Logic Engine

Table of contents

The Rule

The rule object represents the building block for the system. A rule is an abstraction for a function acting on the value of a type and returning a boolean response.

> **DEFINITION**: A `Rule` is satisfied by an item `t` of type `T` if the associated function `f: T ⟶ bool` returns true if `f(t)` is `true`.

Given a type to be applied to, a rule is defined by a set of fields

- `Property`: identifies the property against which to execute the evaluation
- `Operator`: defines the operation to execute on the property
- `Value`: identifies the value against which the result of the operator on the property should be compared
- `Code`: the error code to be generated when the rules applied on an object fail (returns `false`)

The Operators

The `Operator` can assume different possible values depending on the `Property` it is applied to and on the value, the result should be compared to.

Operators are classified based on the way they work and their behavior. The rules categorization is also influenced by some implementation details.

Direct operators

These operators directly compare the `Property` to the `Value` considered as a constant:

- `Equal`: equality on value types (strings, numbers, ...)
- `NotEqual`: inequality on value types (strings, numbers, ...)
- `GreaterThan`: only applies to numbers
- `GreaterThanOrEqual`: only applies to numbers
- `LessThan`: only applies to numbers
- `LessThanOrEqual`: only applies to numbers

```
public class MyClass
{
    public string StringProperty {get; set;}
    public int IntegerProperty {get; set;}
}

var stringRule = new Rule("StringProperty", OperatorType.Equal, "Some Value",
"code 1");
var integerRule = new Rule("IntegerProperty", OperatorType.Equal, "10", "code 2");

var myObj = new MyClass
{
    StringProperty = "Some Value",
    IntegerProperty = 11
}

var result1 = stringRule.Apply(myObj); // returns true
var result2 = integerRule.Apply(myObj); // returns false
```

> Sample rules with direct operators

Internal direct operators

Internal direct rules are similar to direct rules, but they are meant to be applied to values that are other fields of the same type; in this case, `Value` should correspond to the name of

another field in the analyzed type:

- `InnerEqual`: equality between two value typed fields
- `InnerNotEqual`: equality between two value typed fields
- `InnerGreaterThan`: only applies when `Property` and `Value` are numbers
- `InnerGreaterThanOrEqual`: only applies when `Property` and `Value` are numbers
- `InnerLessThan`: only applies when `Property` and `Value` are numbers
- `InnerLessThanOrEqual`: only applies when `Property` and `Value` are numbers

```
public class MyClass
{
    public string StringProperty1 {get; set;}
    public string StringProperty2 {get; set;}
    public int IntegerProperty1 {get; set;}
    public int IntegerProperty2 {get; set;}
}

var stringRule = new Rule("StringProperty1", OperatorType.InnerEqual,
"StringProperty2", "code 1");
var integerRule = new Rule("IntegerProperty1", OperatorType.InnerGreaterThan,
"IntegerProperty2", "code 2");
```

> Sample rules with internal direct operators

String direct operators

These rules are specific to strings:

- `StringStartsWith`: checks that the string in `Property` starts with `Value`
- `StringEndsWith`: checks that the string in `Property` ends with `Value`
- `StringContains`: checks that the string in `Property` contains `Value`
- `StringRegexIsMatch`: checks that the string in `Property` matches `Value`

```
public class MyClass
{
    public string StringProperty {get; set;}
}

var stringRule = new Rule("StringProperty", OperatorType.StringStartsWith, "start",
"code 1");
```

> Sample rule with string direct operator

Enumerable operators

These rules apply to operands of generic enumerable type:

- `Contains`: checks that `Property` contains `Value`
- `NotContains`: checks that `Property` does not `Value`
- `Overlaps`: checks that `Property` has a non-empty intersection with `Value`
- `NotOverlaps`: checks that `Property` has an empty intersection with `Value`

```
public class MyClass
{
    public IEnumerable<string> StringEnumerableProperty {get; set;}
}

var rule1 = new Rule("StringEnumerableProperty", OperatorType.Contains, "value",
"code 1");
var rule2 = new Rule("StringEnumerableProperty", OperatorType.Overlaps,
"value1,value2", "code 2");
```

> Sample rules with enumerable operators

Internal enumerable operators

These operators act on enumerable fields by comparing them against fields of the same type:

- `InnerContains`: checks that `Property` contains the value contained in the property `Value`
- `InnerNotContains`: checks that `Property` doesn't contain the value contained in the property `Value`
- `InnerOverlaps`: checks that `Property` has a non-empty intersection with the value contained in the property `Value`
- `InnerNotOverlaps`: checks that `Property` has an empty intersection with the value contained in the property `Value`

```
public class MyClass
{
    public IEnumerable<int> EnumerableProperty1 {get; set;}
    public IEnumerable<int> EnumerableProperty2 {get; set;}
    public int IntegerField {get; set;}
}

var rule1 = new Rule("EnumerableProperty1", OperatorType.InnerContains,
"IntegerField");
```

```
var rule2 = new Rule("EnumerableProperty1", OperatorType.InnerOverlaps,
"EnumerableProperty2");
```

> Sample rules for internal enumerable operators

Key-value operators

These operators act on dictionary-like objects:

- `ContainsKey`: checks that the `Property` contains the specific key defined by the `Value`
- `NotContainsKey`: checks that the `Property` doesn't contain the specific key defined by the `Value`
- `ContainsValue`: checks that the dictionary `Property` contains a value defined by the `Value`
- `NotContainsValue`: checks that the dictionary `Property` doesn't contain a value defined by the `Value`
- `KeyContainsValue`: checks that the dictionary `Property` has a key with a specific value
- `NotKeyContainsValue`: checks that the dictionary `Property` doesn't have a key with a specific value

```
public class MyClass
{
    public IDictionary<string, int> DictProperty {get; set;}
}

var rule1 = new Rule("DictProperty", OperatorType.ContainsKey, "mykey");
var rule2 = new Rule("DictProperty", OperatorType.KeyContainsValue,
"mykey[myvalue]");
```

> sample rules for key-value enumerable operators

Inverse enumerable operators

These rules apply to scalars against enumerable fields:

- `IsContained`: checks that `Property` is contained in a specific set
- `IsNotContained`: checks that `Property` is not contained in a specific set

```
public class MyClass
{
    public int IntProperty {get; set;}
}
```

```
var rule1 = new Rule("IntProperty", OperatorType.IsContained, "1,2,3");
```

> Sample rules for inverse enumerable operators

The RulesSets

A `RulesSet` is a set of rules. From a functional point of view, it represents a boolean typed function composed by a set of functions on a given type.

> **DEFINITION**: A `RulesSet` is satisfied by an item `t` of type `T` if all the functions of the set are satisfied by `t`.

A `RulesSet` corresponds to the logical `AND` operator on its rules.

The RulesCatalog

A `RulesCatalog` represents a set of `RulesSet`, and functionally corresponds to a boolean typed function composed by a set of sets of functions on a given type.

> **DEFINITION**: A `RulesCatalog` is satisfied by an item `t` of type `T` if at least one of its `RulesSet`s is satisfied by `t`.

A `RulesCatalog` corresponds to the logical `OR` operator on its `RulesSet`s.

The Algebraic model

As discussed above, composite types `RulesSet` and `RulesCatalog` represent logical operations on the field of functions `f: T ⟶ bool`; it seems than possible to define an algebraic model defining the composition of different entities.

RulesSets product

> **DEFINITION**: The product of two `RulesSet`s is a new `RulesSet`, and its rules are a set of rules obtained by concatenating the rules of the two `RulesSet`s

```
rs1 = {r1, r2, r3}
rs2 = {r4, r5}

⟶ rs1 * rs2 = {r1, r2, r3, r4, r5}
```

> product of two `RulesSet`s

RulesCatalog sum

The sum of two `RulesCatalog` objects is a `RulesCatalog` with a set of `RulesSet` obtained by simply concatenating the two sets of `RulesSet`:

```
c1 = {rs1, rs2, rs3}
c2 = {rs4, rs5}

⟶ c1 + c2 = {rs1, rs2, rs3, rs4, rs5}
```

> sum of two `RulesCatalog`

RulesCatalog product

The product of two catalogs is a catalog with a set of all the `RulesSet` obtained concatenating a set of the first catalog with one of the second.

```
c1 = {rs1, rs2, rs3}
c2 = {rs4, rs5}

⟶ c1 * c2 = {(rs1*rs4), (rs1*rs5), (rs2*rs4), (rs2*rs5), (rs3*rs4), (rs3*rs5)}
```

> product of two `RulesCatalog`

Compilers and compiled objects

The `RuleCompiler` is the component that parses and compiles a `Rule` into executable code.

Every rule becomes an `Option<CompiledRule<T>>`, where the `None` status of the option corresponds to a `Rule` that is not formally correct and hence cannot be compiled[^1]. A `CompiledRule<T>` is the actual portion of code that can be applied to an item of type `T` to provide a boolean result using its `ApplyApply(T item)` method. Sometimes the boolean result is not enough: when the rule is not satisfied it could be useful to understand the reason why it failed. For this reason, a dedicated `Either<string, Unit> DetailedApply(T item)` method returns `Unit` when the rule is satisfied, or a string (the rule code) in case of failure.

Like the `RuleCompiler`, the `RulesSetCompiler` transforms a `RulesSet` into an `Option<CompiledRulesSet<T>>`. A `CompiledRulesSet<T>` can logically be seen as a set of

compiled rules, hence, when applied to an item of type `T` it returns a boolean that is `true` if all the compiled rules return `true` on it. From a logical point of view, a `CompiledRulesSet<T>` represents the `AND` superposition of its `CompiledRule<T>`. The corresponding `Either<string, Unit> DetailedApply(T item)` method of the `CompiledRulesSet<T>` returns `Unit` when all the rules are satisfied, or the set of codes for the rules that are not.

Finally, the `RulesCatalogCompiler` transforms a `RulesCatalog` into an `Option<CompiledCatalog<T>>`, where the `None` status represents a catalog that cannot be compiled. A `CompiledCatalog<T>` logically represents the executable code that applies a set of rule sets to an object of type `T`: the result of its application can be `true` if at least one set of rules returns `true`, otherwise `false` (this represents the logical `OR` composition operations on rules joined by a logical `AND`). Similar to the `Either<string, Unit> DetailedApply(T item)` of the `CompiledRulesSet<T>`, it can return `Unit` when at least one internal rule set returns `Unit`, otherwise the flattened set of all the codes for all the rules that don't successfully apply.

Known limitations

The current implementation of the rules system has some limitations:

- it is designed to work on plain objects (instances of classes, records, or structures) with an empty constructor
- rules can only be applied to 'first level members', no nesting is currently supported

---

Breaking changes

If you want to upgrade from a version < 3.0.0 to the latest version you will need to adapt your implementation to manage the breaking changes introduced.

The main differences can be condensed in the removal of the managers: the entire logic is now completely captured by the compiled objects `CompiledRule<T>`, `CompiledRulesSet<T>`, `CompiledCatalog<T>`, without the need of external wrappers.

This means that the typical workflow to update the library requires:

1. getting the rules definition
2. pass them to the appropriate compiler
3. use the generated compiled objects to validate your objects according to the rules definition

---

How to install the package

If you are using `nuget.org` you can add the dependency in your project using

```
dotnet add package logic-engine --version <version>
```

To install the **logic-engine** library from GitHub's packages system please refer to the packages page.

[^0]: from a technical perspective this is obtained with a concrete implementation of the railway pattern

[^1]: null or empty codes are removed because they don't carry reusable info

Namespace LogicEngine

Classes

CompiledCatalog<T>

CompiledRule<T>

CompiledRulesSet<T>