External Data Source Integration - Complete Guide

Welcome to the comprehensive guide for APEX's external data source integration! This guide will help you connect APEX to your existing systems and data sources, making your rules more powerful and dynamic.

What you'll learn: This guide covers everything you need to know about connecting APEX to external systems - from simple database connections to complex multi-source integrations. Whether you're a developer setting up your first data source or an architect designing enterprise-scale integrations, you'll find practical guidance and real-world examples.

Why external data sources matter: Instead of having static rules that only work with the data you provide, external data sources let your rules access live data from your databases, APIs, files, and other systems. This means your rules can make decisions based on current customer information, real-time market data, or any other external information your business needs.

Integration with Scenarios: APEX's scenario-based configuration system provides powerful routing capabilities for external data sources. Different scenarios can use different data source configurations, enabling environment-specific setups (dev/test/prod), jurisdiction-specific data sources, and business-domain-specific integrations. This guide shows how to configure external data sources that work seamlessly with APEX's scenario management system.

Table of Contents

Overview

APEX's external data source integration is designed to make connecting to your existing systems as simple and reliable as possible. Think of it as a universal adapter that lets your rules talk to any system in your organization.

What External Data Sources Give You

Multiple Data Source Types - Connect to the systems you already use:

- **Databases**: PostgreSQL, MySQL, Oracle, SQL Server, H2 - all with connection pooling and optimization
- **REST APIs**: Any HTTP/HTTPS service with authentication, retries, and circuit breakers
- **File Systems**: CSV, JSON, XML files with automatic parsing and monitoring
- **Caches**: High-speed in-memory storage for frequently accessed data
- **Custom Sources**: Build your own connectors for specialized systems

Unified Interface - Learn once, use everywhere: All data sources work the same way in your rules, regardless of whether you're connecting to a database, API, or file. This means less complexity and more consistency in your code.

Enterprise Features - Production-ready from day one:

- **Connection pooling**: Efficiently manage database connections for high performance
- **Health monitoring**: Automatically detect and handle system failures
- **Caching**: Store frequently accessed data in memory for faster access
- **Circuit breakers**: Protect your systems from cascading failures
- **Load balancing**: Distribute requests across multiple data sources
- **Automatic recovery**: Reconnect automatically when systems come back online

YAML Configuration - Simple, readable setup: Configure everything in human-readable YAML files with environment-specific overrides. No complex programming required for basic setups.

High Availability - Built for mission-critical systems: Load balancing, automatic failover, and recovery mechanisms ensure your rules keep working even when individual systems have problems.

Performance Monitoring - Know what's happening: Comprehensive metrics and statistics help you understand performance, identify bottlenecks, and optimize your integrations.

Thread Safety - Concurrent access without worries: All components are designed for multi-threaded environments with proper synchronization, so you can safely use them in high-concurrency applications.

Quick Start

Let's get you connected to your first external data source in just a few minutes! This example shows how to connect to a PostgreSQL database, but the same principles apply to all data source types.

1. Add Dependencies

First, make sure you have the APEX core dependency in your project:

```
<dependency>
    <groupId>dev.mars.rulesengine</groupId>
    <artifactId>rules-engine-core</artifactId>
    <version>1.0.0</version>
</dependency>
```

What this gives you: All the external data source integration capabilities are included in the core APEX library - no additional dependencies needed for basic functionality.

2. Create Your Configuration File

Create a YAML file that describes your data source. Don't worry about understanding every option - we'll explain the key parts:

```
# data-sources.yaml
name: "My Application"                    # A friendly name for your configuration
version: "1.0.0"                         # Version for tracking changes

dataSources:                             # List of all your data sources
  - name: "user-database"                # Unique name for this data source
    type: "database"                     # Type of data source
    sourceType: "postgresql"             # Specific database type
    enabled: true                        # Turn this data source on/off

    connection:                          # How to connect to your database
      host: "localhost"                  # Database server location
      port: 5432                         # Database port (5432 is PostgreSQL default)
      database: "myapp"                  # Database name
      username: "app_user"               # Database username
      password: "${DB_PASSWORD}"         # Password from environment variable (secure!)

    queries:                             # Named queries you can use in your rules
      getUserById: "SELECT * FROM users WHERE id = :id"
      getAllUsers: "SELECT * FROM users ORDER BY created_at DESC"

    parameterNames:                      # Parameters your queries expect
      - "id"

    cache:                               # Optional caching for better performance
      enabled: true                      # Turn caching on
      ttlSeconds: 300                    # Cache data for 5 minutes
      maxSize: 1000                      # Store up to 1000 cached results
```

Key concepts explained:

- **Environment variables**: `${DB_PASSWORD}` gets the password from your environment, keeping it secure
- **Named queries**: Instead of writing SQL in your Java code, define queries here with descriptive names
- **Parameters**: Use `:id` in queries and list parameter names so APEX knows what to expect
- **Caching**: Automatically cache query results to improve performance

3. Initialize and Use in Your Code

Now you can use your data source in Java code:

```
// Step 1: Load your configuration file
DataSourceConfigurationService configService = DataSourceConfigurationService.getInstance();
YamlRuleConfiguration yamlConfig = loadYamlConfiguration("data-sources.yaml");
configService.initialize(yamlConfig);

// Step 2: Get your data source by name
ExternalDataSource userDb = configService.getDataSource("user-database");

// Step 3: Execute queries with parameters
Map<String, Object> parameters = Map.of("id", 123);
List<Object> results = userDb.query("getUserById", parameters);

// Step 4: Get a single result (useful when you expect one record)
Object user = userDb.queryForObject("getUserById", parameters);
```

What's happening here:

1. **Configuration loading**: APEX reads your YAML file and sets up the data source

2. **Data source retrieval**: Get your configured data source by the name you gave it
3. **Query execution**: Run your named queries with parameters
4. **Result handling**: Get back Java objects you can use in your rules

That's it! You now have a working external data source integration. Your rules can access live database data using simple method calls.

Supported Data Sources

APEX supports a wide variety of data sources, each optimized for different use cases. Here's what's available and when to use each type:

Database Sources - For Structured Data

Perfect for accessing your existing business data stored in relational databases.

Supported Databases:

- **PostgreSQL**: Full-featured support with connection pooling, SSL, and advanced features
- **MySQL**: Complete integration with SSL support and MySQL-specific optimizations
- **Oracle**: Enterprise-grade Oracle database connectivity with connection pooling
- **SQL Server**: Microsoft SQL Server integration with Windows authentication support
- **H2**: Lightweight in-memory and file-based database, perfect for testing and development

When to use database sources:

- Customer information, transaction records, product catalogs
- Any structured data you already store in databases
- When you need ACID transactions and data consistency
- For complex queries with joins and aggregations

Key features: Connection pooling, prepared statements, transaction support, SSL encryption

REST API Sources - For External Services

Connect to web services, microservices, and third-party APIs.

Authentication Methods:

- **Bearer tokens**: For modern APIs with JWT or similar tokens
- **API keys**: Simple key-based authentication in headers or query parameters
- **Basic auth**: Username/password authentication for legacy systems
- **OAuth2**: Full OAuth2 flow support for secure integrations

HTTP Methods: GET, POST, PUT, DELETE, PATCH - all the standard REST operations

Enterprise Features:

- **Circuit breakers**: Protect your system when external APIs fail
- **Retry logic**: Automatically retry failed requests with exponential backoff
- **Response caching**: Cache API responses to improve performance and reduce API calls
- **JSON parsing**: Automatic parsing with JSONPath support for extracting specific data

When to use REST API sources:

- Third-party services (payment processors, address validation, etc.)
- Microservices in your architecture
- Real-time data that changes frequently
- When you need to send data to external systems

File System Sources - For File-Based Data

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

Supported File Formats:

- **CSV Files**: Configurable delimiters, headers, automatic data type conversion
- **JSON Files**: JSONPath extraction, nested object handling, array processing
- **XML Files**: XPath queries, namespace support, attribute extraction
- **Fixed-Width**: Legacy mainframe file formats with column definitions
- **Plain Text**: Log files, unstructured text, custom parsing

Advanced Features:

- **File watching**: Automatically detect when files change
- **Batch processing**: Handle large files efficiently
- **Error handling**: Skip malformed records and continue processing
- **Encoding support**: Handle different character encodings (UTF-8, ISO-8859-1, etc.)

When to use file system sources:

- Daily batch files from other systems
- Configuration files that change periodically
- Log file analysis and monitoring
- Legacy system integration where files are the only interface

Cache Sources - For High-Speed Data Access

Store frequently accessed data in memory for ultra-fast retrieval.

Cache Types:

- **In-Memory**: High-performance local caching with LRU (Least Recently Used) eviction
- **Distributed**: Ready for Redis/Hazelcast integration for multi-server deployments

Key Features:

- **TTL support**: Automatic expiration of cached data
- **Pattern matching**: Find cached items using wildcards
- **Statistics collection**: Monitor cache hit rates and performance
- **Multiple eviction policies**: LRU, LFU, FIFO, and more

When to use cache sources:

- Frequently accessed lookup data (currency rates, product categories)
- Expensive calculation results that don't change often

- Session data and user preferences
- Any data where speed is more important than absolute freshness

Configuration

Configuration is where you tell APEX how to connect to your data sources. APEX uses YAML configuration files because they're human-readable and easy to maintain. Let's explore the key configuration concepts:

Mandatory Metadata Requirements

All external data source configuration files must include proper metadata:

```yaml
metadata:
  name: "Customer Database Configuration"
  version: "1.0.0"
  description: "PostgreSQL connection for customer data lookups"
  type: "rule-config"                        # Required: File type identifier
  author: "data.integration@company.com"     # Required: Configuration author
  created: "2025-08-02"                     # Optional: Creation date
  business-domain: "Customer Management"     # Optional: Business context
  environment: "production"                  # Optional: Target environment
```

Why Metadata Matters for Data Sources:

- **Validation**: Ensures configuration files are properly structured
- **Documentation**: Provides clear identification and purpose
- **Audit Trail**: Tracks who created and modified configurations
- **Environment Management**: Helps identify configuration scope and purpose
- **Automated Processing**: Enables scenario-based routing and validation

Required Fields:

- `name` : Human-readable configuration name
- `version` : Semantic version for change tracking
- `description` : Clear explanation of the data source purpose
- `type` : Must be "rule-config" for data source configurations
- `author` : Email or identifier of the configuration creator

Environment Variables - Keeping Secrets Safe

Why use environment variables? Never put passwords, API keys, or other sensitive information directly in configuration files. Environment variables keep your secrets secure and allow the same configuration to work in different environments.

```yaml
connection:
  username: "app_user"
  password: "${DB_PASSWORD}"  # Gets password from environment variable
  apiKey: "${API_KEY}"        # Gets API key from environment variable
```

How to set environment variables:

```
# Linux/Mac
export DB_PASSWORD="your_secure_password"
export API_KEY="your_api_key"

# Windows
set DB_PASSWORD=your_secure_password
set API_KEY=your_api_key
```

Benefits:

- Secrets never appear in your code or configuration files
- Different environments (dev, test, prod) can use different values
- Easier to rotate passwords and keys without changing code

Environment-Specific Overrides - One Config, Multiple Environments

Instead of maintaining separate configuration files for each environment, use overrides to customize settings:

```
# Base configuration that applies everywhere
dataSources:
  - name: "user-database"
    type: "database"
    sourceType: "postgresql"
    # ... base settings ...

# Environment-specific overrides
environments:
  development:                           # Settings for development environment
    dataSources:
      - name: "user-database"
        connection:
          host: "localhost"              # Use local database in dev
          maxPoolSize: 5                 # Smaller pool for dev
        cache:
          ttlSeconds: 60                 # Short cache time for testing

  production:                            # Settings for production environment
    dataSources:
      - name: "user-database"
        connection:
          host: "prod-db.example.com"    # Production database server
          maxPoolSize: 50                # Larger pool for production load
        cache:
          ttlSeconds: 600                # Longer cache time for performance
          maxSize: 5000                  # More cache entries
```

How it works:

1. APEX loads the base configuration first
2. Then it applies environment-specific overrides based on your current environment
3. Only the specified settings are overridden - everything else stays the same

Health Checks - Monitoring Your Data Sources

Health checks automatically monitor your data sources and detect problems before they affect your rules:

```
healthCheck:
  enabled: true                          # Turn health monitoring on
  intervalSeconds: 30                    # Check every 30 seconds
  timeoutSeconds: 5                      # Wait up to 5 seconds for response
  failureThreshold: 3                    # Mark unhealthy after 3 failures
  recoveryThreshold: 2                   # Mark healthy after 2 successes
  query: "SELECT 1"                      # Simple query to test database
```

What health checks do:

- Regularly test if your data sources are responding
- Automatically mark failed data sources as unhealthy
- Provide metrics on data source availability
- Enable automatic failover to backup data sources

Custom health check queries:

- **Database**: `"SELECT 1"` or `"SELECT COUNT(*) FROM users"`
- **REST API**: A simple GET request to a health endpoint
- **File System**: Check if a directory is accessible
- **Cache**: Test a simple get/put operation

Caching Configuration - Speed Up Your Rules

Caching stores frequently accessed data in memory for much faster access:

```
cache:
  enabled: true                          # Turn caching on
  ttlSeconds: 300                        # Cache data for 5 minutes
  maxSize: 1000                          # Store up to 1000 cached results
  keyPrefix: "myapp"                     # Prefix for cache keys (helps avoid conflicts)
  evictionPolicy: "LRU"                  # Remove least recently used items when full
```

Cache timing guidelines:

- **Frequently changing data**: 30-60 seconds
- **Stable reference data**: 5-60 minutes
- **Configuration data**: 1-24 hours
- **Static lookup data**: Several hours or days

Eviction policies explained:

- **LRU (Least Recently Used)**: Remove items that haven't been accessed recently
- **LFU (Least Frequently Used)**: Remove items that are accessed least often
- **FIFO (First In, First Out)**: Remove the oldest items first
- **TTL-based**: Remove items when they expire (regardless of usage)

Usage Examples

Once you have your data sources configured, using them in your Java code is straightforward. Here are practical examples for each type of data source:

Database Operations - Working with SQL Data

Simple queries without parameters:

```
// Get all users - no parameters needed
List<Object> users = dataSource.query("getAllUsers", Collections.emptyMap());

// The query "getAllUsers" was defined in your YAML configuration
// Returns a List of Map objects, each representing a database row
```

Parameterized queries - the safe way to pass data:

```
// Get a specific user by ID and status
Map<String, Object> params = Map.of(
    "id", 123,
    "status", "active"
);
Object user = dataSource.queryForObject("getUserById", params);

// This executes: SELECT * FROM users WHERE id = :id AND status = :status
// Parameters are safely bound to prevent SQL injection
```

Batch operations for efficiency:

```
// Update multiple records in one database round-trip
List<String> updates = List.of(
    "UPDATE users SET last_login = NOW() WHERE id = 1",
    "UPDATE users SET last_login = NOW() WHERE id = 2",
    "UPDATE users SET last_login = NOW() WHERE id = 3"
);
dataSource.batchUpdate(updates);

// Much faster than executing updates one by one
```

Working with complex parameters:

```
// Create a new user with multiple fields
Map<String, Object> newUserParams = Map.of(
    "username", "john_doe",
    "email", "john@example.com",
    "firstName", "John",
    "lastName", "Doe",
    "status", "ACTIVE"
);
Object result = dataSource.query("createUser", newUserParams);
```

REST API Operations - Calling External Services

GET requests - retrieving data:

```
// Get user profile from external API
Map<String, Object> params = Map.of("userId", 123);
```

```
Object userProfile = apiSource.queryForObject("getUserProfile", params);

// This might call: GET https://api.example.com/users/123
// The URL pattern is defined in your YAML configuration
```

POST requests - sending data:

```
// Create a new user via API (configured in YAML)
Map<String, Object> newUser = Map.of(
    "name", "John Doe",
    "email", "john@example.com",
    "department", "Engineering"
);
Object result = apiSource.query("createUser", newUser);

// This sends a POST request with the user data as JSON
```

Handling API responses:

```
// API calls return parsed JSON as Java objects
Map<String, Object> response = (Map<String, Object>) apiSource.queryForObject("getUserProfile", params);

// Access response fields
String userName = (String) response.get("name");
String email = (String) response.get("email");
Map<String, Object> address = (Map<String, Object>) response.get("address");
```

File System Operations - Processing Files

Reading CSV files:

```
// Read CSV file and get parsed data
Object csvData = fileSource.getData("csv", "users.csv");

// Returns List<Map<String, Object>> where each Map is a CSV row
List<Map<String, Object>> users = (List<Map<String, Object>>) csvData;
for (Map<String, Object> user : users) {
    String name = (String) user.get("name");
    String email = (String) user.get("email");
}
```

Reading JSON configuration files:

```
// Read JSON file with parameters
Map<String, Object> params = Map.of("filename", "config.json");
Object config = fileSource.queryForObject("getConfig", params);

// Access JSON data
Map<String, Object> configData = (Map<String, Object>) config;
String apiUrl = (String) configData.get("apiUrl");
Integer timeout = (Integer) configData.get("timeout");
```

Processing XML files:

```
// Read XML file with XPath query
Map<String, Object> xmlParams = Map.of(
    "filename", "data.xml",
    "xpath", "//user[@status='active']"  // XPath to find active users
);
List<Object> activeUsers = fileSource.query("getActiveUsers", xmlParams);
```

Cache Operations - High-Speed Data Storage

Storing data in cache:

```
// Store user data in cache for fast access
Map<String, Object> params = Map.of(
    "key", "user:123",
    "value", userData
);
cacheSource.query("put", params);

// Data is now stored in memory for fast retrieval
```

Retrieving cached data:

```
// Get data from cache
Map<String, Object> getParams = Map.of("key", "user:123");
Object cachedData = cacheSource.queryForObject("get", getParams);

if (cachedData != null) {
    // Cache hit - use the cached data
    System.out.println("Found in cache: " + cachedData);
} else {
    // Cache miss - need to fetch from original source
    System.out.println("Not in cache, fetching from database...");
}
```

Pattern-based cache operations:

```
// Find all cached user keys
Map<String, Object> patternParams = Map.of("pattern", "user:*");
List<Object> userKeys = cacheSource.query("keys", patternParams);

// Get multiple cached items at once
Map<String, Object> batchParams = Map.of(
    "keys", List.of("user:1", "user:2", "user:3")
);
Map<String, Object> batchResults = (Map<String, Object>) cacheSource.queryForObject("getAll", batchParams);
```

Architecture

Understanding APEX's external data source architecture helps you make better decisions about how to structure your integrations. The architecture is designed around the principle of separation of concerns - each component has a specific job to do.

Core Components - The Building Blocks

Think of these components as different layers in a well-organized system:

1. DataSourceConfigurationService - The Orchestrator This is your main entry point and the component you'll interact with most. It's like a conductor that coordinates all the other components.

- Loads and manages your YAML configuration files
- Provides a simple API for accessing data sources
- Handles environment-specific overrides
- Manages the lifecycle of all data sources

2. DataSourceManager - The Coordinator The manager handles the complex task of coordinating multiple data sources, especially when you have multiple instances of the same type.

- Provides load balancing across multiple data sources
- Handles automatic failover when data sources fail
- Manages health monitoring and recovery
- Coordinates batch operations across data sources

3. DataSourceRegistry - The Directory The registry is like a phone book that keeps track of all your data sources and their current status.

- Maintains a centralized directory of all data sources
- Tracks health status and availability
- Provides fast lookup by name or type
- Manages data source lifecycle events

4. DataSourceFactory - The Builder The factory is responsible for creating and configuring data source instances based on your configuration.

- Creates data source instances from YAML configuration
- Handles type-specific setup and initialization
- Manages resource allocation and cleanup
- Supports custom data source types through plugins

Data Flow - How Requests Work

Here's what happens when your rule needs data from an external source:

```
Your Rule → ConfigurationService → Manager → Registry → DataSource → External System
    ↑                                                                        ↓
    ←←←←←←←←←←←←←←←←←←←← Response Data ←←←←←←←←←←←←←←←←←←←←←←←←←←←←←←
```

Step by step:

1. **Your rule** requests data using a simple method call
2. **ConfigurationService** routes the request to the appropriate manager
3. **Manager** selects the best available data source (load balancing, health checks)
4. **Registry** provides the actual data source instance
5. **DataSource** executes the query against the external system
6. **Response** flows back through the same path with caching applied at appropriate levels

Thread Safety - Built for Concurrency

All APEX components are designed to work safely in multi-threaded environments:

Thread-Safe Data Structures:

- Uses `ConcurrentHashMap` and other thread-safe collections
- Atomic operations for counters and statistics
- Immutable configuration objects that can't be accidentally modified

Connection Management:

- Database connection pooling handles concurrent access automatically
- HTTP clients are thread-safe and can handle multiple simultaneous requests
- File system access is properly synchronized to prevent conflicts

Proper Synchronization:

- Critical sections are protected with appropriate locking mechanisms
- Lock-free algorithms where possible for better performance
- No shared mutable state between threads unless properly synchronized

What this means for you:

- You can safely call data source methods from multiple threads
- No need to worry about synchronization in your application code
- APEX handles all the complexity of concurrent access internally

Best Practices

Following these best practices will help you build reliable, secure, and performant external data source integrations. These recommendations come from real-world experience and will save you time and trouble in the long run.

Configuration Management - Organize for Success

1. Use Environment Variables for Sensitive Data Why: Keeps secrets out of your code and configuration files, making them more secure and easier to manage.

```yaml
# ✅ DO: Use environment variables
connection:
  username: "app_user"
  password: "${DB_PASSWORD}"
  apiKey: "${API_KEY}"

# ❌ DON'T: Hardcode sensitive data
connection:
  password: "hardcoded_password"  # Never do this!
```

2. Use Environment-Specific Configurations Why: One configuration file can work across all your environments (dev, test, prod) with appropriate overrides.

```
# Base configuration
dataSources:
  - name: "user-database"
    # ... common settings ...

# Environment overrides
environments:
  development:
    dataSources:
      - name: "user-database"
        connection:
          host: "localhost"
          maxPoolSize: 5
  production:
    dataSources:
      - name: "user-database"
        connection:
          host: "prod-db.example.com"
          maxPoolSize: 50
```

3. Validate Configurations Before Deployment Why: Catch configuration errors early, before they cause runtime failures.

```
public void validateConfiguration(DataSourceConfiguration config) {
    if (config.getName() == null || config.getName().trim().isEmpty()) {
        throw new IllegalArgumentException("Data source name is required");
    }

    if (config.getConnection() == null) {
        throw new IllegalArgumentException("Connection configuration is required");
    }

    // Add more validation as needed
}
```

4. Document Your Queries and Endpoints Why: Makes maintenance easier and helps other developers understand your integrations.

```
queries:
  getUserById:
    sql: "SELECT * FROM users WHERE id = :id"
    description: "Retrieves a single user by their unique ID"
    parameters: ["id"]
    returns: "Single user object or null if not found"
```

Performance Optimization - Make It Fast

1. Enable Caching with Appropriate TTL Values Why: Reduces load on external systems and improves response times.

```
cache:
  enabled: true
  # Choose TTL based on how often your data changes
  ttlSeconds: 300     # 5 minutes for frequently changing data
  ttlSeconds: 3600    # 1 hour for stable reference data
  ttlSeconds: 86400   # 24 hours for configuration data
```

2. Configure Connection Pooling Based on Load Why: Proper pool sizing prevents connection exhaustion and optimizes resource usage.

```yaml
connection:
  # For high-throughput applications
  maxPoolSize: 50
  minPoolSize: 10

  # For low-latency requirements
  connectionTimeout: 5000
  idleTimeout: 300000
```

3. Use Efficient Queries and Indexes Why: Faster queries mean better performance and lower resource usage.

```yaml
queries:
  # ✅ DO: Use specific columns and indexed fields
  getUserByEmail: "SELECT id, username, email FROM users WHERE email = :email"

  # ❌ DON'T: Use SELECT * or unindexed searches
  getAllUserData: "SELECT * FROM users"
  findUserByName: "SELECT * FROM users WHERE LOWER(name) LIKE '%:name%'"
```

4. Group Operations into Batches Why: Reduces network round-trips and improves throughput.

```java
// ✅ DO: Use batch operations
List<String> updates = List.of(
    "UPDATE users SET last_login = NOW() WHERE id = 1",
    "UPDATE users SET last_login = NOW() WHERE id = 2",
    "UPDATE users SET last_login = NOW() WHERE id = 3"
);
dataSource.batchUpdate(updates);

// ❌ DON'T: Execute operations one by one
for (int id : userIds) {
    dataSource.query("updateLastLogin", Map.of("id", id));
}
```

Error Handling - Plan for Failures

1. Configure Circuit Breakers for External APIs Why: Protects your system from cascading failures when external services are down.

```yaml
circuitBreaker:
  enabled: true
  failureThreshold: 5      # Open circuit after 5 failures
  recoveryTimeout: 30000   # Try again after 30 seconds
  halfOpenMaxCalls: 3      # Test with 3 calls when recovering
```

2. Use Retry Logic with Exponential Backoff Why: Handles transient failures gracefully without overwhelming failing systems.

```yaml
connection:
  retryAttempts: 3
  retryDelay: 1000         # Start with 1 second
  retryMultiplier: 2.0     # Double the delay each time (1s, 2s, 4s)
  maxRetryDelay: 30000     # Cap at 30 seconds
```

3. Monitor Data Source Health Continuously Why: Early detection of problems allows for proactive response.

```
healthCheck:
  enabled: true
  intervalSeconds: 30      # Check every 30 seconds
  timeoutSeconds: 5        # 5 second timeout
  failureThreshold: 3      # Mark unhealthy after 3 failures
```

4. Implement Graceful Degradation Why: Your application can continue working even when some data sources fail.

```
try {
    Object userData = userDatabase.queryForObject("getUserById", params);
    return userData;
} catch (DataSourceException e) {
    // Fallback to cache or default values
    logger.warn("Database unavailable, using cached data", e);
    return userCache.queryForObject("get", Map.of("key", "user:" + userId));
}
```

Security - Protect Your Data

1. Always Use Encrypted Connections in Production Why: Protects sensitive data in transit from interception.

```
connection:
  sslEnabled: true
  sslMode: "require"       # For PostgreSQL
  useSSL: true            # For MySQL
  encrypt: true           # For SQL Server
```

2. Use Strong Authentication Methods Why: Prevents unauthorized access to your data sources.

```
authentication:
  type: "bearer"          # Use token-based auth when possible
  token: "${API_TOKEN}"

  # Or for databases
  type: "certificate"     # Use certificate-based auth for high security
  certificatePath: "/path/to/cert.pem"
```

3. Limit Database Permissions to Minimum Required Why: Reduces the impact of security breaches.

```
-- ✅ DO: Create specific users with limited permissions
CREATE USER app_user WITH PASSWORD 'secure_password';
GRANT SELECT, INSERT, UPDATE ON users TO app_user;
GRANT SELECT ON products TO app_user;

-- ❌ DON'T: Use admin accounts for applications
-- GRANT ALL PRIVILEGES TO app_user;  -- Too much access!
```

4. Log All Data Access for Compliance Why: Provides audit trails required by many regulations and helps with troubleshooting.

```
// Log data access events
logger.info("Data access: user={}, query={}, parameters={}",
    currentUser, queryName, sanitizeParameters(parameters));
```

Monitoring - Know What's Happening

1. Enable Comprehensive Metrics Collection Why: Provides visibility into performance and helps identify issues.

```
// Regularly check and log metrics
DataSourceMetrics metrics = dataSource.getMetrics();
logger.info("Data source performance: success_rate={}%, avg_response_time={}ms, cache_hit_ratio={}%",
    metrics.getSuccessRate() * 100,
    metrics.getAverageResponseTime(),
    metrics.getCacheHitRatio() * 100);
```

2. Set Up Alerts for Health Check Failures Why: Enables rapid response to system problems.

```
// Monitor health and send alerts
if (dataSource.getConnectionStatus().getState() != ConnectionStatus.State.CONNECTED) {
    alertService.sendAlert("Data source unhealthy: " + dataSource.getName());
}
```

3. Monitor Response Times and Throughput Why: Helps identify performance degradation before it affects users.

```
// Track performance trends
if (metrics.getAverageResponseTime() > 1000) {  // 1 second threshold
    logger.warn("Slow response time detected: {}ms", metrics.getAverageResponseTime());
}
```

4. Track Resource Usage for Capacity Planning Why: Helps you plan for growth and avoid resource exhaustion.

```
// Monitor connection pool usage
int activeConnections = metrics.getActiveConnections();
int totalConnections = metrics.getTotalConnections();
double utilization = (double) activeConnections / totalConnections;

if (utilization > 0.8) {  // 80% threshold
    logger.warn("High connection pool utilization: {}%", utilization * 100);
}
```

Troubleshooting

Common Issues

Connection Failures

```
Error: Failed to connect to database
Solution: Check connection parameters, network connectivity, and credentials
```

Cache Misses

```
Issue: Low cache hit ratio
Solution: Increase TTL, review cache key patterns, check cache size limits
```

Circuit Breaker Trips

```
Issue: Circuit breaker preventing API calls
Solution: Check API health, review failure thresholds, verify network connectivity
```

Debugging

Enable debug logging:

```yaml
logging:
  level:
    dev.mars.rulesengine.core.service.data.external: DEBUG
```

Check health status:

```java
ConnectionStatus status = dataSource.getConnectionStatus();
System.out.println("Status: " + status.getState());
System.out.println("Message: " + status.getMessage());
```

Review metrics:

```java
DataSourceMetrics metrics = dataSource.getMetrics();
System.out.println("Success rate: " + metrics.getSuccessRate());
System.out.println("Avg response time: " + metrics.getAverageResponseTime());
```

Performance Tuning

1. **Database Connection Pools**:

    ```yaml
    connection:
      maxPoolSize: 20        # Adjust based on load
      minPoolSize: 5         # Keep minimum connections
      connectionTimeout: 30000
      idleTimeout: 600000
    ```

2. **API Circuit Breakers**:

```
circuitBreaker:
  failureThreshold: 5    # Number of failures before opening
  recoveryTimeout: 30000 # Time before attempting recovery
  halfOpenMaxCalls: 3    # Test calls in half-open state
```

3. **Cache Optimization**:

```
cache:
  maxSize: 10000         # Increase for better hit rates
  ttlSeconds: 600        # Balance freshness vs performance
  evictionPolicy: "LRU"  # Use appropriate eviction strategy
```

Architecture

Core Components

The external data source integration is built around several key components that work together to provide a robust and scalable data access layer:

1. **DataSourceRegistry**: Centralized registry for all data sources with health monitoring
2. **DataSourceFactory**: Creates and configures data source instances with resource caching
3. **DataSourceManager**: Coordinates multiple data sources with load balancing and failover
4. **DataSourceConfigurationService**: High-level service for configuration management

Core Interfaces

ExternalDataSource

The main interface for all data source implementations:

```
public interface ExternalDataSource extends DataSource {
    // Basic properties
    String getName();
    DataSourceType getSourceType();
    String getDataType();
    DataSourceConfiguration getConfiguration();

    // Lifecycle management
    void initialize(DataSourceConfiguration configuration) throws DataSourceException;
    void shutdown() throws DataSourceException;
    void refresh() throws DataSourceException;

    // Data operations
    Object getData(String queryName, Object... parameters) throws DataSourceException;
    List<Object> query(String query, Map<String, Object> parameters) throws DataSourceException;
    Object queryForObject(String query, Map<String, Object> parameters) throws DataSourceException;
    void batchUpdate(List<String> updates) throws DataSourceException;

    // Health and monitoring
    boolean isHealthy();
    boolean testConnection();
    ConnectionStatus getConnectionStatus();
    DataSourceMetrics getMetrics();

    // Capabilities
    boolean supportsDataType(String dataType);
    String[] getParameterNames();
}
```

DataSourceType Enumeration

Supported data source types:

```
public enum DataSourceType {
    DATABASE("database", "Database", "Relational database systems"),
    REST_API("rest-api", "REST API", "HTTP REST API endpoints"),
    MESSAGE_QUEUE("message-queue", "Message Queue", "Message queue systems"),
    FILE_SYSTEM("file-system", "File System", "File-based data sources"),
    CACHE("cache", "Cache", "In-memory cache systems"),
    CUSTOM("custom", "Custom", "Custom data source implementations");
}
```

Implementation Classes

- **DatabaseDataSource** - Database connectivity with connection pooling (PostgreSQL, MySQL, Oracle, SQL Server, H2)
- **RestApiDataSource** - REST API integration with circuit breakers and authentication
- **FileSystemDataSource** - File processing with format-specific readers (CSV, JSON, XML, Fixed-width)
- **CacheDataSource** - In-memory caching with TTL and eviction policies

Data Flow

```
YAML Config → ConfigurationService → Manager → Registry → DataSource → External System
```

Thread Safety

All components are designed for concurrent access:

- Thread-safe data structures (ConcurrentHashMap, etc.)
- Proper synchronization for shared resources
- Connection pooling for database sources
- Immutable configuration objects

Configuration

Environment Variables

Use environment variables for sensitive data:

```
connection:
  username: "app_user"
  password: "${DB_PASSWORD}"  # Resolved from environment
  apiKey: "${API_KEY}"
```

Environment-Specific Overrides

```yaml
environments:
  development:
    dataSources:
      - name: "user-database"
        connection:
          host: "localhost"
        cache:
          ttlSeconds: 60

  production:
    dataSources:
      - name: "user-database"
        connection:
          host: "prod-db.example.com"
          maxPoolSize: 50
        cache:
          ttlSeconds: 600
          maxSize: 5000
```

Configuration Classes

DataSourceConfiguration

Main configuration class containing all settings:

```java
public class DataSourceConfiguration {
    private String name;
    private String type;
    private String sourceType;
    private String description;
    private boolean enabled = true;
    private String implementation;

    private ConnectionConfig connection;
    private CacheConfig cache;
    private HealthCheckConfig healthCheck;
    private AuthenticationConfig authentication;

    // Type-specific configurations
    private Map<String, String> queries;
    private Map<String, String> endpoints;
    private Map<String, String> topics;
    private Map<String, String> keyPatterns;
    private FileFormatConfig fileFormat;
    private CircuitBreakerConfig circuitBreaker;
    private ResponseMappingConfig responseMapping;

    // Custom properties for extensibility
    private Map<String, Object> customProperties;
}
```

ConnectionConfig

Connection-specific settings for different data source types:

```
public class ConnectionConfig {
    // Database connection properties
    private String host;
    private Integer port;
    private String database;
    private String schema;
    private String username;
    private String password;
    private boolean sslEnabled = false;

    // HTTP/REST API connection properties
    private String baseUrl;
    private Integer timeout = 30000;
    private Integer retryAttempts = 3;
    private Integer retryDelay = 1000;
    private Map<String, String> headers;

    // Connection pooling configuration
    private ConnectionPoolConfig connectionPool;

    // Custom connection properties
    private Map<String, Object> customProperties;
}
```

CacheConfig

Caching configuration with multiple eviction policies:

```
public class CacheConfig {
    public enum EvictionPolicy {
        LRU, LFU, FIFO, TTL_BASED, RANDOM
    }

    private Boolean enabled = true;
    private Long ttlSeconds = 3600L;
    private Long maxIdleSeconds = 1800L;
    private Integer maxSize = 10000;
    private EvictionPolicy evictionPolicy = EvictionPolicy.LRU;
    private Boolean preloadEnabled = false;
    private Boolean refreshAhead = false;
    private Long refreshAheadFactor = 75L;
    private Boolean statisticsEnabled = true;
    private String keyPrefix;
    private Boolean compressionEnabled = false;
    private String serializationFormat = "json";
}
```

HealthCheckConfig

Health monitoring configuration:

```
public class HealthCheckConfig {
    private Boolean enabled = true;
    private Long intervalSeconds = 60L;
    private Long timeoutSeconds = 10L;
    private Integer retryAttempts = 3;
    private Long retryDelay = 1000L;
    private String query;
    private String endpoint;
```

```
        private String expectedResponse;
        private Integer failureThreshold = 3;
        private Integer successThreshold = 1;
        private Boolean logFailures = true;
        private Boolean alertOnFailure = false;
        private String alertEndpoint;

        // Circuit breaker integration
        private Boolean circuitBreakerIntegration = false;
        private Integer circuitBreakerFailureThreshold = 5;
        private Long circuitBreakerTimeoutSeconds = 60L;
    }
```

AuthenticationConfig

Authentication configuration supporting multiple methods:

```
    public class AuthenticationConfig {
        public enum AuthenticationType {
            NONE, BASIC, BEARER_TOKEN, API_KEY, OAUTH2, CERTIFICATE, CUSTOM
        }

        private String type = "none";
        private String username;
        private String password;
        private String token;
        private String apiKey;
        private String apiKeyHeader = "X-API-Key";
        private String tokenHeader = "Authorization";
        private String tokenPrefix = "Bearer ";

        // OAuth2 configuration
        private String clientId;
        private String clientSecret;
        private String tokenUrl;
        private String scope;
        private String grantType = "client_credentials";

        // Certificate configuration
        private String certificatePath;
        private String certificatePassword;
        private String keyStorePath;
        private String keyStorePassword;

        // Token refresh configuration
        private Boolean autoRefresh = true;
        private Long refreshThresholdSeconds = 300L;
        private Integer maxRefreshAttempts = 3;
    }
```

Database Configuration

Supported Databases

APEX provides robust integration with all major relational databases. Each database type has its own specific configuration options and best practices, but they all share common features like connection pooling, query management, caching, health monitoring, and security.

What you get with database integration:

- **Connection pooling**: Efficiently manage database connections for high performance
- **Query management**: Define named queries in YAML for better maintainability
- **Automatic caching**: Cache query results to reduce database load
- **Health monitoring**: Continuously monitor database connectivity and performance
- **Security**: SSL/TLS encryption, credential management, and access control
- **Transaction support**: Handle database transactions properly
- **Prepared statements**: Automatic SQL injection protection

PostgreSQL - The Popular Open Source Choice

PostgreSQL is a powerful, open-source relational database that's popular for its reliability, performance, and rich feature set. It's an excellent choice for most applications.

When to choose PostgreSQL:

- You need a reliable, ACID-compliant database
- You want advanced features like JSON support, full-text search, and custom data types
- You're building a new application and want a modern database
- You need good performance with complex queries
- You want strong community support and extensive documentation

```yaml
dataSources:
  - name: "postgres-db"
    type: "database"
    sourceType: "postgresql"
    connection:
      host: "localhost"                    # Database server hostname
      port: 5432                          # PostgreSQL default port
      database: "myapp"                   # Database name
      username: "app_user"                # Database username
      password: "${DB_PASSWORD}"          # Password from environment variable
      schema: "public"                    # Database schema (usually "public")
      sslEnabled: true                    # Enable SSL encryption
      sslMode: "require"                  # Force SSL (use "prefer" for dev)

      # Optional PostgreSQL-specific settings
      applicationName: "APEX-Rules-Engine" # Shows up in PostgreSQL logs
      connectTimeout: 10000               # 10 seconds to establish connection
      socketTimeout: 30000                # 30 seconds for query timeout

      # Connection pool settings for optimal performance
      maxPoolSize: 20                     # Maximum connections in pool
      minPoolSize: 5                      # Minimum connections to maintain
      connectionTimeout: 30000            # Wait 30s for connection from pool
      idleTimeout: 600000                 # Close idle connections after 10 minutes
      maxLifetime: 1800000                # Recreate connections after 30 minutes

    queries:
      # PostgreSQL uses $1, $2, etc. for parameters (not :name syntax)
      getUserById: "SELECT * FROM users WHERE id = $1"
      getUsersByStatus: "SELECT * FROM users WHERE status = $1 ORDER BY created_at DESC"
      createUser: "INSERT INTO users (username, email, status) VALUES ($1, $2, $3) RETURNING id"

      # Complex query example
      getUserWithProfile: |
        SELECT u.id, u.username, u.email, u.status, u.created_at,
               p.first_name, p.last_name, p.phone
        FROM users u
        LEFT JOIN profiles p ON u.id = p.user_id
        WHERE u.id = $1 AND u.status = $2

    parameterNames:
      - "id"
      - "status"
      - "username"
      - "email"
      - "first_name"
      - "last_name"
```

PostgreSQL-specific tips:

- **Parameter syntax**: Use `$1, $2, $3` instead of `:name` syntax
- **SSL modes**: Use `require` for production, `prefer` for development
- **RETURNING clause**: Great for getting generated IDs after INSERT operations
- **Application name**: Helps identify your application in PostgreSQL logs and monitoring
- **JSON support**: PostgreSQL has excellent JSON and JSONB support for semi-structured data

MySQL - The World's Most Popular Database

MySQL is the world's most popular open-source database, known for its speed, reliability, and ease of use. It's particularly popular for web applications and is part of the classic LAMP stack.

When to choose MySQL:

- You're building web applications or content management systems
- You need fast read performance and simple queries
- You want a database with a huge community and lots of hosting options
- You're working with existing MySQL infrastructure
- You need a lightweight database with good performance

```
dataSources:
  - name: "mysql-db"
    type: "database"
    sourceType: "mysql"
    connection:
      host: "localhost"                    # Database server hostname
      port: 3306                          # MySQL default port
      database: "myapp"                   # Database name
      username: "app_user"                # Database username
      password: "${DB_PASSWORD}"          # Password from environment variable
      useSSL: true                        # Enable SSL encryption
      serverTimezone: "UTC"               # Set timezone (important for date/time handling)
      characterEncoding: "utf8mb4"        # Full UTF-8 support (including emojis)

      # MySQL-specific performance settings
      cachePrepStmts: true                # Cache prepared statements
      prepStmtCacheSize: 250              # Number of statements to cache
      prepStmtCacheSqlLimit: 2048         # Max SQL length to cache
      useServerPrepStmts: true            # Use server-side prepared statements

      # Connection pool settings
      maxPoolSize: 25                     # Maximum connections in pool
      minPoolSize: 5                      # Minimum connections to maintain
      connectionTimeout: 20000            # Wait 20s for connection from pool
      idleTimeout: 600000                 # Close idle connections after 10 minutes
      maxLifetime: 1800000                # Recreate connections after 30 minutes
    queries:
      # MySQL uses ? for parameters (not $1 or :name)
      getUserById: "SELECT * FROM users WHERE id = ?"
      getUsersByStatus: "SELECT * FROM users WHERE status = ? ORDER BY created_at DESC"
      createUser: "INSERT INTO users (username, email, status) VALUES (?, ?, ?)"

      # Get the last inserted ID (MySQL-specific)
      getLastInsertId: "SELECT LAST_INSERT_ID()"

      # Complex query with joins
      getUserWithProfile: |
        SELECT u.id, u.username, u.email, u.status, u.created_at,
               p.first_name, p.last_name, p.phone
        FROM users u
        LEFT JOIN profiles p ON u.id = p.user_id
        WHERE u.id = ? AND u.status = ?

    parameterNames:
      - "id"
      - "status"
      - "username"
      - "email"
      - "first_name"
      - "last_name"
```

MySQL-specific tips:

- **Parameter syntax**: Use `?` for parameters (JDBC standard)
- **Character encoding**: Always use `utf8mb4` for full UTF-8 support (including emojis)
- **Timezone**: Set `serverTimezone` to avoid timezone-related issues
- **Prepared statements**: Enable caching for better performance
- **Getting IDs**: Use `LAST_INSERT_ID()` to get auto-generated IDs after INSERT

Oracle

```
dataSources:
  - name: "oracle-db"
    type: "database"
    sourceType: "oracle"
    connection:
      host: "localhost"
      port: 1521
      serviceName: "ORCL"  # Use serviceName instead of database
      username: "app_user"
      password: "${DB_PASSWORD}"
      schema: "APP_SCHEMA"
```

SQL Server

```
dataSources:
  - name: "sqlserver-db"
    type: "database"
    sourceType: "sqlserver"
    connection:
      host: "localhost"
      port: 1433
      database: "myapp"
      username: "app_user"
```

```yaml
      password: "${DB_PASSWORD}"
      integratedSecurity: false
      encrypt: true
      trustServerCertificate: false
```

H2 Database

```yaml
dataSources:
  - name: "h2-db"
    type: "database"
    sourceType: "h2"
    connection:
      url: "jdbc:h2:mem:testdb;DB_CLOSE_DELAY=-1"
      username: "sa"
      password: ""
      mode: "PostgreSQL"  # Compatibility mode
      initScript: "classpath:schema.sql"
```

Connection Configuration

Connection Parameters

Parameter	Description	Default	Required
`host`	Database server hostname	localhost	Yes
`port`	Database server port	DB-specific	No
`database`	Database name	-	Yes*
`serviceName`	Oracle service name	-	Oracle only
`username`	Database username	-	Yes
`password`	Database password	-	Yes
`schema`	Default schema	-	No
`url`	Complete JDBC URL	Generated	No

*For H2, you can use `url` instead of individual parameters.

SSL Configuration

```yaml
connection:
  sslEnabled: true
  sslMode: "require"           # PostgreSQL: disable, allow, prefer, require, verify-ca, verify-full
  sslCert: "/path/to/cert.pem" # Client certificate
  sslKey: "/path/to/key.pem"   # Client private key
  sslRootCert: "/path/to/ca.pem" # CA certificate
  sslPassword: "${SSL_PASSWORD}" # SSL key password
```

Connection Pooling

HikariCP Configuration (Recommended)

```
connection:
  # Pool sizing
  maxPoolSize: 20              # Maximum pool size
  minPoolSize: 5               # Minimum idle connections

  # Timeouts (milliseconds)
  connectionTimeout: 30000     # Max wait for connection
  idleTimeout: 600000          # Max idle time (10 minutes)
  maxLifetime: 1800000         # Max connection lifetime (30 minutes)

  # Validation
  validationTimeout: 5000      # Connection validation timeout
  leakDetectionThreshold: 60000 # Connection leak detection (1 minute)

  # Performance
  cachePrepStmts: true         # Enable prepared statement caching
  prepStmtCacheSize: 250       # Prepared statement cache size
  prepStmtCacheSqlLimit: 2048  # Max SQL length for caching
```

Query Configuration

Named Queries

```
queries:
  # Simple select
  getUserById: "SELECT * FROM users WHERE id = :id"

  # Complex query with joins
  getUserWithProfile: |
    SELECT u.id, u.username, u.email, p.first_name, p.last_name
    FROM users u
    LEFT JOIN profiles p ON u.id = p.user_id
    WHERE u.id = :id

  # Insert query
  createUser: |
    INSERT INTO users (username, email, status, created_at)
    VALUES (:username, :email, :status, NOW())
    RETURNING id

  # Update query
  updateUserEmail: |
    UPDATE users
    SET email = :email, updated_at = NOW()
    WHERE id = :id

  # Delete query
  deleteUser: "DELETE FROM users WHERE id = :id"

  # Batch query
  getUsersByIds: "SELECT * FROM users WHERE id IN (:ids)"

  # Health check query (required)
  default: "SELECT 1"
```

Parameter Binding

```
parameterNames:
  - "id"
  - "username"
  - "email"
  - "status"
  - "ids"        # For IN clauses
  - "startDate"  # For date ranges
  - "endDate"
  - "limit"      # For pagination
  - "offset"
```

Database-Specific Settings

PostgreSQL

```yaml
connection:
  applicationName: "MyApp"
  connectTimeout: 10
  socketTimeout: 0
  tcpKeepAlive: true
  logUnclosedConnections: true
```

MySQL

```yaml
connection:
  useSSL: true
  serverTimezone: "UTC"
  characterEncoding: "utf8mb4"
  useUnicode: true
  autoReconnect: true
  maxReconnects: 3
  initialTimeout: 2
```

Oracle

```yaml
connection:
  serviceName: "ORCL"
  connectionProperties:
    oracle.jdbc.ReadTimeout: "30000"
    oracle.net.CONNECT_TIMEOUT: "10000"
    oracle.jdbc.implicitStatementCacheSize: "25"
```

API Reference

Configuration Service

DataSourceConfigurationService

High-level service for managing data source configurations:

```java
public class DataSourceConfigurationService {
    // Singleton access
    public static DataSourceConfigurationService getInstance();
    // Lifecycle management
    public void initialize(YamlRuleConfiguration yamlConfig) throws DataSourceException;
    public void shutdown();
    public boolean isInitialized();
    public boolean isRunning();

    // Configuration management
    public void reloadFromYaml(YamlRuleConfiguration yamlConfig) throws DataSourceException;
    public DataSourceConfiguration getConfiguration(String name);
    public Set<String> getConfigurationNames();

    // Data source access
    public ExternalDataSource getDataSource(String name);
    public DataSourceManager getDataSourceManager();

    // Event handling
    public void addListener(DataSourceConfigurationListener listener);
    public void removeListener(DataSourceConfigurationListener listener);
}
```

Usage Example

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

// Access data sources
ExternalDataSource userDb = service.getDataSource("user-database");
ExternalDataSource apiSource = service.getDataSource("external-api");

// Get configuration details
DataSourceConfiguration config = service.getConfiguration("user-database");
System.out.println("Data source type: " + config.getDataSourceType());

// Reload configuration
YamlRuleConfiguration newConfig = loadConfiguration("updated-config.yaml");
service.reloadFromYaml(newConfig);
```

Data Source Manager

DataSourceManager

Coordinates multiple data sources with load balancing and failover:

```
public class DataSourceManager {
    // Constructors
    public DataSourceManager();
    public DataSourceManager(DataSourceRegistry registry, DataSourceFactory factory);

    // Lifecycle management
    public void initialize(List<DataSourceConfiguration> configurations) throws DataSourceException;
    public void shutdown();
    public boolean isInitialized();
    public boolean isRunning();

    // Data source management
    public void addDataSource(DataSourceConfiguration configuration) throws DataSourceException;
    public boolean removeDataSource(String name);
    public ExternalDataSource getDataSource(String name);
    public Set<String> getDataSourceNames();

    // Type-based access
    public List<ExternalDataSource> getDataSourcesByType(DataSourceType type);
    public ExternalDataSource getDataSourceWithLoadBalancing(DataSourceType type);
    public List<ExternalDataSource> getHealthyDataSourcesByType(DataSourceType type);

    // Health monitoring
    public List<ExternalDataSource> getHealthyDataSources();
    public List<ExternalDataSource> getUnhealthyDataSources();
    public void refreshAll();

    // Advanced operations
    public List<Object> queryWithFailover(DataSourceType type, String query, Map<String, Object> parameters) throws DataS
    public CompletableFuture<List<Object>> queryAsync(String dataSourceName, String query, Map<String, Object> parameters

    // Statistics and monitoring
    public DataSourceManagerStatistics getStatistics();

    // Event handling
    public void addListener(DataSourceManagerListener listener);
    public void removeListener(DataSourceManagerListener listener);
}
```

Data Source Registry

DataSourceRegistry

Centralized registry for all data sources:

```
public class DataSourceRegistry {
    // Singleton access
    public static DataSourceRegistry getInstance();

    // Registration management
    public void register(ExternalDataSource dataSource) throws DataSourceException;
    public boolean unregister(String name);
    public boolean isRegistered(String name);
    public int size();

    // Data source access
    public ExternalDataSource getDataSource(String name);
    public Set<String> getDataSourceNames();

    // Type-based queries
    public List<ExternalDataSource> getDataSourcesByType(DataSourceType type);
    public List<ExternalDataSource> getDataSourcesByTag(String tag);

    // Health monitoring
    public List<ExternalDataSource> getHealthyDataSources();
    public List<ExternalDataSource> getUnhealthyDataSources();
    public void refreshAll();

    // Statistics
    public RegistryStatistics getStatistics();

    // Event handling
    public void addListener(DataSourceRegistryListener listener);
    public void removeListener(DataSourceRegistryListener listener);

    // Lifecycle
    public void shutdown();
}
```

Data Source Factory

DataSourceFactory

Creates and configures data source instances:

```
public class DataSourceFactory {
    // Singleton access
    public static DataSourceFactory getInstance();

    // Data source creation
    public ExternalDataSource createDataSource(DataSourceConfiguration configuration) throws DataSourceException;
    public Map<String, ExternalDataSource> createDataSources(List<DataSourceConfiguration> configurations) throws DataSou

    // Custom provider management
    public void registerProvider(String type, DataSourceProvider provider);
    public void unregisterProvider(String type);

    // Type support queries
    public boolean isTypeSupported(DataSourceType type);
    public boolean isCustomTypeSupported(String type);
    public Set<String> getSupportedTypes();

    // Resource management
    public void clearCache();
    public void shutdown();
}
```

Data Source Implementations

DatabaseDataSource

Database-specific implementation:

```
public class DatabaseDataSource implements ExternalDataSource {
    // Constructor
    public DatabaseDataSource(DataSource dataSource, DataSourceConfiguration configuration);

    // Database-specific methods
    public DataSource getDataSource();
    public JdbcTemplate getJdbcTemplate();

    // Query execution
    public List<Map<String, Object>> queryForList(String sql, Map<String, Object> parameters);
    public Map<String, Object> queryForMap(String sql, Map<String, Object> parameters);
    public <T> T queryForObject(String sql, Map<String, Object> parameters, Class<T> requiredType);

    // Batch operations
    public int[] batchUpdate(String sql, List<Map<String, Object>> batchParameters);
}
```

RestApiDataSource

REST API-specific implementation:

```
public class RestApiDataSource implements ExternalDataSource {
    // Constructor
    public RestApiDataSource(HttpClient httpClient, DataSourceConfiguration configuration);

    // HTTP-specific methods
    public HttpClient getHttpClient();
    public HttpResponse<String> executeRequest(HttpRequest request) throws IOException, InterruptedException;

    // Request building
    public HttpRequest buildGetRequest(String endpoint, Map<String, Object> parameters);
    public HttpRequest buildPostRequest(String endpoint, Object body);
    public HttpRequest buildPutRequest(String endpoint, Object body);
    public HttpRequest buildDeleteRequest(String endpoint);
}
```

FileSystemDataSource

File system-specific implementation:

```
public class FileSystemDataSource implements ExternalDataSource {
    // Constructor
    public FileSystemDataSource(DataSourceConfiguration configuration);

    // File operations
    public List<Path> listFiles(String pattern);
    public Object readFile(Path filePath);
    public Object readFile(String filename);

    // Format-specific readers
    public List<Map<String, Object>> readCsvFile(Path filePath);
    public Object readJsonFile(Path filePath);
    public List<Map<String, Object>> readXmlFile(Path filePath);
    public List<Map<String, Object>> readFixedWidthFile(Path filePath);
}
```

CacheDataSource

Cache-specific implementation:

```
public class CacheDataSource implements ExternalDataSource {
    // Constructor
    public CacheDataSource(DataSourceConfiguration configuration);

    // Cache operations
    public void put(String key, Object value);
    public void put(String key, Object value, long ttlSeconds);
    public Object get(String key);
    public boolean containsKey(String key);
    public void remove(String key);
    public void clear();

    // Pattern operations
    public Set<String> getKeys(String pattern);
    public Map<String, Object> getAll(Set<String> keys);

    // Statistics
    public long size();
    public CacheStatistics getCacheStatistics();
}
```

Exception Handling

DataSourceException

Main exception class for data source operations:

```java
public class DataSourceException extends Exception {
    public enum ErrorType {
        CONNECTION_ERROR("Connection failed"),
        CONFIGURATION_ERROR("Configuration error"),
        EXECUTION_ERROR("Execution failed"),
        DATA_FORMAT_ERROR("Data format error"),
        TIMEOUT_ERROR("Operation timed out"),
        AUTHENTICATION_ERROR("Authentication failed"),
        VALIDATION_ERROR("Validation failed"),
        RESOURCE_ERROR("Resource error"),
        CIRCUIT_BREAKER_OPEN("Circuit breaker is open"),
        HEALTH_CHECK_FAILED("Health check failed");

        private final String defaultMessage;

        ErrorType(String defaultMessage) {
            this.defaultMessage = defaultMessage;
        }

        public String getDefaultMessage() {
            return defaultMessage;
        }
    }

    // Constructors
    public DataSourceException(ErrorType errorType, String message);
    public DataSourceException(ErrorType errorType, String message, Throwable cause);

    // Properties
    public ErrorType getErrorType();
    public String getDataSourceName();
    public long getTimestamp();
}
```

Exception Handling Example

```java
try {
    ExternalDataSource dataSource = factory.createDataSource(config);
    List<Object> results = dataSource.query("getUserById", parameters);
} catch (DataSourceException e) {
    switch (e.getErrorType()) {
        case CONNECTION_ERROR:
            logger.error("Connection failed for data source: " + e.getDataSourceName(), e);
            // Implement retry logic
            break;
        case AUTHENTICATION_ERROR:
            logger.error("Authentication failed: " + e.getMessage(), e);
            // Check credentials
            break;
        case EXECUTION_ERROR:
            logger.error("Query execution failed: " + e.getMessage(), e);
            // Check query syntax
            break;
        default:
            logger.error("Unexpected error: " + e.getMessage(), e);
    }
}
```

Metrics and Monitoring

DataSourceMetrics

Metrics collection for data sources:

```
public class DataSourceMetrics {
    // Request metrics
    public long getSuccessfulRequests();
    public long getFailedRequests();
    public long getTotalRequests();
    public double getSuccessRate();

    // Timing metrics
    public double getAverageResponseTime();
    public long getMinResponseTime();
    public long getMaxResponseTime();

    // Cache metrics
    public long getCacheHits();
    public long getCacheMisses();
    public double getCacheHitRatio();

    // Connection metrics
    public int getActiveConnections();
    public int getIdleConnections();
    public int getTotalConnections();

    // Error metrics
    public Map<String, Long> getErrorCounts();
    public long getTimeoutCount();

    // Data volume metrics
    public long getBytesRead();
    public long getBytesWritten();
    public long getRecordsProcessed();

    // Lifecycle
    public LocalDateTime getCreatedAt();
    public LocalDateTime getLastResetTime();
    public void reset();
}
```

ConnectionStatus

Health and connection status information:

```
public class ConnectionStatus {
    public enum State {
        NOT_INITIALIZED("Not initialized"),
        CONNECTING("Connecting"),
        CONNECTED("Connected"),
        DISCONNECTED("Disconnected"),
        ERROR("Error"),
        SHUTDOWN("Shutdown");

        private final String description;

        State(String description) {
            this.description = description;
        }

        public String getDescription() {
            return description;
        }
    }

    // Static factory methods
    public static ConnectionStatus notInitialized();
    public static ConnectionStatus connecting();
    public static ConnectionStatus connected(String message);
    public static ConnectionStatus disconnected(String message);
    public static ConnectionStatus error(String message, Throwable error);
    public static ConnectionStatus shutdown();

    // Properties
    public State getState();
    public LocalDateTime getLastUpdated();
    public LocalDateTime getLastConnected();
    public String getMessage();
    public Throwable getError();
    public long getConnectionAttempts();
    public long getSuccessfulConnections();
}
```

RegistryStatistics

Statistics for the data source registry:

```
public class RegistryStatistics {
    // Basic counts
    public int getTotalDataSources();
    public int getHealthyDataSources();
    public int getUnhealthyDataSources();
    public double getHealthPercentage();

    // Type distribution
    public Map<DataSourceType, Integer> getCountByType();
    public Map<String, Integer> getCountByTag();

    // Health status
    public boolean isAllHealthy();
    public List<String> getUnhealthyDataSourceNames();

    // Summary
    public String getSummary();
    public LocalDateTime getLastUpdated();
}
```

Monitoring Example

```
// Collect metrics
DataSourceMetrics metrics = dataSource.getMetrics();
RegistryStatistics registryStats = registry.getStatistics();
```

```
// Log performance metrics
logger.info("Data source performance:");
logger.info("  Success rate: {}%", metrics.getSuccessRate() * 100);
logger.info("  Average response time: {}ms", metrics.getAverageResponseTime());
logger.info("  Cache hit ratio: {}%", metrics.getCacheHitRatio() * 100);

// Monitor registry health
logger.info("Registry health: {}% ({}/{} healthy)",
    registryStats.getHealthPercentage(),
    registryStats.getHealthyDataSources(),
    registryStats.getTotalDataSources());

// Alert on issues
if (registryStats.getHealthPercentage() < 90.0) {
    alertService.sendAlert("Data source health below 90%: " + registryStats.getSummary());
}
```

Usage Examples

Database Operations

```
// Simple query
List<Object> users = dataSource.query("getAllUsers", Collections.emptyMap());

// Parameterized query
Map<String, Object> params = Map.of("id", 123, "status", "active");
Object user = dataSource.queryForObject("getUserById", params);

// Batch operations
List<String> updates = List.of(
    "UPDATE users SET last_login = NOW() WHERE id = 1",
    "UPDATE users SET last_login = NOW() WHERE id = 2"
);
dataSource.batchUpdate(updates);

// Query usage in Java with complex parameters
Map<String, Object> complexParams = Map.of(
    "username", "john_doe",
    "email", "john@example.com",
    "status", "ACTIVE"
);
List<Object> results = dataSource.query("createUser", complexParams);

// Array parameters for IN clauses
Map<String, Object> arrayParams = Map.of("ids", List.of(1, 2, 3, 4, 5));
List<Object> users = dataSource.query("getUsersByIds", arrayParams);
```

REST API Operations

```
// GET request
Map<String, Object> params = Map.of("userId", 123);
Object userProfile = apiSource.queryForObject("getUserProfile", params);

// POST request (configured in YAML)
Map<String, Object> newUser = Map.of("name", "John", "email", "john@example.com");
Object result = apiSource.query("createUser", newUser);
```

```
// Custom headers and authentication
Map<String, Object> requestParams = Map.of(
    "endpoint", "users/123",
    "headers", Map.of("Accept", "application/json"),
    "timeout", 30000
);
Object response = apiSource.queryForObject("customRequest", requestParams);
```

File System Operations

```
// Read CSV file
Object csvData = fileSource.getData("csv", "users.csv");

// Read JSON file with parameters
Map<String, Object> params = Map.of("filename", "config.json");
Object config = fileSource.queryForObject("getConfig", params);

// Read XML file with XPath
Map<String, Object> xmlParams = Map.of(
    "filename", "data.xml",
    "xpath", "//user[@status='active']"
);
List<Object> activeUsers = fileSource.query("getActiveUsers", xmlParams);

// Watch for file changes
fileSource.getData("watch", "*.json");
```

Cache Operations

```
// Store in cache
Map<String, Object> params = Map.of("key", "user:123", "value", userData);
cacheSource.query("put", params);

// Retrieve from cache
Map<String, Object> getParams = Map.of("key", "user:123");
Object cachedData = cacheSource.queryForObject("get", getParams);

// Pattern-based operations
Map<String, Object> patternParams = Map.of("pattern", "user:*");
List<Object> userKeys = cacheSource.query("keys", patternParams);

// Batch cache operations
Map<String, Object> batchParams = Map.of(
    "keys", List.of("user:1", "user:2", "user:3")
);
Map<String, Object> batchResults = (Map<String, Object>) cacheSource.queryForObject("getAll", batchParams);
```

Complete Integration Example

```
public class DataSourceIntegrationExample {

    public void demonstrateIntegration() throws DataSourceException {
        // Initialize configuration service
        DataSourceConfigurationService configService = DataSourceConfigurationService.getInstance();
        YamlRuleConfiguration yamlConfig = loadYamlConfiguration("data-sources.yaml");
        configService.initialize(yamlConfig);
        // Get data sources
        ExternalDataSource userDb = configService.getDataSource("user-database");
        ExternalDataSource userCache = configService.getDataSource("user-cache");
        ExternalDataSource userApi = configService.getDataSource("user-api");

        // Implement caching strategy
        String userId = "123";
        Object userData = getUserWithCaching(userDb, userCache, userId);

        // Sync with external API
        syncUserWithExternalSystem(userApi, userData);

        // Monitor health
        monitorDataSourceHealth(configService);
    }

    private Object getUserWithCaching(ExternalDataSource db, ExternalDataSource cache, String userId)
            throws DataSourceException {

        // Try cache first
        Map<String, Object> cacheParams = Map.of("key", "user:" + userId);
        try {
            Object cachedUser = cache.queryForObject("get", cacheParams);
            if (cachedUser != null) {
                return cachedUser;
            }
        } catch (DataSourceException e) {
            // Cache miss or error, continue to database
        }

        // Fetch from database
        Map<String, Object> dbParams = Map.of("id", userId);
        Object user = db.queryForObject("getUserById", dbParams);

        // Store in cache
        if (user != null) {
            Map<String, Object> putParams = Map.of("key", "user:" + userId, "value", user);
            cache.query("put", putParams);
        }

        return user;
    }

    private void syncUserWithExternalSystem(ExternalDataSource api, Object userData)
            throws DataSourceException {

        Map<String, Object> syncParams = Map.of("userData", userData);
        try {
            api.query("syncUser", syncParams);
        } catch (DataSourceException e) {
            if (e.getErrorType() == DataSourceException.ErrorType.CIRCUIT_BREAKER_OPEN) {
                // Handle circuit breaker open state
                logger.warn("External API circuit breaker is open, skipping sync");
            } else {
                throw e;
            }
        }
    }

    private void monitorDataSourceHealth(DataSourceConfigurationService configService) {
        DataSourceManager manager = configService.getDataSourceManager();
        RegistryStatistics stats = manager.getRegistry().getStatistics();

        logger.info("Data source health summary: {}", stats.getSummary());

        if (!stats.isAllHealthy()) {
            List<String> unhealthy = stats.getUnhealthyDataSourceNames();
            logger.warn("Unhealthy data sources: {}", unhealthy);
        }
    }
}
```

Best Practices

Configuration Best Practices

1. Environment-Specific Configuration

✅ **DO**: Use environment-specific overrides

```yaml
environments:
  development:
    dataSources:
      - name: "user-database"
        connection:
          host: "localhost"
          maxPoolSize: 5
        cache:
          ttlSeconds: 60

  production:
    dataSources:
      - name: "user-database"
        connection:
          host: "prod-db.example.com"
          maxPoolSize: 50
        cache:
          ttlSeconds: 600
```

❌ **DON'T**: Hardcode environment-specific values in base configuration

2. Credential Management

✅ **DO**: Use environment variables for sensitive data

```yaml
connection:
  username: "app_user"
  password: "${DB_PASSWORD}"
  apiKey: "${API_KEY}"
```

❌ **DON'T**: Store credentials in configuration files

```yaml
# BAD - Never do this
connection:
  password: "hardcoded_password"
```

3. Configuration Validation

✅ **DO**: Validate configurations before deployment

```
public void validateConfiguration(DataSourceConfiguration config) {
    if (config.getName() == null || config.getName().trim().isEmpty()) {
        throw new IllegalArgumentException("Data source name is required");
    }

    if (config.getConnection() == null) {
        throw new IllegalArgumentException("Connection configuration is required");
    }

    // Validate connection parameters
    ConnectionConfig conn = config.getConnection();
    if (conn.getHost() == null && conn.getBaseUrl() == null) {
        throw new IllegalArgumentException("Either host or URL must be specified");
    }
}
```

4. Naming Conventions

✅ **DO**: Use consistent, descriptive names

```
dataSources:
  - name: "user-database-primary"      # Clear and specific
  - name: "customer-api-external"      # Indicates purpose and type
  - name: "config-files-local"         # Descriptive of content and location
```

❌ **DON'T**: Use generic or ambiguous names

```
dataSources:
  - name: "db1"                        # Too generic
  - name: "api"                        # Ambiguous
  - name: "files"                      # Not specific enough
```

5. Documentation and Tags

✅ **DO**: Document data sources with descriptions and tags

```
dataSources:
  - name: "user-database"
    description: "Primary user database containing authentication and profile data"
    tags:
      - "production"
      - "primary"
      - "users"
      - "authentication"
```

Health Monitoring Configuration

✅ **DO**: Configure comprehensive health checks

```
healthCheck:
  enabled: true
  intervalSeconds: 30
  timeoutSeconds: 5
  failureThreshold: 3
  recoveryThreshold: 2
  query: "SELECT 1"
```

✅ **DO**: Monitor health status programmatically

```
// Monitor connection pool usage
DataSourceMetrics metrics = dataSource.getMetrics();
int activeConnections = metrics.getActiveConnections();
int totalConnections = metrics.getTotalConnections();
double utilization = (double) activeConnections / totalConnections;

if (utilization > 0.8) {
    logger.warn("High connection pool utilization: {}%", utilization * 100);
}
```

Caching Configuration

✅ **DO**: Use appropriate TTL values

```
cache:
  # For frequently changing data
  ttlSeconds: 60              # 1 minute

  # For stable reference data
  ttlSeconds: 3600            # 1 hour

  # For configuration data
  ttlSeconds: 86400           # 24 hours
```

✅ **DO**: Size caches appropriately

```
cache:
  # Consider memory usage vs hit ratio
  maxSize: 10000              # Balance memory and performance
  evictionPolicy: "LRU"       # Use appropriate eviction strategy
```

Cache Usage Patterns

Cache keys are generated as: `{keyPrefix}:{queryName}:{parameterHash}`

Example: `myapp:getUserById:a1b2c3d4`

```
// First call - hits database, stores in cache
Object user1 = dataSource.queryForObject("getUserById", Map.of("id", 123));

// Second call - hits cache (if within TTL)
Object user2 = dataSource.queryForObject("getUserById", Map.of("id", 123));

// Different parameters - new cache entry
Object user3 = dataSource.queryForObject("getUserById", Map.of("id", 456));
```

Performance Optimization

Connection Pooling

✅ **DO**: Configure appropriate pool sizes

```
connection:
  # For high-throughput applications
  maxPoolSize: 50
  minPoolSize: 10

  # For low-latency requirements
  connectionTimeout: 5000
  idleTimeout: 300000
```

✅ **DO**: Monitor pool utilization

```
// Monitor connection pool metrics
DataSourceMetrics metrics = dataSource.getMetrics();
int activeConnections = metrics.getActiveConnections();
int totalConnections = metrics.getTotalConnections();
double utilization = (double) activeConnections / totalConnections;

if (utilization > 0.8) {
    logger.warn("High connection pool utilization: {}%", utilization * 100);
}
```

Query Optimization

✅ **DO**: Use efficient queries

```
queries:
  # Use indexes effectively
  getUserByEmail: "SELECT id, username, email FROM users WHERE email = :email"

  # Limit result sets
  getRecentUsers: "SELECT * FROM users ORDER BY created_at DESC LIMIT :limit"

  # Use specific columns
  getUserSummary: "SELECT id, username FROM users WHERE id = :id"
```

❌ **DON'T**: Use inefficient queries

```
queries:
  # Avoid SELECT *
  getAllUserData: "SELECT * FROM users"

  # Avoid unindexed searches
  findUserByName: "SELECT * FROM users WHERE LOWER(name) LIKE '%:name%'"
```

Batch Operations

✅ **DO**: Use batch operations for multiple updates

```
// Batch database updates
List<String> updates = Arrays.asList(
    "UPDATE users SET last_login = NOW() WHERE id = 1",
    "UPDATE users SET last_login = NOW() WHERE id = 2",
    "UPDATE users SET last_login = NOW() WHERE id = 3"
);
dataSource.batchUpdate(updates);
```

Connection Pool Tuning

```
connection:
  # For high-load applications
  maxPoolSize: 50
  minPoolSize: 10
  connectionTimeout: 10000

  # For low-latency requirements
  maxPoolSize: 20
  minPoolSize: 10
  idleTimeout: 300000          # 5 minutes
  maxLifetime: 900000          # 15 minutes
```

Cache Optimization

```
cache:
  # For frequently accessed data
  ttlSeconds: 600              # 10 minutes
  maxSize: 5000

  # For rarely changing data
  ttlSeconds: 3600             # 1 hour
  maxSize: 10000
```

Performance Monitoring

✅ **DO**: Collect comprehensive metrics

```
// Monitor key performance indicators
DataSourceMetrics metrics = dataSource.getMetrics();

// Response time metrics
double avgResponseTime = metrics.getAverageResponseTime();
long maxResponseTime = metrics.getMaxResponseTime();

// Success rate metrics
double successRate = metrics.getSuccessRate();
long failedRequests = metrics.getFailedRequests();

// Cache metrics
double cacheHitRatio = metrics.getCacheHitRatio();
long cacheHits = metrics.getCacheHits();

// Log or send to monitoring system
```

```
monitoringService.recordMetric("datasource.response_time.avg", avgResponseTime);
monitoringService.recordMetric("datasource.success_rate", successRate);
monitoringService.recordMetric("datasource.cache_hit_ratio", cacheHitRatio);
```

Performance Tuning Guidelines

1. **Database Connection Pools**:

   ```
   connection:
     maxPoolSize: 20        # Adjust based on load
     minPoolSize: 5         # Keep minimum connections
     connectionTimeout: 30000
     idleTimeout: 600000
   ```

2. **API Circuit Breakers**:

   ```
   circuitBreaker:
     failureThreshold: 5    # Number of failures before opening
     recoveryTimeout: 30000 # Time before attempting recovery
     halfOpenMaxCalls: 3    # Test calls in half-open state
   ```

3. **Cache Optimization**:

   ```
   cache:
     maxSize: 10000         # Increase for better hit rates
     ttlSeconds: 600        # Balance freshness vs performance
     evictionPolicy: "LRU"  # Use appropriate eviction strategy
   ```

Security Guidelines

Authentication and Authorization

✅ **DO**: Use strong authentication methods

```
authentication:
  type: "oauth2"
  clientId: "${CLIENT_ID}"
  clientSecret: "${CLIENT_SECRET}"
  tokenUrl: "https://auth.example.com/oauth/token"
  scope: "read:data"
```

✅ **DO**: Implement proper access controls

```
-- Grant minimum required permissions
GRANT SELECT, INSERT, UPDATE ON users TO app_user;
-- Don't grant unnecessary permissions like DROP, CREATE, etc.
```

Encryption and SSL

✅ **DO**: Always use SSL/TLS in production

```
connection:
  sslEnabled: true
  sslMode: "require"
  sslVerifyServerCertificate: true
```

✅ **DO**: Encrypt sensitive configuration data

```
connection:
  password: "ENC(AES256:encrypted_password_here)"
```

Network Security

✅ **DO**: Use network security controls

```
connection:
  # Use private network addresses
  host: "10.0.1.100"

  # Configure appropriate timeouts
  connectionTimeout: 10000
  readTimeout: 30000
```

Credential Management

```
connection:
  username: "app_user"
  password: "${DB_PASSWORD}"    # Environment variable

  # Or use encrypted passwords
  password: "ENC(encrypted_password_here)"
```

SSL/TLS Configuration

```
connection:
  sslEnabled: true
  sslMode: "require"

  # Certificate-based authentication
  sslCert: "/etc/ssl/certs/client-cert.pem"
  sslKey: "/etc/ssl/private/client-key.pem"
  sslRootCert: "/etc/ssl/certs/ca-cert.pem"

  # Verify server certificate
  sslVerifyServerCertificate: true
```

Database Permissions

Grant minimum required permissions:

```
-- PostgreSQL example
CREATE USER app_user WITH PASSWORD 'secure_password';
GRANT CONNECT ON DATABASE myapp TO app_user;
GRANT USAGE ON SCHEMA public TO app_user;
GRANT SELECT, INSERT, UPDATE, DELETE ON ALL TABLES IN SCHEMA public TO app_user;
GRANT USAGE, SELECT ON ALL SEQUENCES IN SCHEMA public TO app_user;
```

Security Best Practices

1. **SSL/TLS**: Always use encrypted connections in production
2. **Authentication**: Use strong authentication methods
3. **Access Control**: Limit database permissions to minimum required
4. **Audit Logging**: Log all data access for compliance
5. **Network Security**: Use private networks and appropriate firewall rules
6. **Credential Rotation**: Regularly rotate passwords and API keys
7. **Encryption**: Encrypt sensitive data at rest and in transit

Error Handling and Resilience

Circuit Breaker Pattern

✅ **DO**: Configure circuit breakers for external APIs

```
circuitBreaker:
  enabled: true
  failureThreshold: 5         # Open after 5 failures
  recoveryTimeout: 30000      # Wait 30 seconds before retry
  halfOpenMaxCalls: 3         # Test with 3 calls in half-open state
```

Retry Logic

✅ **DO**: Implement appropriate retry strategies

```
connection:
  retryAttempts: 3
  retryDelay: 1000            # 1 second initial delay
  retryBackoffMultiplier: 2.0 # Exponential backoff
```

Graceful Degradation

✅ **DO**: Handle failures gracefully

```
public Object getUserData(String userId) {
    try {
        // Try primary data source
        return primaryDataSource.queryForObject("getUserById", Map.of("id", userId));
    } catch (DataSourceException e) {
        logger.warn("Primary data source failed, trying cache", e);

        try {
            // Fallback to cache
            return cacheDataSource.get("user:" + userId);
        } catch (DataSourceException cacheError) {
            logger.error("Both primary and cache failed", cacheError);

            // Return default/empty response
            return createDefaultUserResponse(userId);
        }
    }
}
```

Failover Implementation

✅ **DO**: Implement automatic failover

```
// Manager provides failover capabilities
DataSourceManager manager = configService.getDataSourceManager();

// Query with automatic failover
Map<String, Object> params = Map.of("id", 123);
List<Object> results = manager.queryWithFailover(DataSourceType.DATABASE, "getUserById", params);
```

Circuit Breaker Configuration

```
circuitBreaker:
  enabled: true
  failureThreshold: 5         # Number of failures before opening circuit
  timeoutSeconds: 60          # Time to wait before trying half-open
  successThreshold: 3         # Number of successes needed to close circuit
  requestVolumeThreshold: 10  # Minimum requests before evaluating failure rate
  failureRateThreshold: 50.0  # Failure rate percentage to open circuit
  fallbackResponse: "Service temporarily unavailable"
  logStateChanges: true
  metricsEnabled: true
```

Error Handling Patterns

✅ **DO**: Use specific exception handling

```
try {
    Object result = dataSource.queryForObject("getUserById", params);
    return result;
} catch (DataSourceException e) {
    switch (e.getErrorType()) {
        case CONNECTION_ERROR:
            // Retry with exponential backoff
            return retryWithBackoff(() -> dataSource.queryForObject("getUserById", params));

        case CIRCUIT_BREAKER_OPEN:
            // Use cached data or default response
            return getCachedUserOrDefault(userId);

        case TIMEOUT_ERROR:
            // Log and return partial data
            logger.warn("Query timeout for user {}", userId);
            return getPartialUserData(userId);

        case AUTHENTICATION_ERROR:
            // Refresh credentials and retry
            refreshCredentials();
            return dataSource.queryForObject("getUserById", params);

        default:
            // Log error and propagate
            logger.error("Unexpected data source error", e);
            throw e;
    }
}
```

Health Check Integration

✅ **DO**: Integrate health checks with circuit breakers

```
healthCheck:
  enabled: true
  intervalSeconds: 30
  timeoutSeconds: 5
  failureThreshold: 3
  recoveryThreshold: 2

  # Circuit breaker integration
  circuitBreakerIntegration: true
  circuitBreakerFailureThreshold: 5
  circuitBreakerTimeoutSeconds: 60
```

Resilience Patterns

1. **Circuit Breaker**: Prevent cascading failures
2. **Retry with Backoff**: Handle transient failures
3. **Timeout**: Prevent hanging operations
4. **Bulkhead**: Isolate critical resources
5. **Fallback**: Provide alternative responses
6. **Health Checks**: Monitor system health

Monitoring and Observability

Metrics Collection

✅ **DO**: Collect comprehensive metrics

```
// Monitor key performance indicators
DataSourceMetrics metrics = dataSource.getMetrics();

// Response time metrics
double avgResponseTime = metrics.getAverageResponseTime();
long maxResponseTime = metrics.getMaxResponseTime();
```

```
// Success rate metrics
double successRate = metrics.getSuccessRate();
long failedRequests = metrics.getFailedRequests();

// Cache metrics
double cacheHitRatio = metrics.getCacheHitRatio();
long cacheHits = metrics.getCacheHits();

// Log or send to monitoring system
monitoringService.recordMetric("datasource.response_time.avg", avgResponseTime);
monitoringService.recordMetric("datasource.success_rate", successRate);
monitoringService.recordMetric("datasource.cache_hit_ratio", cacheHitRatio);
```

Alerting

✅ **DO**: Set up proactive alerts

```
// Alert on high error rates
if (metrics.getSuccessRate() < 0.95) {
    alertService.sendAlert(AlertLevel.WARNING,
        "Data source success rate below 95%: " + metrics.getSuccessRate());
}

// Alert on slow response times
if (metrics.getAverageResponseTime() > 1000) {
    alertService.sendAlert(AlertLevel.WARNING,
        "Data source response time above 1 second: " + metrics.getAverageResponseTime() + "ms");
}

// Alert on health check failures
if (!dataSource.isHealthy()) {
    alertService.sendAlert(AlertLevel.CRITICAL,
        "Data source health check failed: " + dataSource.getName());
}
```

Logging

✅ **DO**: Implement structured logging

```
// Use structured logging with context
logger.info("Data source query executed",
    Map.of(
        "dataSource", dataSource.getName(),
        "query", queryName,
        "parameters", parameters,
        "responseTime", responseTime,
        "resultCount", results.size()
    ));

// Log errors with full context
logger.error("Data source query failed",
    Map.of(
        "dataSource", dataSource.getName(),
        "query", queryName,
        "parameters", parameters,
        "errorType", e.getErrorType(),
        "errorMessage", e.getMessage()
    ), e);
```

Health Monitoring

✅ **DO**: Monitor data source health continuously

```
// Check health status
ConnectionStatus status = dataSource.getConnectionStatus();
System.out.println("State: " + status.getState());
System.out.println("Healthy: " + dataSource.isHealthy());
System.out.println("Last Check: " + status.getLastCheckTime());

// Monitor registry health
RegistryStatistics registryStats = registry.getStatistics();
logger.info("Registry health: {}% ({}/{} healthy)",
    registryStats.getHealthPercentage(),
    registryStats.getHealthyDataSources(),
    registryStats.getTotalDataSources());
```

Observability Best Practices

1. **Metrics Collection**: Enable comprehensive metrics
2. **Health Monitoring**: Set up alerts for health check failures
3. **Performance Tracking**: Monitor response times and throughput
4. **Capacity Planning**: Track resource usage trends
5. **Distributed Tracing**: Trace requests across data sources
6. **Log Aggregation**: Centralize logs for analysis

Testing Strategies

Unit Testing

✅ **DO**: Test data source configurations

```
@Test
public void testDatabaseConfiguration() {
    DataSourceConfiguration config = createDatabaseConfig();

    // Validate configuration
    assertNotNull(config.getName());
    assertNotNull(config.getConnection());
    assertTrue(config.isEnabled());

    // Test data source creation
    ExternalDataSource dataSource = factory.createDataSource(config);
    assertNotNull(dataSource);
    assertEquals(DataSourceType.DATABASE, dataSource.getSourceType());
}

@Test
public void testConfigurationValidation() {
    DataSourceConfiguration config = new DataSourceConfiguration();

    // Test validation failures
    assertThrows(IllegalArgumentException.class, () -> config.validate());

    // Test valid configuration
    config.setName("test-db");
    config.setType("database");
    config.setConnection(createValidConnectionConfig());

    assertDoesNotThrow(() -> config.validate());
}
```

Integration Testing

✅ **DO**: Test end-to-end workflows

```
@Test
public void testDataSourceIntegration() throws Exception {
    // Initialize manager with test configuration
    DataSourceManager manager = new DataSourceManager();
    manager.initialize(testConfigurations);

    // Test data retrieval
    ExternalDataSource dataSource = manager.getDataSource("test-database");
    Map<String, Object> params = Map.of("id", 1);
    Object result = dataSource.queryForObject("getUserById", params);

    assertNotNull(result);

    // Test health monitoring
    assertTrue(dataSource.isHealthy());

    // Test metrics collection
    DataSourceMetrics metrics = dataSource.getMetrics();
    assertTrue(metrics.getTotalRequests() > 0);
}

@Test
public void testFailoverScenario() throws Exception {
    // Setup primary and backup data sources
    DataSourceManager manager = setupManagerWithFailover();

    // Simulate primary failure
    simulatePrimaryFailure();

    // Test automatic failover
    Map<String, Object> params = Map.of("id", 123);
    List<Object> results = manager.queryWithFailover(DataSourceType.DATABASE, "getUserById", params);

    assertNotNull(results);
    assertFalse(results.isEmpty());
}
```

Performance Testing

✅ **DO**: Test under load

```
@Test
public void testDataSourcePerformance() throws Exception {
    int threadCount = 10;
    int operationsPerThread = 100;
    ExecutorService executor = Executors.newFixedThreadPool(threadCount);
    CountDownLatch latch = new CountDownLatch(threadCount);

    long startTime = System.currentTimeMillis();
    for (int i = 0; i < threadCount; i++) {
        executor.submit(() -> {
            try {
                for (int j = 0; j < operationsPerThread; j++) {
                    dataSource.queryForObject("getUserById", Map.of("id", j));
                }
            } finally {
                latch.countDown();
            }
        });
    }

    latch.await(30, TimeUnit.SECONDS);
    long endTime = System.currentTimeMillis();

    double operationsPerSecond = (double) (threadCount * operationsPerThread) /
                                ((endTime - startTime) / 1000.0);

    // Assert performance requirements
    assertTrue("Performance too low: " + operationsPerSecond + " ops/sec",
               operationsPerSecond > 100);
}
```

Mock Testing

✅ **DO**: Use mocks for external dependencies

```
@Test
public void testWithMockDataSource() {
    // Create mock data source
    ExternalDataSource mockDataSource = Mockito.mock(ExternalDataSource.class);

    // Setup mock behavior
    when(mockDataSource.queryForObject(eq("getUserById"), any()))
        .thenReturn(createMockUser());
    when(mockDataSource.isHealthy()).thenReturn(true);

    // Test business logic
    UserService userService = new UserService(mockDataSource);
    User user = userService.getUser(123);

    assertNotNull(user);
    assertEquals("test-user", user.getUsername());

    // Verify interactions
    verify(mockDataSource).queryForObject("getUserById", Map.of("id", 123));
}
```

Test Configuration

✅ **DO**: Use test-specific configurations

```
# test-data-sources.yaml
dataSources:
  - name: "test-database"
    type: "database"
    sourceType: "h2"
    connection:
      url: "jdbc:h2:mem:testdb;DB_CLOSE_DELAY=-1"
      username: "sa"
      password: ""
    queries:
      getUserById: "SELECT * FROM users WHERE id = :id"
    cache:
      enabled: false  # Disable caching for predictable tests
    healthCheck:
      enabled: false  # Disable health checks for faster tests
```

Testing Best Practices

1. **Unit Tests**: Test individual components in isolation
2. **Integration Tests**: Test complete workflows
3. **Performance Tests**: Validate performance requirements
4. **Mock External Dependencies**: Use mocks for reliable tests
5. **Test Data Management**: Use consistent test data
6. **Environment Isolation**: Use separate test environments

Deployment and Operations

Configuration Management

✅ **DO**: Use configuration management tools

```
# Use environment-specific configuration files
kubectl create configmap datasource-config --from-file=data-sources-prod.yaml

# Use secrets for sensitive data
kubectl create secret generic datasource-secrets \
  --from-literal=DB_PASSWORD=secure_password \
  --from-literal=API_KEY=secret_api_key
```

Health Checks

✅ **DO**: Implement readiness and liveness probes

```
# Kubernetes deployment example
spec:
  containers:
  - name: app
    livenessProbe:
      httpGet:
        path: /health/datasources
        port: 8080
      initialDelaySeconds: 30
      periodSeconds: 10

    readinessProbe:
      httpGet:
        path: /ready/datasources
        port: 8080
      initialDelaySeconds: 5
```

```
        periodSeconds: 5
```

Capacity Planning

✅ **DO**: Monitor resource usage

```java
// Monitor connection pool usage
int activeConnections = metrics.getActiveConnections();
int maxConnections = config.getConnection().getMaxPoolSize();
double poolUtilization = (double) activeConnections / maxConnections;

// Monitor cache usage
long cacheSize = cacheMetrics.getCurrentSize();
long maxCacheSize = config.getCache().getMaxSize();
double cacheUtilization = (double) cacheSize / maxCacheSize;

// Plan for growth
if (poolUtilization > 0.8) {
    logger.warn("Consider increasing connection pool size");
}
```

Production Configuration Example

```yaml
dataSources:
  - name: "production-database"
    type: "database"
    sourceType: "postgresql"
    enabled: true
    description: "Production PostgreSQL database"
    tags: ["production", "primary"]

    connection:
      host: "prod-db.example.com"
      port: 5432
      database: "myapp_prod"
      username: "app_user"
      password: "${PROD_DB_PASSWORD}"
      schema: "public"

      maxPoolSize: 30
      minPoolSize: 10
      connectionTimeout: 20000
      idleTimeout: 300000
      maxLifetime: 900000

      sslEnabled: true
      sslMode: "require"
      sslRootCert: "/etc/ssl/certs/ca-cert.pem"

    queries:
      getUserById: "SELECT id, username, email, status, created_at FROM users WHERE id = :id"
      getUserByEmail: "SELECT id, username, email, status FROM users WHERE email = :email"
      getActiveUsers: "SELECT id, username, email FROM users WHERE status = 'ACTIVE' ORDER BY last_login DESC LIMIT :limi
      createUser: "INSERT INTO users (username, email, status) VALUES (:username, :email, 'ACTIVE') RETURNING id"
      updateUserStatus: "UPDATE users SET status = :status, updated_at = NOW() WHERE id = :id"
      getUserStats: "SELECT COUNT(*) as total, COUNT(CASE WHEN status = 'ACTIVE' THEN 1 END) as active FROM users"
      default: "SELECT 1"

    parameterNames: ["id", "email", "username", "status", "limit"]
    cache:
      enabled: true
      ttlSeconds: 300
      maxSize: 2000
      keyPrefix: "proddb"

    healthCheck:
      enabled: true
      intervalSeconds: 30
      timeoutSeconds: 5
      failureThreshold: 2
      query: "SELECT COUNT(*) FROM users LIMIT 1"
```

Troubleshooting

Common Issues and Solutions

Connection Failures

```
Error: Failed to connect to database
Solution: Check connection parameters, network connectivity, and credentials
```

Debugging Steps:

1. Verify connection parameters (host, port, database name)
2. Test network connectivity: `telnet host port`
3. Check credentials and permissions
4. Review SSL/TLS configuration
5. Check firewall rules

Cache Misses

```
Issue: Low cache hit ratio
Solution: Increase TTL, review cache key patterns, check cache size limits
```

Debugging Steps:

1. Monitor cache metrics: `metrics.getCacheHitRatio()`
2. Review TTL settings
3. Check cache key generation logic
4. Verify cache size limits
5. Analyze access patterns

Circuit Breaker Trips

```
Issue: Circuit breaker preventing API calls
Solution: Check API health, review failure thresholds, verify network connectivity
```

Debugging Steps:

1. Check circuit breaker state
2. Review failure threshold settings
3. Test API endpoint manually
4. Check network connectivity
5. Review error logs

High CPU Usage

```
Issue: High CPU usage from data source operations
Solution: Reduce polling frequency, optimize queries, check connection pool settings
```

Debugging Steps:

1. Profile application CPU usage
2. Review health check intervals
3. Optimize database queries
4. Check connection pool configuration
5. Monitor thread usage

Memory Issues

```
Issue: OutOfMemoryError related to data sources
Solution: Reduce cache sizes, check for connection leaks, optimize data structures
```

Debugging Steps:

1. Monitor heap usage
2. Check for connection leaks
3. Review cache configurations
4. Analyze memory dumps
5. Optimize data structures

Debugging Tools

✅ **DO**: Use comprehensive logging

```yaml
logging:
  level:
    dev.mars.rulesengine.core.service.data.external: DEBUG
    com.zaxxer.hikari: DEBUG
```

✅ **DO**: Enable JMX monitoring

```java
// Enable JMX for connection pools and caches
System.setProperty("com.zaxxer.hikari.housekeeping.periodMs", "30000");
```

✅ **DO**: Monitor health status

```
ConnectionStatus status = dataSource.getConnectionStatus();
System.out.println("Status: " + status.getState());
System.out.println("Message: " + status.getMessage());
```

✅ **DO**: Review metrics regularly

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

Performance Tuning

1. **Database Connection Pools**:

```
connection:
  maxPoolSize: 20        # Adjust based on load
  minPoolSize: 5         # Keep minimum connections
  connectionTimeout: 30000
  idleTimeout: 600000
```

2. **API Circuit Breakers**:

```
circuitBreaker:
  failureThreshold: 5    # Number of failures before opening
  recoveryTimeout: 30000 # Time before attempting recovery
  halfOpenMaxCalls: 3    # Test calls in half-open state
```

3. **Cache Optimization**:

```
cache:
  maxSize: 10000         # Increase for better hit rates
  ttlSeconds: 600        # Balance freshness vs performance
  evictionPolicy: "LRU"  # Use appropriate eviction strategy
```

Operational Checklist

- □ Configuration validated and tested
- □ Environment variables properly set
- □ SSL certificates installed and valid
- □ Database permissions configured
- □ Health checks enabled and working
- □ Monitoring and alerting configured
- □ Performance baselines established
- □ Backup and recovery procedures tested
- □ Documentation updated
- □ Team trained on operations

This comprehensive guide covers all aspects of external data source integration in the SpEL Rules Engine, providing developers and operators with the knowledge needed to successfully implement, configure, and maintain robust data integration solutions.