specified"
    severity: "ERROR"

  # Access specific leg by position with safety
  - id: "first-leg-validation"
    name: "First Leg Validation"
    condition: "#trade?.otcTrade?.otcLeg?.size() > 0 && #trade.otcTrade.otcLeg[0]?.stbRuleName?.matches('[A-Z_]+')"
    message: "First leg must have valid rule name format"
    severity: "WARNING"
```

Advanced Patterns

Complex vs Simple: When to Use Each

✅ **Use Simple Patterns When:**

- New team members need to understand the logic quickly
- Debugging is required
- Business logic changes frequently
- Testing each logical step independently

✅ **Use Complex Patterns When:**

- Performance is critical
- Atomic operations are required

- Mathematical calculations must execute as one unit
- Experienced team with advanced SpEL knowledge

Example: Simple vs Complex

```
# ❌ COMPLEX: Everything in one expression
condition: "#trade?.legs?.size() > 1 && #trade.legs.?[notional > 0 && currency != null].size() == #trade.legs.size()"

# ✅ SIMPLE: Break into logical steps
condition: "#trade?.legs?.size() > 1"
condition: "#trade.legs.?[notional > 0].size() == #trade.legs.size()"
condition: "#trade.legs.?[currency != null].size() == #trade.legs.size()"
```

Summary

Key Takeaways

1. **Consistency:** SpEL is now used across ALL APEX features including field mappings
2. **Safety First:** Always use safe navigation (`?.`) and bounds checking
3. **Readability:** Prefer simple, step-by-step expressions over complex one-liners
4. **Backward Compatible:** Existing configurations continue to work unchanged
5. **Powerful:** Full SpEL capabilities for nested fields, arrays, and complex expressions

The `#` Prefix Rule

- **With `#`:** SpEL expression (evaluated at runtime)
- **Without `#`:** Simple field name (direct lookup)

Recommended Approach

1. **Start Simple:** Begin with readable, step-by-step expressions
2. **Use Safe Navigation:** Always use `?.` to prevent null pointer exceptions
3. **Check Bounds:** Validate array sizes before accessing elements
4. **Test Thoroughly:** Verify expressions work with various data scenarios
5. **Optimize Later:** Combine into complex expressions only if performance requires it

Implementation Notes

Version 2.3 Changes:

- Added SpEL support to `source-field` and `target-field` in field mappings
- Modified `getFieldValue()` and `setFieldValue()` methods in `YamlEnrichmentProcessor.java`
- 100% backward compatible with existing configurations
- Comprehensive test coverage (15 tests, all passing)
- No new dependencies required
- Graceful error handling (logs warnings, doesn't throw exceptions)

Files Modified:

- `apex-core/src/main/java/dev/mars/apex/core/service/enrichment/YamlEnrichmentProcessor.java`

Test Files Created:

- `apex-core/src/test/java/dev/mars/apex/core/service/enrichment/SpelFieldMappingTest.java`
- `apex-core/src/test/java/dev/mars/apex/core/service/enrichment/SpelFieldMappingIntegrationTest.java`

Remember: The most elegant code is often the simplest code that clearly expresses business intent.