Rulez

Release 0.1.4

Dec 28, 2020

Contents:

CHAPTER 1

rulez: JSON based conditional rules

Rulez is a library that help with defining comparator/conditional operations declaratively through dictionary / JSON structure. Currently, it provides a compiler that compiles the declarative rules into python function, sqlalchemy comparators/conditions, and elasticsearch comparators.

CHAPTER 2

Quickstart

2.1 Installation

```
pip install rulez
```

2.2 Defining & Compiling Comparators

`rulez` provide a way to compile comparators from JSON based ruleset.

```
from rulez import Engine
from rulez import OperatorNotAllowedError
from rulez import NestedOperationNotAllowedError
from rulez.operator import Operator

rule = {
    "operator": "or", "value": [
        {"field": "age", "operator": "<=", "value": 16},
        {"field": "age", "operator": ">=", "value": 21},
        {"field": "age", "operator": "==", "value": 18}
    ]
}

engine = Engine()

f = engine.compile_condition('native', rule)

assert f({'age': 13}) is True
assert f({'age': 17}) is False
assert f({'age': 21}) is True
assert f({'age': 19}) is False
```

2.2.1 Available Compilation Method

- `native` - compile to python condition
- `sqlalchemy` - compile to a factory function to generate sqlalchemy filter
- `elasticsearch` - compile a factory function that generate elasticsearch filter

2.3 Defining & Compiling Rule Chain

Besides compiling conditionals, `rulez` also provide a way to compile if-elif-else ruleset.

```
from rulez import Engine

rulechain = [
    {'condition': {
        'field': 'age',
        'operator': '<',
        'value': 18},
      'actions': [{
          'action': 'set',
          'parameter': {
              'field': 'category',
              'value': 'underage'
          }
      }]},
    {'condition': {
        'field': 'age',
        'operator': '>',
        'value': 50},
      'actions': [{
          'action': 'set',
          'parameter': {
              'field': 'category',
              'value': 'senior'}}]},
    {'actions': [{
        'action': 'set',
        'parameter': {
            'field': 'category',
            'value': 'adult'}}]}
]

engine = Engine()
c = engine.compile_rulechain('native', rulechain)

assert c({'age': 10}) == {'age': 10, 'category': 'underage'}
assert c({'age': 18}) == {'age': 18, 'category': 'adult'}
assert c({'age': 55}) == {'age': 55, 'category': 'senior'}
```

2.3.1 Available Compilation Method

- `native` - compile to python rulechain
- `sqlalchemy` - compile to a factory function to generate sqlalchemy query that adds a new result column

CHAPTER 3

Comparators

Comparator operations are declared using JSON with following structure:

```
{"field": "<fieldname>", "operator", "<operator>", "value": "<comparison value>"}
```

3.1 Available operators

class rulez.operator.**And**(*operator*, *engine*, *value*, *value_type=None*)
Apply boolean AND condition to list of values

Operator and

Value_types list, tuple

Items in values have to be a compilable rulez operator JSON

class rulez.operator.**Equal**(*operator*, *engine*, *field*, *value*, *value_type=None*)
Compare whether contents of field is same as value

Operator ==

Value_types rulez_operation, str, float, int

class rulez.operator.**Get**(*operator*, *engine*, *value*, *value_type=None*)
Get value from a field

Operator get

Value_types field_name

class rulez.operator.**GreaterEqualThan**(*operator*, *engine*, *field*, *value*, *value_type=None*)
Compare whether contents of field is greater equal than value

Operator >=

Value_types rulez_operation, str, float, int

class rulez.operator.**GreaterThan**(*operator*, *engine*, *field*, *value*, *value_type=None*)
Compare whether contents of field is greater than value

Operator >

Value_types rulez_operation, str, float, int

class rulez.operator.**In**(*operator*, *engine*, *field*, *value*, *value_type=None*)
Apply boolean OR condition to list of values

Operator or

Value_types list[rulez_operation], tuple[rulez_operation], list[str], list[float], list[int]

class rulez.operator.**LessEqualThan**(*operator*, *engine*, *field*, *value*, *value_type=None*)
Compare whether contents of field is less equal than value

Operator <=

Value_types rulez_operation, str, float, int

class rulez.operator.**LessThan**(*operator*, *engine*, *field*, *value*, *value_type=None*)
Compare whether contents of field is less than value

Operator <

Value_types rulez_operation, str, float, int

class rulez.operator.**Like**(*operator*, *engine*, *field*, *value*, *value_type=None*)
Compare whether contents of field is like value

Operator ~

Value_types str

class rulez.operator.**Match**(*operator*, *engine*, *field*, *value*, *value_type=None*)
Compare whether contents of field matches value

Operator match

Value_types str

> **Warning:** This operator only supported in sqlalchemy compiler.

class rulez.operator.**NotEqual**(*operator*, *engine*, *field*, *value*, *value_type=None*)
Compare whether contents of field is not equal to value

Operator !=

Value_types rulez_operation, str, float, int

class rulez.operator.**Or**(*operator*, *engine*, *value*, *value_type=None*)
Apply boolean OR condition to list of values

Operator or

Value_types list[rulez_operation], tuple[rulez_operation]

Items in values have to be a compilable rulez operation JSON

3.2 Comparing contents of 2 fields

We can also compare contents of 2 fields by using comparison operators

```
{
      "field": "<fieldname1>",
      "operator": "<operator>",
      "value": {
          "operator": "get",
          "value": "<fieldname2>"
       }
}
```

CHAPTER 4

Syntax Sugar / DSL

Rulez provide several syntax sugar to help with constructing comparison operations.

4.1 Pythonic Operation

You can easily create a comparison operation using `rulez.field`. Eg:

`FieldGetter.__call__(key)`

```
>>> import rulez

>>> rulez.field('field1')
{ 'operator': 'get', 'value': 'field1' }

>>> rulez.field('field1') == 'myvalue'
{ 'field': 'field1', 'operator': '==', 'value': 'myvalue' }

>>> rulez.field('field1') == rulez.field('field2')
{ 'field': 'field1', 'operator': '==',
  'value': {
        'operator': 'get',
        'value': 'field2' }}

>>> rulez.field('field1') & rulez.field('field2')
{'operator': 'and',
 'value': [
     {'operator': 'get', 'value': 'field1'},
     {'operator': 'get', 'value': 'field2'}]}

>>> rulez.field('field1') | rulez.field('field2')
{'operator': 'or',
 'value': [
```

(continues on next page)

(continued from previous page)

```
    {'operator': 'get', 'value': 'field1'},
    {'operator': 'get', 'value': 'field2'}]}
```

FieldGetter.**__getitem__**(*key*)

```
>>> import rulez

>>> rulez.field['field1']
{ 'operator': 'get', 'value': 'field1' }

>>> rulez.field['field1'] == 'myvalue'
{ 'field': 'field1', 'operator': '==', 'value': 'myvalue' }

>>> rulez.field['field1'] == rulez.field['field2']
{ 'field': 'field1', 'operator': '==',
  'value': {
        'operator': 'get',
        'value': 'field2' }}

>>> rulez.field['field1'] & rulez.field['field2']
{'operator': 'and',
 'value': [
    {'operator': 'get', 'value': 'field1'},
    {'operator': 'get', 'value': 'field2'}]}

>>> rulez.field['field1'] | rulez.field['field2']
{'operator': 'or',
 'value': [
    {'operator': 'get', 'value': 'field1'},
    {'operator': 'get', 'value': 'field2'}]}
```

Note: `rulez.field[key]` is deprecated at version 0.1.4 in favor of `rulez.field(key)`.

4.2 Boolean DSL Statement

Comparison operation can also be created from string using `rulez.parse_dsl`

rulez.**parse_dsl**(*s*, *allowed_operators: Optional[List[str]] = None*) → rulez.dsl.Operation

Parse string and output comparison operation.

Parameters

- **s** – query string
- **allowed_operators** (*typing.Optional[typing.List[str]]*) – List of allowed operators

```
>>> import rulez
>>> rulez.parse_dsl("field1 == field2")
{'field': 'field1', 'operator': '==',
 'value': {'operator': 'get',
           'value': 'field2'}}
```

(continues on next page)

(continued from previous page)

```
>>> rulez.parse_dsl("field1 == 'hello world'")
{'field': 'field1', 'operator': '==',
 'value': 'hello world'}

>>> rulez.parse_dsl("(field1 == field2) or (field3 == 'value1')")
{'operator': 'or',
 'value': [{'field': 'field1', 'operator': '==',
            'value': {'operator': 'get', 'value': 'field2'}},
           {'field': 'field3', 'operator': '==',
            'value': 'value1'}]}

>>> rulez.parse_dsl('field1 in ["a","b","c"]')
{'field': 'field1', 'operator': 'in', 'value': ['a', 'b', 'c']}

>>> rulez.parse_dsl('field1 in in [1.0,2.0,3.0]')
{'field': 'field1', 'operator': 'in', 'value': [1.0, 2.0, 3.0]}
```

CHAPTER 5

Compilers

5.1 Native

Compile into Python function

This compiler compiles ruleset into an executable Python function.

Example:

```
>>> import rulez
>>> engine = rulez.Engine()
>>> rule = {
...     "operator": "or", "value": [
...         {"field": "age", "operator": "<=", "value": 16},
...         {"field": "age", "operator": ">=", "value": 21},
...         {"field": "age", "operator": "==", "value": 18}
...     ]
... }
>>> f = engine.compile_condition('native', rule)
>>> f({'age': 13})
True
```

5.2 SQLAlchemy

Compile into SQLAlchemy condition factory.

This compiler compiles into a factory function that that accepts SQLAlchemy ORM model as its parameter, and outputs a SQLAlchemy condition object for use in `.filter()` method in queries.

Example:

```
>>> from sqlalchemy.ext.declarative import declarative_base
>>> import sqlalchemy as sa
```

(continues on next page)

(continued from previous page)

```
>>> from sqlalchemy.orm import sessionmaker
>>> import rulez

>>> Session = sessionmaker()
>>> Base = declarative_base()

>>> class Object(Base):
...     __tablename__ = 'test_objects'
...     id = sa.Column(sa.Integer, primary_key=True, autoincrement=True)
...     age = sa.Column(sa.Integer, name='Age')

>>> engine = sa.create_engine('sqlite://')
>>> Session.configure(bind=engine)
>>> Base.metadata.create_all(engine)
>>> session = Session()

>>> session.add(Object(age=50))
>>> session.add(Object(age=18))
>>> session.add(Object(age=10))
>>> session.add(Object(age=17))
>>> session.add(Object(age=19))
>>> session.flush()

>>> rule = {
...     "operator": "or", "value": [
...         {"field": "age", "operator": "<=", "value": 16},
...         {"field": "age", "operator": ">=", "value": 21},
...         {"field": "age", "operator": "==", "value": 18}
...     ]
... }

>>> engine = rulez.Engine()
>>> cond = engine.compile_condition('sqlalchemy', rule)

>>> q = session.query(Object).filter(cond(Object))
>>> sorted([o.age for o in q.all()])
[10, 18, 50]
```

5.3 ElasticSearch

Compile into ElasticSearch condition factory.

This compiler compiles ruleset into a factory function that returns a json dictionary for use in elasticsearch `'query'` body.

Example:

```
>>> import rulez
>>> engine = rulez.Engine()
>>> rule = {
...     "operator": "or", "value": [
...         {"field": "age", "operator": "<=", "value": 16},
...         {"field": "age", "operator": ">=", "value": 21},
...         {"field": "age", "operator": "==", "value": 18}
...     ]
```

(continues on next page)

(continued from previous page)

```
... }

>>> f = engine.compile_condition('elasticsearch', rule)
>>> f()
{'bool': {'minimum_should_match': 1,
          'should': [{'range': {'age': {'lte': 16}}},
                     {'range': {'age': {'gte': 21}}},
                     {'term': {'age': 18}}]}}
```

Python Module Index

r

Index

Symbols

A

E

G

I

L

M

N

O

P

R