**Rule Engine Grammar**

**Overview**

This document provides a comprehensive guide to the Rule Engine Grammar, a JSON-based language for defining and executing business rules. This grammar allows users to express complex logic, access diverse data sources, and trigger actions based on rule outcomes.

**Key Features**

* **Human-readable syntax:** Uses JSON, a widely-used and easily understood data format.
* **Flexible logic:** Supports logical operators (and, or, not), conditional branching (if, elif, else), and nested conditions for expressing complex rules.
* **Data access:** Integrates with various data sources, including databases, APIs, and cloud services, to retrieve the information needed for rule evaluation.
* **Actionable outcomes:** Triggers predefined actions based on rule results, such as sending notifications, updating records, or integrating with other systems.
* **Extensible:** Allows for custom actions and data sources to be added, adapting to specific business needs.
* **Schema validation:** Supports schema definition to ensure data integrity and early error detection.

**Grammar Structure**

The Rule Engine Grammar is structured as a JSON object with the following key components:

**1. ruleGrammarVersion**

Specifies the version of the grammar used.

**Example:**

JSON

"ruleGrammarVersion": "5.0"

**2. metadata**

Contains metadata about the rule, including:

* uid: Unique identifier for the rule.
* ruleName: Human-readable name of the rule.
* domain: Business domain the rule belongs to.
* description: Description of the rule's purpose.
* copiedFromRule: Indicates if the rule was copied from another rule.
* copiedFromVersion: Version of the copied rule.
* createdBy: User who created the rule.
* approvedBy: User who approved the rule.
* status: Current status of the rule (e.g., "active", "inactive").
* ruleExpiry: Defines the expiry date of the rule.
* createdDate: Date and time when the rule was created.
* lastUpdated: Date and time when the rule was last updated.
* ruleVersion: Version number of the rule.
* categories: Categories the rule belongs to.
* dependencies: Other rules that this rule depends on.

**Example:**

JSON

"metadata": {

"uid": "rule123",

"ruleName": "Customer Eligibility Rule",

"domain": "Eligibility",

"description": "Determines customer eligibility for a promotion.",

// ... other metadata fields ...

}

**3. cache**

Defines caching options for the rule, including:

* cachable: Indicates whether the rule results can be cached.
* ttl: Time-to-live for cached results (in seconds).
* cacheKey: Key used to identify cached results.

**Example:**

JSON

"cache": {

"cachable": true,

"ttl": 3600,

"cacheKey": "customer\_${customerId}"

}

**4. actions**

Specifies actions to be taken based on the rule outcome. Actions are categorized by type:

* always: Actions to be executed regardless of the rule outcome.
* onSuccess: Actions to be executed if the rule evaluates to true.
* onFailure: Actions to be executed if the rule evaluates to false.
* onError: Actions to be executed if an error occurs during rule evaluation.
* afterRun: Actions to be executed after the rule has been evaluated, regardless of the outcome.

Each action type has an actions array containing individual actions. Each action has a sequence number to define the order of execution.

**Supported action types:**

* postToQueue: Posts a message to a queue.
* log: Logs a message.
* conditional: Executes actions based on a condition.
* sendEmail: Sends an email.
* sendSMS: Sends an SMS message.
* notifyAdmin: Notifies an administrator.
* custom: Executes a custom action.

**Example:**

JSON

"actions": [

{

"type": "onSuccess",

"actions": [

{

"sequence": 1,

"type": "log",

"message": "Customer is eligible."

},

{

"sequence": 2,

"type": "sendEmail",

"to": { "parameter": "$customerEmail" },

"subject": "Eligibility Confirmation"

}

]

}

]

**5. parameters**

Defines input parameters for the rule. Each parameter has:

* value: Value of the parameter.
* mandatory: Indicates whether the parameter is required.
* description: Description of the parameter.
* type: Data type of the parameter.
* optionalFilterBehavior: Behavior if the parameter is optional and not provided.
* enum: Allowed values for the parameter (if applicable).
* allowedValues: Alternative way to specify allowed values.

**Example:**

JSON

"parameters": {

"customerId": {

"value": "$customerId",

"mandatory": true,

"description": "Unique identifier for the customer.",

"type": "string"

}

}

**6. preCondition**

Specifies conditions that must be met before the rule is evaluated. It uses the same structure as the ruleDefinition.

**Example:**

JSON

"preCondition": {

"op": "exists",

"entityName": "customer",

"entityType": "data",

"filters": [

{

"field": "customer\_id",

"type": "string",

"comparison": "equals",

"value": { "parameter": "$customerId" }

}

]

}

**7. ruleDefinition**

Contains the core logic of the rule, using operators and conditions to determine the outcome.

**Operators:**

* and, or, not: Logical operators for combining conditions.
* ifelse: Defines an if-else branching structure with if, elif, and else operators.
* exists: Checks for the existence of an entity.

**Conditions:**

* field: Specifies the field to be evaluated.
* type: Specifies the data type of the field.
* comparison: Specifies the comparison operator (e.g., equals, greaterThan, contains).
* value: Specifies the value to compare the field against.

**Example:**

JSON

"ruleDefinition": {

"op": "ifelse",

"terms": [

{

"op": "if",

"terms": [

// ... conditions and actions ...

]

},

{

"op": "elif",

"terms": [

// ... conditions and actions ...

]

},

{

"op": "else",

"terms": [

// ... conditions and actions ...

]

}

]

}

**8. dataExtraction**

Defines how data is extracted from various sources. It specifies:

* entityName: Name of the entity to extract data from.
* entityType: Type of the entity (e.g., "data").
* filters: Filters to apply when extracting data.
* extractedAttributes: Attributes to extract.
* aggregator: Aggregator function to apply to extracted data.
* transformation: Transformation to apply to extracted data.

**Example:**

JSON

"dataExtraction": [

{

"entityName": "customer",

"entityType": "data",

"filters": [

// ... filters ...

],

"extractedAttributes": [

"customer\_id",

"age",

"email"

]

}

]

**9. dataAccess**

Specifies data sources and access methods. It includes:

* dataSources: Database connections.
* apis: API endpoints.
* functions: Custom functions.
* cloudServices: Cloud service integrations (e.g., AWS Lambda, SageMaker).
* graphql: GraphQL endpoints.
* connections: Connection details for data sources.

**Example:**

JSON

"dataAccess": {

"dataSources": [

{

"alias": "customerDb",

"type": "database",

"connection": "customerDbConnection",

// ... other database connection details ...

}

]

}

**10. result**

Defines the output structure of the rule, including:

* ruleResult: Status, timestamp, execution time, and failure reasons.
* extractedData: Data extracted from various sources.
* ruleAttributes: Attributes used in the rule evaluation.
* outcome: Outcome of the rule evaluation (success, failure, error) and associated return values.

**Example:**

JSON

"result": {

"ruleResult": {

"status": {

"type": "string",

"enum": ["passed", "failed", "error"]

},

// ... other rule result fields ...

}

}

**Schema Validation**

The schema field within the ruleDefinition allows you to define the expected structure of the data using JSON Schema. This helps ensure data integrity and early error detection.

**Example:**

JSON

"ruleDefinition": {

"schema": {

"type": "object",

"properties": {

"customer": {

"type": "object",

"properties": {

"age": { "type": "integer", "minimum": 0 },

"status": { "type": "string", "enum": ["active", "inactive"] }

},

"required": ["age", "status"]

}

},

"required": ["customer"]

},

// ... your existing rule logic ...

}

**Best Practices**

* Use clear and concise rule names and descriptions.
* Break down complex rules into smaller, more manageable ones.
* Use schema validation to ensure data integrity.
* Implement robust error handling mechanisms.
* Thoroughly test your rules before deploying them.

**Conclusion**

The Rule Engine Grammar provides a powerful and flexible way to define and execute business rules. By understanding its structure and features, you can leverage its capabilities to automate decision-making processes and improve business efficiency.

Reasoned for a second

Below is an enhanced table of capabilities for the rule engine, including the new features such as nesting of terms, calling different rule engines, flexible filter conditions, and robust action handling.

| **Capability** | **Description** |
| --- | --- |
| **Dynamic Parameter Substitution** | Allows the use of parameter references (e.g., $customerId) to dynamically inject values into rules at runtime, ensuring reusability and flexibility. |
| **Advanced Branching Logic** | Supports conditional branching using dedicated structures (e.g., ifelse with nested if, elif, and else branches) to handle complex decision trees. |
| **Nesting of Terms** | Enables deeply nested logical expressions, allowing rules to combine conditions in multiple layers for complex evaluations. |
| **Precondition Evaluation** | Enforces preliminary checks (e.g., customer existence, non-weekend execution) before executing the main rule logic. |
| **Core Rule Evaluation** | Uses nested logical operators (and, or, exists, not) to evaluate conditions based on field comparisons, expressions, and aggregated data. |
| **Calling Different Rule Engines** | Supports integration with multiple rule engines (e.g., via a ruleEngine property) so that specific rules can be delegated to different processing engines if needed. |
| **Flexible Filter Conditions** | Provides a robust filtering system to compare fields using various operators (equals, neq, gt, gte, lt, lte, between, matches, etc.), supporting both literal and dynamic values. |
| **Aggregation and Transformation** | Enables data enrichment by applying aggregation functions (e.g., sum, count) and transformation expressions (e.g., string manipulation) to extracted data. |
| **Data Extraction for Enrichment** | Separates data enrichment logic into a dedicated block, allowing selective extraction of attributes from external data sources without interfering with rule logic. |
| **Comprehensive Data Access** | Integrates with multiple external sources (databases, APIs, cloud services, GraphQL) and centralizes connection management for scalable configuration. |
| **Caching Support** | Provides caching capabilities with TTL and dynamic cache keys to optimize performance by reusing evaluated rule results when possible. |
| **Centralized Security** | Offers a centralized security block that applies global settings (masking, tokenization, encryption) with the option to override for sensitive parameters. |
| **Robust Error Handling** | Includes onError configurations at multiple levels to specify strategies (retry, evaluateFalse, log, etc.) and ensure graceful degradation and debugging support. |
| **Rich Action Handling** | Defines actions for multiple execution phases (always, onSuccess, onFailure, onError, afterRun) with sequencing and conditional logic to drive diverse outcomes. |
| **Result Structuring** | Structures the final output into distinct sections (ruleResult, extractedData, ruleAttributes, outcome) for clear communication of execution metadata and business outcomes. |

This table provides a comprehensive overview of the capabilities of the rule engine DSL, highlighting its flexibility, extensibility, and robust design for enterprise-grade rule processing.