APEX Rules Engine REST API - Complete Guide

Version: 1.0 **Date:** 2025-08-23 **Author:** Mark Andrew Ray-Smith Cityline Ltd

A comprehensive REST API for APEX (Advanced Processing Engine for eXpressions) with YAML Dataset Enrichment functionality.

Table of Contents

25. Support & Resources

1. Overview

The APEX Rules Engine REST API provides a complete interface for APEX, enabling rule evaluation, validation, configuration management, and system monitoring through HTTP endpoints. The API is built with Spring Boot and includes OpenAPI/Swagger documentation.

What You Can Do with the API

The APEX REST API provides comprehensive endpoints for:

- **Rule Evaluation**: Simple rule checking and complex validation scenarios
- **Expression Evaluation**: SpEL expression processing with context management
- **Data Transformation**: Transform and normalize data using configurable rules
- **Object Enrichment**: Enrich objects with data from external sources and datasets
- **Template Processing**: Generate dynamic content using templates with SpEL expressions
- **Data Source Management**: Manage and interact with external data sources
- **Configuration Management**: Load and manage YAML configurations via API
- **Performance Monitoring**: System health checks and performance metrics

Key Features

- **OpenAPI Documentation**: Interactive API documentation with Swagger UI
- **Spring Boot Actuator**: Production-ready monitoring and management endpoints
- **Comprehensive Error Handling**: Detailed error responses and validation
- **Named Rules Management**: Define and reuse named rules
- **Bootstrap Integration**: Complete integration with APEX bootstrap demonstrations
- **Batch Operations**: Process multiple items efficiently
- **Performance Metrics**: Built-in performance monitoring and optimization

Learning Path

This guide is structured to provide a gradual learning curve:

1. **Start Simple**: Begin with basic rule evaluation
2. **Build Understanding**: Learn request/response patterns
3. **Add Complexity**: Progress to transformations and enrichments
4. **Master Advanced Features**: Batch operations, authentication, optimization
5. **Real-World Usage**: Complete workflows and production deployment

Let's start with your first API call!

2. Quick Start

Prerequisites

- Java 17 or higher
- Spring Boot 3.x

- APEX Rules Engine Core modules

Build and Run

```
# Build the project
mvn clean package

# Run the application
java -jar target/apex-rest-api-1.0-SNAPSHOT.jar

# Or run with Maven
mvn spring-boot:run
```

Access the API

Once running, you can access:

- **API Base URL**: http://localhost:8080/api/
- **Swagger UI**: http://localhost:8080/swagger-ui.html
- **API Documentation**: http://localhost:8080/api-docs
- **Health Check**: http://localhost:8080/actuator/health

Verify Everything is Working

Test the health endpoint to make sure the API is running:

```
curl http://localhost:8080/actuator/health
```

You should see a response like:

```
{
  "status": "UP"
}
```

Great! Now you're ready for your first real API call.

3. Your First API Call

Let's start with the simplest possible API call - checking if a rule condition is true.

Simple Rule Check

This is the most basic operation - checking if a condition is met:

```
curl -X POST http://localhost:8080/api/rules/check \
   -H "Content-Type: application/json" \
   -d '{
     "condition": "#age >= 18",
     "data": {"age": 25},
```

```
    "ruleName": "age-check",
    "message": "User is an adult"
  }'
```

Understanding the Request

Let's break down what we just sent:

- **condition**: `"#age >= 18"` - A SpEL expression that checks if age is 18 or older
- **data**: `{"age": 25}` - The data context containing the age value
- **ruleName**: `"age-check"` - A friendly name for this rule
- **message**: `"User is an adult"` - A message to return when the rule matches

Understanding the Response

You should get a response like this:

```
{
  "success": true,
  "matched": true,
  "ruleName": "age-check",
  "message": "User is an adult",
  "timestamp": "2025-08-23T10:30:00Z",
  "evaluationId": "uuid-here"
}
```

What each field means:

- **success**: `true` - The API call was successful
- **matched**: `true` - The rule condition was met (age 25 is >= 18)
- **ruleName**: The name we gave the rule
- **message**: The message we specified
- **timestamp**: When the evaluation happened
- **evaluationId**: A unique ID for this evaluation (useful for debugging)

Try Different Values

Now try changing the age to see what happens:

```
curl -X POST http://localhost:8080/api/rules/check \
  -H "Content-Type: application/json" \
  -d '{
    "condition": "#age >= 18",
    "data": {"age": 16},
    "ruleName": "age-check",
    "message": "User is an adult"
  }'
```

This time you'll get `"matched": false` because 16 is not >= 18.

Congratulations! You've made your first successful API call. Let's learn more about how responses work.

4. Understanding Responses

All APEX API responses follow consistent patterns. Understanding these patterns will help you work with any endpoint.

Standard Success Response

Every successful API call returns a response with this basic structure:

```
{
  "success": true,
  "data": { /* response data */ },
  "timestamp": "2025-08-23T10:30:00Z"
}
```

Detailed Success Response (with metrics)

When you include `"includeMetrics": true` in your request, you get additional performance information:

```
{
  "success": true,
  "data": { /* response data */ },
  "timestamp": "2025-08-23T10:30:00Z",
  "metrics": {
    "executionTimeMs": 45,
    "memoryUsedBytes": 1024,
    "rulesEvaluated": 3
  }
}
```

Error Response

When something goes wrong, you'll get a response like this:

```
{
  "success": false,
  "error": "VALIDATION_ERROR",
  "message": "Detailed error description",
  "details": {
    "field": "condition",
    "rejectedValue": "invalid expression",
    "reason": "SpEL syntax error"
  },
  "correlationId": "abc123-def456",
  "timestamp": "2025-08-23T10:30:00Z"
}
```

Key points about error responses:

- **success**: Always `false` for errors
- **error**: A category code (like `VALIDATION_ERROR`, `EXPRESSION_ERROR`)
- **message**: Human-readable description of what went wrong
- **details**: Specific information about the error

- **correlationId**: Unique ID for tracking this error

Response Patterns to Remember

1. **Always check** `success` **first** - This tells you if the call worked
2. **Look for** `data` - This contains the actual results
3. **Use** `timestamp` - Helpful for debugging and logging
4. **Include metrics when optimizing** - Add `"includeMetrics": true` to requests

Now that you understand responses, let's learn about the different types of requests you can make.

5. Base URL and Content Types

Base URL

All API endpoints use this base URL:

```
http://localhost:8080/api
```

Content Type

All endpoints accept and return JSON:

```
Content-Type: application/json
Accept: application/json
```

Example Request Headers

```
curl -X POST http://localhost:8080/api/rules/check \
  -H "Content-Type: application/json" \
  -H "Accept: application/json" \
  -d '{"condition": "#value > 0", "data": {"value": 10}}'
```

6. Request/Response Models

Understanding the data structures used by the API will help you build requests correctly and interpret responses.

Core Request Models

`RuleEvaluationRequest`

Used for basic rule checking:

```
{
  "condition": "SpEL expression",     // @NotBlank - Required
  "data": { /* evaluation context */ }, // @NotNull - Required
```

```
  "ruleName": "optional-name",        // Optional friendly name
  "message": "optional message",      // Optional message for matches
  "includeMetrics": false            // Optional performance metrics
}
```

`RuleExecutionRequest`

Used for more advanced rule execution:

```
{
  "rule": {
    "name": "rule-name",        // @NotBlank - Required
    "condition": "SpEL expression", // @NotBlank - Required
    "message": "optional message",  // Optional
    "priority": "HIGH|MEDIUM|LOW"   // Optional priority level
  },
  "facts": { /* context data */ } // @NotNull - Required
}
```

`ValidationRequest`

Used for validating data against multiple rules:

```
{
  "data": { /* data to validate */ }, // @NotNull - Required
  "rules": [
    {
      "name": "validation-rule",      // Rule name
      "condition": "SpEL expression", // Rule condition
      "message": "error message",     // Error message
      "severity": "ERROR|WARNING|INFO" // Severity level
    }
  ]
}
```

`ExpressionEvaluationRequest`

Used for evaluating SpEL expressions:

```
{
  "expression": "SpEL expression",    // @NotBlank - Required
  "context": { /* variables */ },    // @NotNull - Required
  "includeMetrics": false,           // Optional metrics
  "validateSyntax": true             // Optional syntax validation
}
```

Batch Request Models

`BatchRuleExecutionRequest`

Execute multiple rules with shared context:

```
{
  "rules": [
```

```
      {
        "name": "rule-name",
        "condition": "SpEL expression",
        "message": "optional message"
      }
    ],
    "facts": { /* shared context data */ }
  }
```

`BatchExpressionRequest`

Evaluate multiple expressions:

```
  {
    "expressions": [
      {
        "name": "expression-name",
        "expression": "SpEL expression"
      }
    ],
    "context": { /* shared context variables */ }
  }
```

Transformation and Enrichment Models

`DynamicTransformationRequest`

Transform data with dynamic rules:

```
  {
    "data": { /* object to transform */ },
    "transformerRules": [
      {
        "name": "rule-name",           // @NotBlank
        "condition": "SpEL expression", // @NotBlank
        "transformation": "SpEL expression", // @NotBlank
        "targetField": "field-name"    // @NotBlank
      }
    ]
  }
```

`EnrichmentRequest`

Enrich objects with YAML configuration:

```
  {
    "targetObject": { /* object to enrich */ },
    "yamlConfiguration": "YAML config string"
  }
```

`BatchEnrichmentRequest`

Enrich multiple objects:

```
{
  "yamlConfiguration": "YAML config string", // @NotNull
  "targetObjects": [ /* array of objects */ ] // @NotNull
}
```

Template Processing Models

`TemplateProcessingRequest`

Process templates with SpEL expressions:

```
{
  "template": "template string with #{expressions}", // @NotBlank
  "context": { /* context variables */ }             // @NotNull
}
```

`BatchTemplateProcessingRequest`

Process multiple templates:

```
{
  "templates": [
    {
      "name": "template-name",
      "type": "JSON|XML|TEXT",
      "template": "template string"
    }
  ],
  "context": { /* shared context variables */ }
}
```

Response Models

Rule Execution Response

```
{
  "success": true,
  "rule": {
    "name": "rule-name",
    "condition": "SpEL expression",
    "message": "rule message",
    "priority": "HIGH"
  },
  "result": {
    "triggered": true,
    "ruleName": "rule-name",
    "message": "rule message",
    "resultType": "SUCCESS",
    "timestamp": "2025-08-23T10:30:00Z"
  },
  "timestamp": "2025-08-23T10:30:00Z"
}
```

Batch Response

```
{
  "success": true,
  "results": [
    { /* individual result 1 */ },
    { /* individual result 2 */ }
  ],
  "summary": {
    "total": 2,
    "successful": 2,
    "failed": 0
  },
  "timestamp": "2025-08-23T10:30:00Z"
}
```

Now let's look at HTTP status codes and error handling.

7. HTTP Status Codes

The API uses standard HTTP status codes to indicate the success or failure of requests.

Success Codes

- `200 OK` - Request successful, data returned
- `201 Created` - Resource created successfully (e.g., rule defined)

Client Error Codes

- `400 Bad Request` - Invalid request format or validation error
- `404 Not Found` - Resource not found (e.g., rule name, transformer)
- `409 Conflict` - Resource already exists (e.g., rule name conflict)
- `422 Unprocessable Entity` - Valid request format but business logic error

Server Error Codes

- `500 Internal Server Error` - Unexpected server error
- `503 Service Unavailable` - Service temporarily unavailable

Error Categories

The API groups errors into categories to help you understand what went wrong:

- `VALIDATION_ERROR` - Request validation failed
- `RULE_EVALUATION_ERROR` - Error evaluating rule condition
- `EXPRESSION_ERROR` - SpEL expression syntax or evaluation error
- `TRANSFORMATION_ERROR` - Data transformation failed
- `ENRICHMENT_ERROR` - Object enrichment failed
- `TEMPLATE_ERROR` - Template processing failed
- `DATA_SOURCE_ERROR` - Data source lookup failed
- `CONFIGURATION_ERROR` - Configuration parsing error

8. Error Handling

Understanding how to handle errors is crucial for building robust applications.

Standard Error Response Format

All errors follow this format:

```
{
  "success": false,
  "error": "ERROR_CATEGORY",
  "message": "Human-readable description",
  "timestamp": "2025-08-23T10:30:00Z",
  "correlationId": "unique-id-for-tracking"
}
```

Validation Error Response

When request validation fails, you get additional details:

```
{
  "success": false,
  "error": "VALIDATION_ERROR",
  "message": "Request validation failed",
  "validationErrors": [
    {
      "field": "rule.condition",
      "message": "Condition cannot be blank",
      "rejectedValue": null
    }
  ],
  "correlationId": "abc123-def456",
  "timestamp": "2025-08-23T10:30:00Z"
}
```

Common Error Scenarios

Expression Syntax Error

```
{
  "success": false,
  "error": "EXPRESSION_ERROR",
  "message": "SpEL syntax error: Unexpected token 'invalid' at position 5",
  "details": {
    "expression": "#invalid && syntax >",
    "position": 5,
    "token": "invalid"
  }
}
```

Rule Evaluation Error

```
{
  "success": false,
  "error": "RULE_EVALUATION_ERROR",
  "message": "Cannot resolve property 'nonExistentField' on object",
  "details": {
    "field": "nonExistentField",
    "availableFields": ["age", "name", "email"]
  }
}
```

Resource Not Found

```
{
  "success": false,
  "error": "RESOURCE_NOT_FOUND",
  "message": "No transformer found with name: invalid-transformer",
  "details": {
    "resourceType": "transformer",
    "resourceName": "invalid-transformer"
  }
}
```

Error Handling Best Practices

1. **Always check the `success` field first**
2. **Use the `error` category to determine error type**
3. **Show the `message` to users (it's human-readable)**
4. **Log the `correlationId` for debugging**
5. **Use `details` for programmatic error handling**

Example Error Handling in JavaScript

```
async function callAPI(endpoint, data) {
  try {
    const response = await fetch(endpoint, {
      method: 'POST',
      headers: { 'Content-Type': 'application/json' },
      body: JSON.stringify(data)
    });

    const result = await response.json();

    if (!result.success) {
      console.error(`API Error [${result.correlationId}]: ${result.message}`);

      // Handle specific error types
      switch (result.error) {
        case 'VALIDATION_ERROR':
          // Show validation errors to user
          break;
        case 'EXPRESSION_ERROR':
          // Help user fix expression syntax
          break;
        default:
          // Generic error handling
          break;
      }
```

```
        return null;
      }

      return result.data;
    } catch (error) {
      console.error('Network error:', error);
      return null;
    }
  }
```

Now that you understand the basics, let's explore the API endpoints, starting with the simplest ones.

9. Simple Operations - Rules API

The Rules API is the perfect place to start because it handles the most fundamental operations. We'll progress from simple to more complex operations.

Base Path: `/api/rules`

All rules operations use this base path. Here's what you can do:

Method	Endpoint	Description	Complexity
POST	`/check`	Check a single rule condition	⭐ Beginner
POST	`/validate`	Validate data against multiple rules	⭐⭐ Intermediate
POST	`/define/{name}`	Define a named rule for reuse	⭐⭐ Intermediate
POST	`/test/{name}`	Test a previously defined rule	⭐⭐ Intermediate
GET	`/defined`	Get all defined rules	⭐ Beginner
POST	`/execute`	Execute single rule with full details	⭐⭐⭐ Advanced
POST	`/batch`	Execute multiple rules	⭐⭐⭐ Advanced

⭐ Beginner: Simple Rule Check

You've already seen this in our first example. Let's explore it more:

```
curl -X POST http://localhost:8080/api/rules/check \
  -H "Content-Type: application/json" \
  -d '{
    "condition": "#balance > 1000",
    "data": {"balance": 1500, "currency": "USD"},
    "ruleName": "minimum-balance-check",
    "message": "Account has sufficient balance"
  }'
```

When to use: Perfect for simple yes/no decisions.

⭐ Beginner: Get Defined Rules

See what named rules are available:

```
curl -X GET http://localhost:8080/api/rules/defined
```

Response:

```
{
  "success": true,
  "rules": [
    {
      "name": "adult-check",
      "condition": "#age >= 18",
      "message": "Customer is an adult"
    }
  ],
  "count": 1,
  "timestamp": "2025-08-23T10:30:00Z"
}
```

⭐⭐ Intermediate: Define Named Rules

Create reusable rules that you can reference by name:

```
curl -X POST http://localhost:8080/api/rules/define/high-value-customer \
  -H "Content-Type: application/json" \
  -d '{
    "condition": "#accountBalance > 50000 && #customerTier == '\''GOLD'\''",
    "message": "High value gold customer identified"
  }'
```

Benefits of named rules:

- Reusable across different API calls
- Centralized rule management
- Easier to maintain and update

⭐⭐ Intermediate: Test Named Rules

Once you've defined a rule, test it with different data:

```
curl -X POST http://localhost:8080/api/rules/test/high-value-customer \
  -H "Content-Type: application/json" \
  -d '{
    "accountBalance": 75000,
    "customerTier": "GOLD",
    "customerId": "CUST001"
  }'
```

⭐⭐ Intermediate: Validate Data Against Multiple Rules

Check data against several rules at once:

```
curl -X POST http://localhost:8080/api/rules/validate \
  -H "Content-Type: application/json" \
  -d '{
    "data": {
      "age": 25,
      "email": "john@example.com",
      "balance": 1500,
      "country": "US"
    },
    "rules": [
      {
        "name": "age-validation",
        "condition": "#data.age >= 18",
        "message": "Age must be at least 18",
        "severity": "ERROR"
      },
      {
        "name": "email-validation",
        "condition": "#data.email != null && #data.email.contains('\''@'\'')",
        "message": "Valid email required",
        "severity": "ERROR"
      },
      {
        "name": "balance-warning",
        "condition": "#data.balance < 1000",
        "message": "Low account balance",
        "severity": "WARNING"
      }
    ]
  }'
```

Response shows all rule results:

```
{
  "success": true,
  "validationResults": [
    {
      "ruleName": "age-validation",
      "passed": true,
      "message": "Age must be at least 18",
      "severity": "ERROR"
    },
    {
      "ruleName": "email-validation",
      "passed": true,
      "message": "Valid email required",
      "severity": "ERROR"
    },
    {
      "ruleName": "balance-warning",
      "passed": true,
      "message": "Low account balance",
      "severity": "WARNING"
    }
  ],
  "overallResult": "PASSED",
  "timestamp": "2025-08-23T10:30:00Z"
}
```

⭐⭐⭐ Advanced: Execute Single Rule with Full Details

For more complex scenarios, use the execute endpoint:

```
curl -X POST http://localhost:8080/api/rules/execute \
  -H "Content-Type: application/json" \
  -d '{
    "rule": {
      "name": "high-value-trade",
      "condition": "#amount > 10000 && #currency == '\''USD'\''",
      "message": "High value USD trade detected",
      "priority": "HIGH"
    },
    "facts": {
      "amount": 15000.0,
      "currency": "USD",
      "customerId": "CUST001",
      "tradeType": "EQUITY"
    }
  }'
```

This gives you more detailed results:

```
{
  "success": true,
  "rule": {
    "name": "high-value-trade",
    "condition": "#amount > 10000 && #currency == 'USD'",
    "message": "High value USD trade detected",
    "priority": "HIGH"
  },
  "result": {
    "triggered": true,
    "ruleName": "high-value-trade",
    "message": "High value USD trade detected",
    "resultType": "SUCCESS",
    "timestamp": "2025-08-23T10:30:00Z"
  },
  "facts": {
    "amount": 15000.0,
    "currency": "USD",
    "customerId": "CUST001",
    "tradeType": "EQUITY"
  },
  "timestamp": "2025-08-23T10:30:00Z"
}
```

⭐⭐⭐ Advanced: Batch Rule Execution

Execute multiple rules against the same data:

```
curl -X POST http://localhost:8080/api/rules/batch \
  -H "Content-Type: application/json" \
  -d '{
    "rules": [
      {
        "name": "high-value-check",
        "condition": "#amount > 1000",
```

```
        "message": "High value transaction"
      },
      {
        "name": "gold-tier-check",
        "condition": "#tier == '\''GOLD'\''",
        "message": "Gold tier customer"
      },
      {
        "name": "risk-assessment",
        "condition": "#amount > 10000 && #country != '\''US'\''",
        "message": "High risk international transaction"
      }
    ],
    "facts": {
      "amount": 1500,
      "tier": "GOLD",
      "country": "US",
      "customerId": "CUST001"
    }
  }'
```

Batch responses show results for all rules:


```
{
  "success": true,
  "results": [
    {
      "triggered": true,
      "ruleName": "high-value-check",
      "message": "High value transaction"
    },
    {
      "triggered": true,
      "ruleName": "gold-tier-check",
      "message": "Gold tier customer"
    },
    {
      "triggered": false,
      "ruleName": "risk-assessment",
      "message": "High risk international transaction"
    }
  ],
  "summary": {
    "total": 3,
    "successful": 3,
    "triggered": 2,
    "failed": 0
  },
  "timestamp": "2025-08-23T10:30:00Z"
}
```


Rules API Summary

- **Start with** `/check` for simple rule evaluation
- **Use** `/validate` when you need to check multiple conditions
- **Define named rules** with `/define/{name}` for reusability
- **Use** `/execute` when you need detailed rule information
- **Use** `/batch` for processing multiple rules efficiently

Ready to move on to expression evaluation? That's our next topic!

10. Expression Evaluation API

The Expression Evaluation API lets you evaluate SpEL (Spring Expression Language) expressions with context data. This is the foundation that powers rules, transformations, and templates.

Base Path: `/api/expressions`

Method	Endpoint	Description	Complexity
POST	`/evaluate`	Evaluate single expression	⭐ Beginner
POST	`/evaluate/detailed`	Evaluate with detailed result	⭐⭐ Intermediate
POST	`/batch`	Evaluate multiple expressions	⭐⭐⭐ Advanced
POST	`/validate`	Validate expression syntax	⭐ Beginner
GET	`/functions`	Get available functions	⭐ Beginner

⭐ Beginner: Simple Expression Evaluation

Start with basic mathematical expressions:

```
curl -X POST http://localhost:8080/api/expressions/evaluate \
  -H "Content-Type: application/json" \
  -d '{
    "expression": "#amount * #rate + #fee",
    "context": {
      "amount": 1000,
      "rate": 0.05,
      "fee": 25
    }
  }'
```

Response:

```
{
  "success": true,
  "expression": "#amount * #rate + #fee",
  "context": {
    "amount": 1000,
    "rate": 0.05,
    "fee": 25
  },
  "result": 75.0,
  "resultType": "Double",
  "timestamp": "2025-08-23T10:30:00Z"
}
```

⭐ Beginner: Validate Expression Syntax

Before using complex expressions, validate them first:

```
curl -X POST http://localhost:8080/api/expressions/validate \
  -H "Content-Type: application/json" \
  -d '{
    "expression": "#customer.tier == '\''GOLD'\'' && #transaction.amount > 1000"
  }'
```

Response for valid expression:

```
{
  "success": true,
  "expression": "#customer.tier == 'GOLD' && #transaction.amount > 1000",
  "valid": true,
  "message": "Expression syntax is valid",
  "timestamp": "2025-08-23T10:30:00Z"
}
```

Response for invalid expression:

```
{
  "success": false,
  "expression": "#invalid && syntax >",
  "valid": false,
  "error": "Unexpected token at position 15",
  "suggestions": [
    "Check for missing operators",
    "Verify parentheses are balanced"
  ],
  "timestamp": "2025-08-23T10:30:00Z"
}
```

⭐ Beginner: Get Available Functions

See what functions you can use in expressions:

```
curl -X GET http://localhost:8080/api/expressions/functions
```

Response:

```
{
  "success": true,
  "functions": {
    "mathematical": [
      "abs(number) - Absolute value",
      "ceil(number) - Ceiling",
      "floor(number) - Floor",
      "round(number) - Round to nearest integer",
      "max(a, b) - Maximum of two values",
      "min(a, b) - Minimum of two values"
    ],
    "string": [
      "length() - String length",
      "substring(start, end) - Extract substring",
      "toLowerCase() - Convert to lowercase",
      "toUpperCase() - Convert to uppercase",
```

```
        "trim() - Remove whitespace",
        "contains(substring) - Check if contains",
        "startsWith(prefix) - Check if starts with",
        "endsWith(suffix) - Check if ends with"
      ],
      "logical": ["&&", "||", "!"],
      "comparison": ["==", "!=", "<", ">", "<=", ">="],
      "date": [
        "T(java.time.LocalDate).now() - Current date",
        "T(java.time.Instant).now() - Current timestamp"
      ]
    },
    "timestamp": "2025-08-23T10:30:00Z"
  }
```

⭐⭐ Intermediate: Complex Expressions

Try more complex expressions with nested objects:

```
curl -X POST http://localhost:8080/api/expressions/evaluate \
  -H "Content-Type: application/json" \
  -d '{
    "expression": "#customer.tier == '\''GOLD'\'' && #transaction.amount > #customer.limits.daily",
    "context": {
      "customer": {
        "id": "CUST001",
        "tier": "GOLD",
        "limits": {
          "daily": 5000,
          "monthly": 50000
        }
      },
      "transaction": {
        "amount": 7500,
        "currency": "USD"
      }
    }
  }'
```

⭐⭐ Intermediate: String Operations

Work with text data:

```
curl -X POST http://localhost:8080/api/expressions/evaluate \
  -H "Content-Type: application/json" \
  -d '{
    "expression": "#firstName.substring(0,1).toUpperCase() + #firstName.substring(1).toLowerCase() + '\'' '\'' + #lastNam
    "context": {
      "firstName": "john",
      "lastName": "doe"
    }
  }'
```

Result: `"John DOE"`

⭐⭐ Intermediate: Conditional Logic

Use ternary operators for conditional logic:

```
curl -X POST http://localhost:8080/api/expressions/evaluate \
  -H "Content-Type: application/json" \
  -d '{
    "expression": "#amount > 10000 ? '\''HIGH'\'' : (#amount > 1000 ? '\''MEDIUM'\'' : '\''LOW'\'')",
    "context": {
      "amount": 5000
    }
  }'
```

Result: `"MEDIUM"`

⭐⭐ Intermediate: Detailed Evaluation

Get more information about the evaluation process:

```
curl -X POST http://localhost:8080/api/expressions/evaluate/detailed \
  -H "Content-Type: application/json" \
  -d '{
    "expression": "#amount * #rate + #fee",
    "context": {
      "amount": 1000,
      "rate": 0.05,
      "fee": 25
    },
    "includeMetrics": true
  }'
```

Response includes performance metrics:

```
{
  "success": true,
  "expression": "#amount * #rate + #fee",
  "result": 75.0,
  "resultType": "Double",
  "evaluationSteps": [
    {"step": 1, "operation": "#amount", "result": 1000},
    {"step": 2, "operation": "#rate", "result": 0.05},
    {"step": 3, "operation": "1000 * 0.05", "result": 50.0},
    {"step": 4, "operation": "#fee", "result": 25},
    {"step": 5, "operation": "50.0 + 25", "result": 75.0}
  ],
  "metrics": {
    "executionTimeMs": 12,
    "memoryUsedBytes": 256
  },
  "timestamp": "2025-08-23T10:30:00Z"
}
```

⭐⭐⭐ Advanced: Batch Expression Evaluation

Evaluate multiple expressions with shared context:

```
curl -X POST http://localhost:8080/api/expressions/batch \
  -H "Content-Type: application/json" \
  -d '{
    "expressions": [
      {
        "name": "total-calculation",
        "expression": "#amount * #rate + #fee"
      },
      {
        "name": "age-check",
        "expression": "#age >= 18"
      },
      {
        "name": "risk-score",
        "expression": "#amount > 10000 ? 10 : (#amount > 1000 ? 5 : 1)"
      },
      {
        "name": "full-name",
        "expression": "#firstName + '\'' '\'' + #lastName"
      }
    ],
    "context": {
      "amount": 1000.0,
      "rate": 0.05,
      "fee": 25.0,
      "age": 25,
      "firstName": "John",
      "lastName": "Doe"
    }
  }'
```

Batch response:


```
{
  "success": true,
  "results": [
    {
      "name": "total-calculation",
      "expression": "#amount * #rate + #fee",
      "result": 75.0,
      "resultType": "Double",
      "success": true
    },
    {
      "name": "age-check",
      "expression": "#age >= 18",
      "result": true,
      "resultType": "Boolean",
      "success": true
    },
    {
      "name": "risk-score",
      "expression": "#amount > 10000 ? 10 : (#amount > 1000 ? 5 : 1)",
      "result": 5,
      "resultType": "Integer",
      "success": true
    },
    {
      "name": "full-name",
      "expression": "#firstName + ' ' + #lastName",
      "result": "John Doe",
      "resultType": "String",
```

```
      "success": true
    }
  ],
  "summary": {
    "total": 4,
    "successful": 4,
    "failed": 0
  },
  "timestamp": "2025-08-23T10:30:00Z"
}
```

Expression API Tips

1. **Always validate complex expressions first** using `/validate`
2. **Use meaningful names** in batch operations for easier debugging
3. **Include metrics** when optimizing performance
4. **Test with sample data** before using in production
5. **Check available functions** with `/functions` when building expressions

Next, let's explore data transformation - where expressions really shine!

11. Data Transformation API

The Transformation API lets you clean, normalize, and transform data using either pre-registered transformers or dynamic rules. This is perfect for data preparation and normalization.

Base Path: `/api/transformations`

Method	Endpoint	Description	Complexity
GET	`/transformers`	Get registered transformers	⭐ Beginner
POST	`/{transformerName}`	Transform with registered transformer	⭐⭐ Intermediate
POST	`/dynamic`	Transform with dynamic rules	⭐⭐⭐ Advanced
POST	`/{transformerName}/detailed`	Transform with detailed result	⭐⭐⭐ Advanced

⭐ Beginner: Get Available Transformers

See what pre-built transformers are available:

```
curl -X GET http://localhost:8080/api/transformations/transformers
```

Response:

```
{
  "success": true,
  "transformers": [
    {
      "name": "customer-normalizer",
      "description": "Normalizes customer names and email addresses",
```

```
      "inputFields": ["firstName", "lastName", "email"],
      "outputFields": ["firstName", "lastName", "email"]
    },
    {
      "name": "address-formatter",
      "description": "Formats and validates addresses",
      "inputFields": ["street", "city", "state", "zip"],
      "outputFields": ["formattedAddress", "isValid"]
    }
  ],
  "count": 2,
  "timestamp": "2025-08-23T10:30:00Z"
}
```

⭐⭐ Intermediate: Use Registered Transformer

Apply a pre-built transformer to your data:

```
curl -X POST http://localhost:8080/api/transformations/customer-normalizer \
  -H "Content-Type: application/json" \
  -d '{
    "firstName": "john",
    "lastName": "DOE",
    "email": "JOHN.DOE@EXAMPLE.COM",
    "phone": "1234567890"
  }'
```

Response:

```
{
  "success": true,
  "transformerName": "customer-normalizer",
  "originalData": {
    "firstName": "john",
    "lastName": "DOE",
    "email": "JOHN.DOE@EXAMPLE.COM",
    "phone": "1234567890"
  },
  "transformedData": {
    "firstName": "John",
    "lastName": "Doe",
    "email": "john.doe@example.com",
    "phone": "1234567890"
  },
  "transformationsApplied": [
    "firstName: capitalized first letter",
    "lastName: proper case formatting",
    "email: converted to lowercase"
  ],
  "timestamp": "2025-08-23T10:30:00Z"
}
```

⭐⭐⭐ Advanced: Dynamic Transformation

Create custom transformation rules on the fly:

```
curl -X POST http://localhost:8080/api/transformations/dynamic \
  -H "Content-Type: application/json" \
  -d '{
    "data": {
      "firstName": "john",
      "lastName": "doe",
      "email": "JOHN.DOE@EXAMPLE.COM",
      "phone": "1-234-567-8900",
      "amount": "1500.50"
    },
    "transformerRules": [
      {
        "name": "normalize-firstName",
        "condition": "#firstName != null",
        "transformation": "#firstName.substring(0,1).toUpperCase() + #firstName.substring(1).toLowerCase()",
        "targetField": "firstName"
      },
      {
        "name": "normalize-lastName",
        "condition": "#lastName != null",
        "transformation": "#lastName.substring(0,1).toUpperCase() + #lastName.substring(1).toLowerCase()",
        "targetField": "lastName"
      },
      {
        "name": "normalize-email",
        "condition": "#email != null",
        "transformation": "#email.toLowerCase().trim()",
        "targetField": "email"
      },
      {
        "name": "clean-phone",
        "condition": "#phone != null",
        "transformation": "#phone.replaceAll('\''[^0-9]'\'', '\'''\'')",
        "targetField": "cleanPhone"
      },
      {
        "name": "parse-amount",
        "condition": "#amount != null",
        "transformation": "T(java.lang.Double).parseDouble(#amount)",
        "targetField": "numericAmount"
      }
    ]
  }'
```

Response shows all transformations:


```
{
  "success": true,
  "originalData": {
    "firstName": "john",
    "lastName": "doe",
    "email": "JOHN.DOE@EXAMPLE.COM",
    "phone": "1-234-567-8900",
    "amount": "1500.50"
  },
  "transformedData": {
    "firstName": "John",
    "lastName": "Doe",
    "email": "john.doe@example.com",
    "phone": "1-234-567-8900",
    "cleanPhone": "12345678900",
    "amount": "1500.50",
```

```
    "numericAmount": 1500.5
  },
  "transformationResults": [
    {
      "ruleName": "normalize-firstName",
      "applied": true,
      "originalValue": "john",
      "transformedValue": "John"
    },
    {
      "ruleName": "normalize-lastName",
      "applied": true,
      "originalValue": "doe",
      "transformedValue": "Doe"
    },
    {
      "ruleName": "normalize-email",
      "applied": true,
      "originalValue": "JOHN.DOE@EXAMPLE.COM",
      "transformedValue": "john.doe@example.com"
    },
    {
      "ruleName": "clean-phone",
      "applied": true,
      "originalValue": "1-234-567-8900",
      "transformedValue": "12345678900"
    },
    {
      "ruleName": "parse-amount",
      "applied": true,
      "originalValue": "1500.50",
      "transformedValue": 1500.5
    }
  ],
  "timestamp": "2025-08-23T10:30:00Z"
}
```


Common Transformation Patterns

Name Normalization


```
{
  "name": "normalize-name",
  "condition": "#name != null && #name.length() > 0",
  "transformation": "#name.substring(0,1).toUpperCase() + #name.substring(1).toLowerCase().trim()",
  "targetField": "name"
}
```


Email Cleaning


```
{
  "name": "clean-email",
  "condition": "#email != null",
  "transformation": "#email.toLowerCase().trim()",
  "targetField": "email"
}
```


Phone Number Cleaning

```
{
  "name": "clean-phone",
  "condition": "#phone != null",
  "transformation": "#phone.replaceAll('[^0-9]', '')",
  "targetField": "cleanPhone"
}
```

Currency Formatting

```
{
  "name": "format-currency",
  "condition": "#amount != null",
  "transformation": "T(java.text.NumberFormat).getCurrencyInstance().format(#amount)",
  "targetField": "formattedAmount"
}
```

Date Parsing

```
{
  "name": "parse-date",
  "condition": "#dateString != null",
  "transformation": "T(java.time.LocalDate).parse(#dateString)",
  "targetField": "parsedDate"
}
```

⭐⭐⭐ Advanced: Detailed Transformation Results

Get detailed information about the transformation process:

```
curl -X POST http://localhost:8080/api/transformations/customer-normalizer/detailed \
  -H "Content-Type: application/json" \
  -d '{
    "firstName": "john",
    "lastName": "DOE",
    "email": "JOHN.DOE@EXAMPLE.COM"
  }'
```

Detailed response includes step-by-step information:

```
{
  "success": true,
  "transformerName": "customer-normalizer",
  "originalData": {
    "firstName": "john",
    "lastName": "DOE",
    "email": "JOHN.DOE@EXAMPLE.COM"
  },
  "transformedData": {
    "firstName": "John",
    "lastName": "Doe",
    "email": "john.doe@example.com"
  },
  "detailedResults": {
    "totalRulesApplied": 3,
```

```
    "executionTimeMs": 15,
    "transformationSteps": [
      {
        "step": 1,
        "ruleName": "firstName-capitalization",
        "condition": "#firstName != null",
        "transformation": "#firstName.substring(0,1).toUpperCase() + #firstName.substring(1).toLowerCase()",
        "beforeValue": "john",
        "afterValue": "John",
        "executionTimeMs": 3
      },
      {
        "step": 2,
        "ruleName": "lastName-capitalization",
        "condition": "#lastName != null",
        "transformation": "#lastName.substring(0,1).toUpperCase() + #lastName.substring(1).toLowerCase()",
        "beforeValue": "DOE",
        "afterValue": "Doe",
        "executionTimeMs": 2
      },
      {
        "step": 3,
        "ruleName": "email-normalization",
        "condition": "#email != null",
        "transformation": "#email.toLowerCase()",
        "beforeValue": "JOHN.DOE@EXAMPLE.COM",
        "afterValue": "john.doe@example.com",
        "executionTimeMs": 1
      }
    ]
  },
  "timestamp": "2025-08-23T10:30:00Z"
}
```


Transformation API Tips

1. **Start with registered transformers** for common operations
2. **Use dynamic rules** for custom or one-off transformations
3. **Test transformations** with sample data first
4. **Chain transformations** by using the output of one as input to another
5. **Include detailed results** when debugging transformation logic

Ready to learn about enriching data with external information? That's next!

12. Object Enrichment API

The Enrichment API adds additional data to your objects from external sources, datasets, or computed values. This is perfect for adding context, lookup data, or calculated fields.

Base Path: `/api/enrichment`

Get Predefined Configurations

```
GET /api/enrichment/configurations
```

Response:

```
{
  "success": true,
  "configurations": ["customer-profile", "trade-enrichment"],
  "count": 2,
  "timestamp": "2024-01-15T10:30:00Z"
}
```

Enrich Object

```
POST /api/enrichment/enrich
```

Request Body:

```
{
  "targetObject": {
    "customerId": "CUST001",
    "transactionAmount": 1500.0
  },
  "yamlConfiguration": "metadata:\n  name: \"Customer Enrichment\"\n  version: \"1.0.0\"\n\nenrichments:\n  - name: \"cus
}
```

Batch Enrichment

```
POST /api/enrichment/batch
```

Request Body:

```
{
  "targetObjects": [
    {"customerId": "CUST001", "amount": 1000},
    {"customerId": "CUST002", "amount": 2500}
  ],
  "yamlConfiguration": "..."
}
```

Template Processing API

Process templates with SpEL expressions for JSON, XML, and text formats.

Base Path: `/api/templates`

Process JSON Template

```
POST /api/templates/json
```

Request Body:

```
{
  "template": "{\n  \"customerId\": \"#{#customerId}\",\n  \"customerName\": \"#{#customerName}\",\n  \"totalAmount\": #{
  "context": {
    "customerId": "CUST001",
    "customerName": "John Doe",
    "totalAmount": 1500.0,
    "amount": 1500.0
  }
}
```

Process XML Template

```
POST /api/templates/xml
```

Request Body:

```
{
  "template": "<?xml version=\"1.0\"?>\n<customer>\n  <id>#{#customerId}</id>\n  <name>#{#customerName}</name>\n  <amount
  "context": {
    "customerId": "CUST001",
    "customerName": "John Doe",
    "totalAmount": 1500.0
  }
}
```

Process Text Template

```
POST /api/templates/text
```

Batch Template Processing

```
POST /api/templates/batch
```

Request Body:

```
{
  "templates": [
    {
      "name": "customer-json",
      "type": "JSON",
      "template": "{\"id\": \"#{#id}\"}"
    },
    {
      "name": "customer-xml",
      "type": "XML",
      "template": "<id>#{#id}</id>"
    }
  ],
  "context": {
    "id": "CUST001"
  }
```

```
}
```

Data Source Management API

Manage and interact with external data sources.

Base Path: `/api/datasources`

Get All Data Sources

```
GET /api/datasources
```

Response:

```
{
  "success": true,
  "dataSources": [
    {
      "name": "customerLookup",
      "type": "MockDataSource",
      "description": "Mock data source for testing",
      "available": true
    }
  ],
  "count": 1,
  "timestamp": "2024-01-15T10:30:00Z"
}
```

Get Specific Data Source

```
GET /api/datasources/{name}
```

Test Data Source

```
POST /api/datasources/{name}/test
```

Request Body:

```
{
  "testKey": "CUST001",
  "expectedFields": ["customerName", "customerTier"]
}
```

Perform Lookup

```
POST /api/datasources/{name}/lookup
```

Request Body:

```
{
  "key": "CUST001"
}
```

Response:

```
{
  "success": true,
  "dataSource": "customerLookup",
  "key": "CUST001",
  "result": {
    "customerName": "John Doe",
    "customerTier": "GOLD",
    "riskRating": "LOW"
  },
  "responseTimeMs": 45,
  "timestamp": "2024-01-15T10:30:00Z"
}
```

Expression Evaluation API

Evaluate Spring Expression Language (SpEL) expressions.

Base Path: `/api/expressions`

Evaluate Expression

```
POST /api/expressions/evaluate
```

Request Body:

```
{
  "expression": "#amount * #rate + #fee",
  "context": {
    "amount": 1000.0,
    "rate": 0.05,
    "fee": 25.0
  }
}
```

Response:

```
{
  "success": true,
  "expression": "#amount * #rate + #fee",
  "context": {
    "amount": 1000.0,
    "rate": 0.05,
    "fee": 25.0
  },
  "result": 75.0,
  "resultType": "Double",
  "timestamp": "2024-01-15T10:30:00Z"
```

```
}
```

Evaluate with Detailed Result

```
POST /api/expressions/evaluate/detailed
```

Batch Expression Evaluation

```
POST /api/expressions/batch
```

Request Body:

```
{
  "expressions": [
    {
      "name": "total-calculation",
      "expression": "#amount * #rate + #fee"
    },
    {
      "name": "age-check",
      "expression": "#age >= 18"
    }
  ],
  "context": {
    "amount": 1000.0,
    "rate": 0.05,
    "fee": 25.0,
    "age": 25
  }
}
```

Validate Expression Syntax

```
POST /api/expressions/validate
```

Request Body:

```
{
  "expression": "#amount > 1000 && #currency == 'USD'"
}
```

Get Available Functions

```
GET /api/expressions/functions
```

Response:

```
{
  "success": true,
  "functions": {
    "mathematical": ["abs(number)", "ceil(number)", "floor(number)"],
    "string": ["length()", "substring(start, end)", "toLowerCase()"],
    "logical": ["&&", "||", "!"],
    "comparison": ["==", "!=", "<", ">", "<=", ">="]
  },
  "timestamp": "2024-01-15T10:30:00Z"
}
```


Rules Execution API

Execute business rules individually or in batches.

Base Path: `/api/rules`

Execute Single Rule

```
POST /api/rules/execute
```

Request Body:


```
{
  "rule": {
    "name": "high-value-transaction",
    "condition": "#amount > 1000 && #currency == 'USD'",
    "message": "High value USD transaction detected"
  },
  "facts": {
    "amount": 1500.0,
    "currency": "USD",
    "customerTier": "GOLD"
  }
}
```


Response:


```
{
  "success": true,
  "facts": {
    "amount": 1500.0,
    "currency": "USD",
    "customerTier": "GOLD"
  },
  "result": {
    "triggered": true,
    "ruleName": "high-value-transaction",
    "message": "High value USD transaction detected",
    "resultType": "MATCH",
    "timestamp": "2024-01-15T10:30:00Z"
  },
  "timestamp": "2024-01-15T10:30:00Z"
}
```

Execute Batch Rules

```
POST /api/rules/batch
```

Request Body:

```
{
  "rules": [
    {
      "name": "high-value",
      "condition": "#amount > 1000",
      "message": "High value transaction"
    },
    {
      "name": "gold-customer",
      "condition": "#customerTier == 'GOLD'",
      "message": "Gold tier customer"
    }
  ],
  "facts": {
    "amount": 1500.0,
    "customerTier": "GOLD"
  }
}
```

Response:

```
{
  "success": true,
  "totalRules": 2,
  "triggeredRules": 2,
  "facts": {
    "amount": 1500.0,
    "customerTier": "GOLD"
  },
  "results": [
    {
      "triggered": true,
      "ruleName": "high-value",
      "message": "High value transaction"
    },
    {
      "triggered": true,
      "ruleName": "gold-customer",
      "message": "Gold tier customer"
    }
  ],
  "timestamp": "2024-01-15T10:30:00Z"
}
```

Error Handling

The API uses standard HTTP status codes and provides detailed error information in the response body.

HTTP Status Codes

- `200 OK` - Request successful
- `400 Bad Request` - Invalid request data or parameters
- `404 Not Found` - Resource not found (e.g., transformer, data source)
- `500 Internal Server Error` - Server error during processing

Error Response Format

```
{
  "success": false,
  "error": "Error category",
  "message": "Detailed error description",
  "timestamp": "2024-01-15T10:30:00Z",
  "additionalInfo": {
    "field": "specific error details"
  }
}
```

Common Error Scenarios

Validation Errors

```
{
  "success": false,
  "error": "Validation failed",
  "message": "Expression cannot be null or empty",
  "timestamp": "2024-01-15T10:30:00Z"
}
```

Resource Not Found

```
{
  "success": false,
  "error": "Transformer not found",
  "message": "No transformer found with name: invalid-transformer",
  "transformerName": "invalid-transformer",
  "timestamp": "2024-01-15T10:30:00Z"
}
```

Expression Evaluation Error

```
{
  "success": false,
  "error": "Expression evaluation failed",
  "expression": "invalid && syntax >",
  "message": "Unexpected token at position 15",
  "timestamp": "2024-01-15T10:30:00Z"
}
```

Best Practices

Request Design

1. **Use Meaningful Names**: Choose descriptive names for rules, transformers, and templates
2. **Validate Input**: Always validate expressions and configurations before sending
3. **Handle Errors Gracefully**: Implement proper error handling in your client applications
4. **Use Batch Operations**: For multiple operations, use batch endpoints for better performance

Performance Optimization

1. **Cache Results**: Cache frequently used transformation and enrichment results
2. **Limit Batch Sizes**: Keep batch operations under 100 items for optimal performance
3. **Use Appropriate Timeouts**: Set reasonable timeouts for long-running operations
4. **Monitor Performance**: Use the performance metrics in responses to optimize

Security Considerations

1. **Validate Expressions**: Always validate SpEL expressions to prevent code injection
2. **Sanitize Input**: Sanitize all input data, especially in templates
3. **Limit Expression Complexity**: Avoid overly complex expressions that could cause performance issues
4. **Use HTTPS**: Always use HTTPS in production environments

Expression Guidelines

1. **Use Variable Prefixes**: Always prefix variables with `#` in SpEL expressions
2. **Handle Null Values**: Check for null values in expressions: `#value != null && #value > 0`
3. **Use Type-Safe Operations**: Be explicit about data types in expressions
4. **Test Expressions**: Use the validation endpoint to test expressions before use

Examples & Workflows

Complete Customer Onboarding Workflow

This example demonstrates a complete customer onboarding process using multiple API endpoints.

Step 1: Transform Raw Customer Data

```
curl -X POST http://localhost:8080/api/transformations/dynamic \
  -H "Content-Type: application/json" \
  -d '{
    "data": {
      "first_name": "john",
      "last_name": "DOE",
      "email_address": "JOHN.DOE@EXAMPLE.COM",
      "phone": "1234567890"
    },
    "transformerRules": [
      {
        "name": "normalize-firstName",
        "condition": "#first_name != null",
        "transformation": "#first_name.substring(0,1).toUpperCase() + #first_name.substring(1).toLowerCase()",
        "targetField": "firstName"
      },
      {
        "name": "normalize-lastName",
        "condition": "#last_name != null",
        "transformation": "#last_name.substring(0,1).toUpperCase() + #last_name.substring(1).toLowerCase()",
```

```
        "targetField": "lastName"
      },
      {
        "name": "normalize-email",
        "condition": "#email_address != null",
        "transformation": "#email_address.toLowerCase()",
        "targetField": "email"
      }
    ]
  }'
```

Step 2: Enrich Customer Data

```
curl -X POST http://localhost:8080/api/enrichment/enrich \
  -H "Content-Type: application/json" \
  -d '{
    "targetObject": {
      "customerId": "CUST001",
      "firstName": "John",
      "lastName": "Doe",
      "email": "john.doe@example.com"
    },
    "yamlConfiguration": "metadata:\n  name: \"Customer Profile Enrichment\"\n  version: \"1.0.0\"\n\nenrichments:\n  - n
  }'
```

Step 3: Apply Business Rules

```
curl -X POST http://localhost:8080/api/rules/batch \
  -H "Content-Type: application/json" \
  -d '{
    "rules": [
      {
        "name": "high-value-customer",
        "condition": "#accountBalance > 10000",
        "message": "High value customer identified"
      },
      {
        "name": "gold-tier-customer",
        "condition": "#customerTier == '\''GOLD'\''",
        "message": "Gold tier customer benefits apply"
      },
      {
        "name": "low-risk-customer",
        "condition": "#riskRating == '\''LOW'\''",
        "message": "Low risk customer - expedited processing"
      }
    ],
    "facts": {
      "customerId": "CUST001",
      "customerTier": "GOLD",
      "riskRating": "LOW",
      "accountBalance": 15000.0
    }
  }'
```

Step 4: Generate Welcome Email Template

```
curl -X POST http://localhost:8080/api/templates/text \
  -H "Content-Type: application/json" \
  -d '{
    "template": "Dear #{#firstName} #{#lastName},\n\nWelcome to our #{#customerTier} tier program!\n\nYour current accoun
    "context": {
      "firstName": "John",
      "lastName": "Doe",
      "customerTier": "GOLD",
      "riskRating": "LOW",
      "accountBalance": 15000.0
    }
  }'
```

Financial Transaction Risk Assessment

This example shows how to assess transaction risk using expressions and rules.

Step 1: Calculate Risk Scores

```
curl -X POST http://localhost:8080/api/expressions/batch \
  -H "Content-Type: application/json" \
  -d '{
    "expressions": [
      {
        "name": "amount-risk",
        "expression": "#amount > 10000 ? '\''HIGH'\'' : (#amount > 1000 ? '\''MEDIUM'\'' : '\''LOW'\'')"
      },
      {
        "name": "velocity-risk",
        "expression": "#dailyTransactionCount > 10 ? '\''HIGH'\'' : '\''LOW'\''"
      },
      {
        "name": "location-risk",
        "expression": "#country == '\''US'\'' ? '\''LOW'\'' : '\''MEDIUM'\''"
      },
      {
        "name": "overall-score",
        "expression": "#amount * 0.3 + #dailyTransactionCount * 2 + (#country == '\''US'\'' ? 0 : 10)"
      }
    ],
    "context": {
      "amount": 5000.0,
      "currency": "USD",
      "dailyTransactionCount": 3,
      "country": "US",
      "customerId": "CUST001"
    }
  }'
```

Step 2: Apply Transaction Rules

```
curl -X POST http://localhost:8080/api/rules/execute \
  -H "Content-Type: application/json" \
  -d '{
    "rule": {
      "name": "high-value-transaction",
      "condition": "#amount > 1000 && #currency == '\''USD'\'' && #country == '\''US'\''",
      "message": "High value domestic USD transaction requires additional verification"
```

```
    },
    "facts": {
      "amount": 5000.0,
      "currency": "USD",
      "country": "US",
      "customerId": "CUST001"
    }
  }'
```

Step 3: Generate Transaction Alert

```
curl -X POST http://localhost:8080/api/templates/json \
  -H "Content-Type: application/json" \
  -d '{
    "template": "{\n  \"alertId\": \"#{T(java.util.UUID).randomUUID().toString()}\",\n  \"transactionId\": \"TXN-#{#custo
    "context": {
      "customerId": "CUST001",
      "amount": 5000.0,
      "currency": "USD",
      "country": "US"
    }
  }'
```

Data Source Integration Example

This example demonstrates how to work with external data sources.

Step 1: List Available Data Sources

```
curl -X GET http://localhost:8080/api/datasources
```

Step 2: Test Data Source Connectivity

```
curl -X POST http://localhost:8080/api/datasources/customerLookup/test \
  -H "Content-Type: application/json" \
  -d '{
    "testKey": "CUST001",
    "expectedFields": ["customerName", "customerTier", "riskRating"]
  }'
```

Step 3: Perform Customer Lookup

```
curl -X POST http://localhost:8080/api/datasources/customerLookup/lookup \
  -H "Content-Type: application/json" \
  -d '{
    "key": "CUST001"
  }'
```

Expression Validation and Testing

This example shows how to validate and test SpEL expressions.

Step 1: Get Available Functions

```
curl -X GET http://localhost:8080/api/expressions/functions
```

Step 2: Validate Expression Syntax

```
curl -X POST http://localhost:8080/api/expressions/validate \
  -H "Content-Type: application/json" \
  -d '{
    "expression": "#customer.tier == '\''GOLD'\'' && #transaction.amount > 1000"
  }'
```

Step 3: Test Expression with Sample Data

```
curl -X POST http://localhost:8080/api/expressions/evaluate \
  -H "Content-Type: application/json" \
  -d '{
    "expression": "#customer.tier == '\''GOLD'\'' && #transaction.amount > 1000",
    "context": {
      "customer": {
        "tier": "GOLD",
        "id": "CUST001"
      },
      "transaction": {
        "amount": 1500.0,
        "currency": "USD"
      }
    }
  }'
```

Multi-Template Processing Workflow

This example demonstrates processing multiple template types in a single request.

```
curl -X POST http://localhost:8080/api/templates/batch \
  -H "Content-Type: application/json" \
  -d '{
    "templates": [
      {
        "name": "customer-json",
        "type": "JSON",
        "template": "{\n  \"customerId\": \"#{#customerId}\",\n  \"fullName\": \"#{#firstName} #{#lastName}\",\n  \"statu
      },
      {
        "name": "customer-xml",
        "type": "XML",
        "template": "<?xml version=\"1.0\"?>\n<customer>\n  <id>#{#customerId}</id>\n  <name>#{#firstName} #{#lastName}</
      },
      {
        "name": "customer-email",
        "type": "TEXT",
        "template": "Dear #{#firstName} #{#lastName},\n\nYour account #{#customerId} is currently #{#isActive ? '\''activ
      }
    ],
    "context": {
      "customerId": "CUST001",
```

```
      "firstName": "John",
      "lastName": "Doe",
      "isActive": true
    }
  }'
```

Advanced Usage Patterns

Chaining API Calls

You can chain multiple API calls to create sophisticated workflows:

1. **Transform → Enrich → Apply Rules → Generate Templates**
2. **Validate Expressions → Evaluate → Apply to Rules**
3. **Test Data Sources → Lookup Data → Enrich Objects**

Error Recovery Strategies

1. **Graceful Degradation**: If enrichment fails, continue with available data
2. **Retry Logic**: Implement exponential backoff for transient failures
3. **Fallback Rules**: Use simpler rules if complex expressions fail
4. **Partial Processing**: Process successful items in batch operations

Performance Monitoring

Monitor these key metrics:

- Response times for each endpoint
- Success/failure rates for batch operations
- Expression evaluation performance
- Data source lookup times

Troubleshooting

Common Issues

1. **Expression Syntax Errors**: Use the validation endpoint first
2. **Data Source Timeouts**: Check connectivity and increase timeouts
3. **Template Processing Failures**: Validate template syntax and context data
4. **Batch Operation Limits**: Reduce batch sizes if experiencing timeouts

Debug Tips

1. **Enable Debug Logging**: Set logging level to DEBUG for detailed information
2. **Test Individual Components**: Test expressions, rules, and templates separately
3. **Use Swagger UI**: Interactive testing through the built-in documentation
4. **Check Response Times**: Monitor performance metrics in responses

Conclusion

The APEX Rules Engine REST API provides a comprehensive set of endpoints for building sophisticated rule-based applications. By combining transformation, enrichment, template processing, and rule execution capabilities, you can create powerful workflows that handle complex business logic with ease.

Key Benefits

- **Modular Design**: Use individual endpoints or combine them in workflows
- **Flexible Configuration**: Support for dynamic rules and configurations
- **Performance Optimized**: Batch operations and performance monitoring
- **Production Ready**: Comprehensive error handling and validation
- **Well Documented**: Complete API documentation with examples

Next Steps

1. **Explore the Swagger UI**: Visit `/swagger-ui.html` for interactive documentation
2. **Run the Examples**: Try the workflow examples provided in this guide
3. **Build Custom Workflows**: Combine endpoints to create your own business processes
4. **Monitor Performance**: Use the built-in metrics to optimize your applications

For more information, consult the individual controller documentation in the source code or reach out to the development team for support.