HACKFEST PROJECT SUBMISSION : CUSTOM RULE ENGINE

By Hitesh Chawla | +91 8708063057 | www.hiteshlabs.com

This Design document provides an overview of a **Custom Rule Engine** Architecture using NodeJS, Kafka and Redis. The architecture is designed to handle high-throughput data streams, manage backpressure, cache frequently accessed data, and evaluate data against dynamic rules. Best suited for Real Time Complex Rule Evaluation in High Transaction Data Streams.

Current Stats: 1000+ Objects / ~35 ms (Avg < 50 ms) *(Tested on I9 13th Gen, AWS Graviton)*

Please Refer to Attached Files with this Documentation (InputData.json, InputRules.json, Output.json)

SOME FEATURES

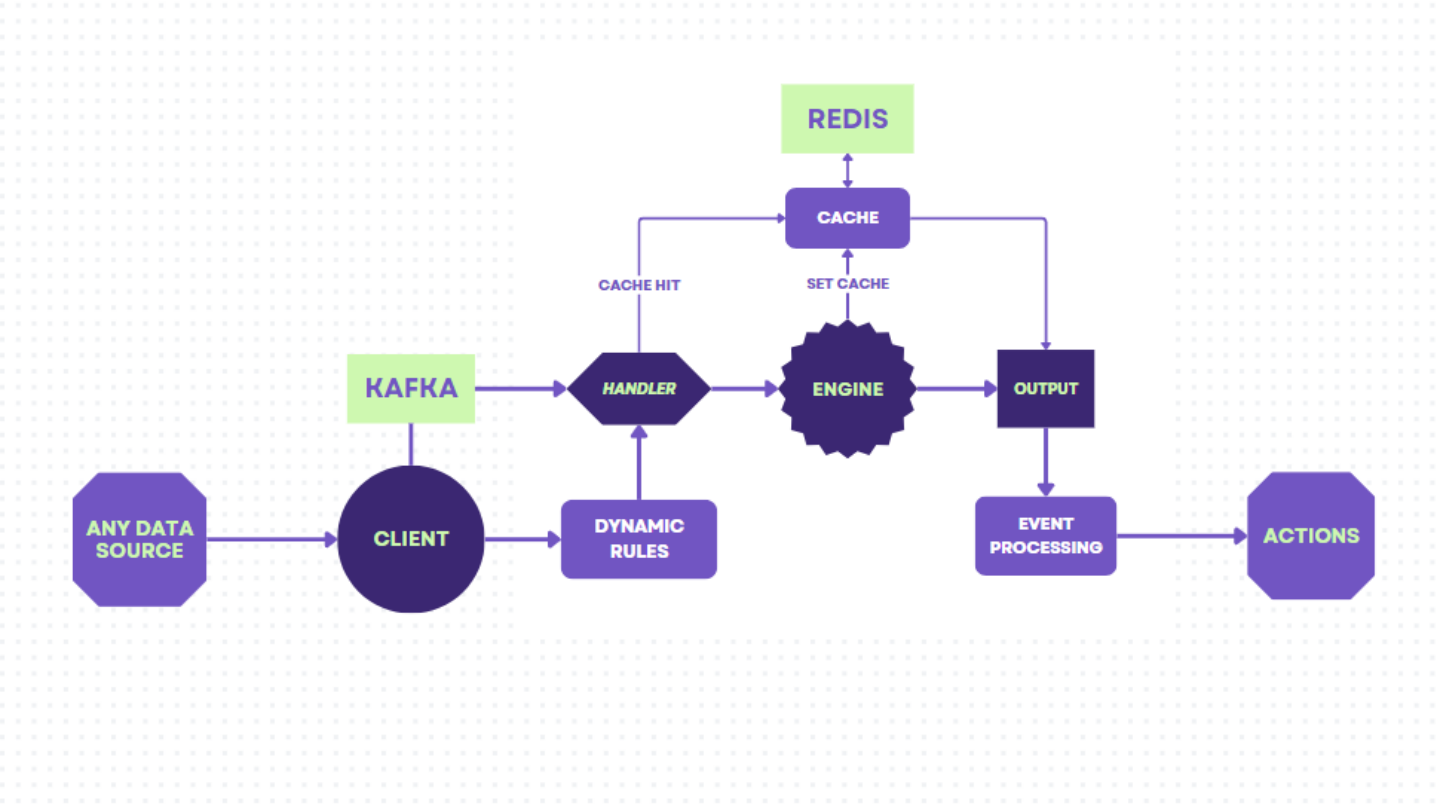
1. Real-Time Rule Evaluation: It can process incoming data streams parallelly in real time, ensuring low-latency responses for high-transaction scenarios.

2. Performance & Scalability: Optimized for high-frequency transactions, ensuring efficient rule execution even under heavy load. The system is designed to scale horizontally (additional instances) and vertically (enhanced processing power).

3. Extensibility: Designed the rule engine to be extensible, supporting new conditions, actions, and data inputs. The system can easily incorporate time-based rules, new action types, and additional data sources in the future.

Architecture

This architecture diagram provides a general overview of technology placement; the final tech stack may vary!



Component Breakdown and Technology Placement

1. Kafka

Kafka topics serve as the entry point for data streams from various sources. Kafka's role is to handle data ingestion at scale. It is designed for high-throughput, real-time data streaming and allows outputting processed results for downstream analysis. Kafka is used here for both data ingestion and output.

2. Kafka Consumer with RxJS for Backpressure Handling

RxJS is integrated with Kafka to manage backpressure in the data processing pipeline. RxJS observables wrap consumed Kafka messages, allowing transformation, buffering, and error handling. Using RxJS operators like throttleTime, bufferTime, and debounceTime ensures that data flows through the system at a manageable rate, adapting to processing bottlenecks and system load.

3. Redis as a Caching Layer

Redis acts as a fast-access cache layer, storing frequently checked or computed data, which reduces redundant processing and speeds up evaluations. Redis can also store unique identifiers to deduplicate data, avoiding reprocessing of previously handled facts.

4. Rule Engine

The rule engine, represented here by a custom logic implementation, evaluates each data event based on defined rules. The data is processed in a manageable flow due to RxJS, and results are categorized into success and failure, depending on rule evaluation outcomes. RxJS also enhances error resilience with its catchError and retry operators. Engine can support multi-threading as well.

```
class Worker {
    constructor(rules, engineOptions) {
        this.#engine = new Engine(rules, engineOptions);
    }
    async run(iterator) {
        while(true) {
            const { value, done } = await iterator.next();
            if (done) {
                break;
            }
            const results = await engine.run(value);
            // do something with results
        }
    }
}

// get the data as an async iterator
const iterator = getDataFromQueue()[Symbol.asyncIterator]();
```

5. Output Handling (Success and Failure)

Data processing results are directed to specific outputs depending on success or failure. Kafka topics, databases, or logs serve as destinations for results. RxJS observables can also regulate output data flow, which further controls data rates if output destinations are a potential bottleneck.

Summary

- Redis: Acts as a cache layer for deduplication and fast access to frequently used data.

- Kafka and RxJS: Kafka handles data ingestion while RxJS manages data flow and backpressure.

- Rule Engine: Evaluates data based on predefined rules, outputting success or failure results to designated streams.

Operators Supported

This Custom rule engine can support below operators and can be extended to support more operators. It can also support Time Interval based Rules.

Here are some of the default operators.

Equality Operators

1. equal - Checks if two values are strictly equal. (Supports multiple Data types)

2. notEqual - Checks if two values are not equal.

List/Array Operators

3. in - Checks if a value is present within an array or string.

4. notIn - Checks if a value is not present within an array or string.

5. contains - Checks if an array contains a specific element.

6. doesNotContain - Checks if an array does not contain a specific element.

Numeric Comparison Operators

7. lessThan - Checks if one number is less than another.

8. lessThanInclusive - Checks if one number is less than or equal to another.

9. greaterThan - Checks if one number is greater than another.

10. greaterThanInclusive - Checks if one number is greater than or equal to another.

Time Comparison Operators

11. before - Checks if a date is before a specified date.

12. after - Checks if a date is after a specified date.

13. onOrBefore - Checks if a date is on or before a specified date.

14. onOrAfter - Checks if a date is on or after a specified date.

15. withinRange - Checks if a date is within a specified start and end date range.

16. notWithinRange - Checks if a date is outside a specified start and end date range.

String Manipulation Operators

17. startsWith - Checks if a string starts with a specified substring.

18. endsWith - Checks if a string ends with a specified substring.

19. matchesRegex - Checks if a string matches a given regular expression pattern.

Open Source Libraries/Frameworks Referred and Used to build this Custom Rule Engine

1. Faust Python: A Python library for stream processing, designed for real-time data processing and complex event-driven workflows.

2. Durable Rules: A C Based Engine used for building rule-based systems with support for long-running workflows and complex rule evaluation.

3. pnpm (8.0): A fast and disk space-efficient package manager for Node.js, providing faster install times and more efficient dependency handling.

4. NodeJS (20.2): A JavaScript runtime environment for executing JavaScript code outside of a browser, enabling server-side development.

5. esbuild (0.24.0): A fast bundler and minifier for JavaScript and TypeScript that significantly improves build performance.

6. eventemitter2 (6.4.4): A highly efficient, asynchronous event emitter for handling custom events in Node.js applications.

7. jsonpath-plus (10.0.0): A powerful JSON querying tool that allows you to extract data from JSON objects using a path-based syntax.

8. hash-it (6.0.0): A lightweight library for generating hash values from strings or objects, used for ensuring data integrity or caching.

9. json-rules-engine (7.0.0): A rule engine for defining and evaluating business rules over JSON data, providing a flexible and scalable solution for rule processing.

10. rulepilot (1.3.1): A Node.js rule engine for evaluating dynamic business logic, designed to integrate easily into applications.

11. rxjs (7.8.1): A reactive programming library for handling asynchronous data streams with support for operators and backpressure handling.

12. tseep (1.3.1): Fastest event emitting utility for efficiently working with Nodejs environments.

13. Fast-Json: A lightning fast on the fly JSON parser able to return JSON values and structures from plain JSON as String or Buffer

References:

- Chapter 2. Phreak rule algorithm in the decision engine | Red Hat Product Documentation
- KafkaJS · KafkaJS, a modern Apache Kafka client for Node.js
- Why Node.js is Your Best Bet for Real-time Apps
- nools
- jruizgit/rules: Durable Rules Engine
- RxJS - Introduction
- pnpm
- esbuild
- eventemitter2
- jsonpath-plus
- hash-it
- json-rules-engine
- rulepilot
- rxjs
- tseep
- Drools - Drools - Business Rules Management System (Java™, Open Source)