APEX Rules Engine REST API Guide

Version: 1.0 **Date:** 2025-08-02 **Author:** Mark Andrew Ray-Smith Cityline Ltd

Table of Contents

Overview

The APEX Rules Engine REST API provides comprehensive endpoints for:

- **Data Transformation** - Transform and normalize data using configurable rules
- **Object Enrichment** - Enrich objects with additional data from external sources
- **Template Processing** - Process JSON, XML, and text templates with SpEL expressions
- **Data Source Management** - Manage and interact with external data sources
- **Expression Evaluation** - Evaluate Spring Expression Language (SpEL) expressions
- **Rules Execution** - Execute business rules individually or in batches

Base URL

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

Content Type

All API endpoints accept and return JSON unless otherwise specified:

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

Getting Started

Prerequisites

- Java 17 or higher
- Spring Boot 3.x
- APEX Rules Engine Core modules

Quick Start

1. Start the APEX REST API application:

```
mvn spring-boot:run -pl apex-rest-api
```

2. Access the Swagger UI documentation:

```
http://localhost:8080/swagger-ui.html
```

3. Test a simple expression evaluation:

```
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
    }
  }'
```

Authentication & Security

Currently, the API operates without authentication for development purposes. For production deployments:

- Implement OAuth 2.0 or JWT-based authentication
- Use HTTPS for all communications
- Implement rate limiting
- Validate and sanitize all inputs

API Endpoints

Transformation API

Transform data using registered transformers or dynamic rules.

Base Path: `/api/transformations`

Get Registered Transformers

```
GET /api/transformations/transformers
```

Response:

```
{
  "success": true,
  "transformers": ["customer-normalizer", "address-formatter"],
  "count": 2,
  "timestamp": "2024-01-15T10:30:00Z"
}
```

Transform Data with Registered Transformer

```
POST /api/transformations/{transformerName}
```

Request Body:

```
{
  "firstName": "john",
  "lastName": "doe",
  "email": "JOHN.DOE@EXAMPLE.COM"
}
```

Response:

```
{
  "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"
  },
  "timestamp": "2024-01-15T10:30:00Z"
}
```

Transform with Dynamic Rules

```
POST /api/transformations/dynamic
```

Request Body:

```
{
  "data": {
```

```
    "firstName": "john",
    "lastName": "doe",
    "email": "JOHN.DOE@EXAMPLE.COM"
  },
  "transformerRules": [
    {
      "name": "normalize-firstName",
      "condition": "#firstName != null",
      "transformation": "#firstName.substring(0,1).toUpperCase() + #firstName.substring(1).toLowerCase()",
      "targetField": "firstName"
    },
    {
      "name": "normalize-email",
      "condition": "#email != null",
      "transformation": "#email.toLowerCase()",
      "targetField": "email"
    }
  ]
}
```

Enrichment API

Enrich objects with additional data using YAML configurations.

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.