DataSpec2

Data Specification 2.0

Draft 1, 2015-08-03

This is the DeviceMap Data specification for patterns and attributes.

Overview

This document goes over how DeviceMap Data 2.0 is defined and how the classifiers process input against the data.

The classification process is broken down into three phases:

- Input Parsing
- Pattern Matching
- Attribute Retrieval

The following definitions are used:

input string::

- *

  ${renderedContent}

token stream::

- *

  ${renderedContent}

pattern::

- *

  ${renderedContent}

pattern tokens::

- *

  ${renderedContent}

pattern type::

- *

  ${renderedContent}

matched tokens::

- *

  ${renderedContent}

pattern rank::

- *

  ${renderedContent}

candidate::

- *

  ${renderedContent}

pattern set::

- *

  ${renderedContent}

file::

- *

  ${renderedContent}

domain::

- *

  ${renderedContent}

client::

- *

  ${renderedContent}

Input Parsing

This step parses the input string and creates the token stream.

Each pattern file defines these input parsing rules:

InputTransformers::

- *

  ${renderedContent}

- *

  ${renderedContent}

TokenSeparators::

- *

  ${renderedContent}

- *

  ${renderedContent}

NgramConcatSize::

- *

${renderedContent}

- *

${renderedContent}

The input string first gets processed thru the transformers. Then it gets tokenized using the configured seperators. Then ngram concatenation happens. The final result of these 3 steps is the token stream.

Notes

Empty tokens are removed from the tokenization step.

When a token is added to the token stream, it can be processed by the pattern matching step before moving on to the next token. This algorithm is pipeline and thread safe.

If the Ngram{{`Concat}}`Size is greater than 1, ngrams must be added to the token stream ordered largest to smallest.

Example

```
InputTransformers: Lowercase(), ReplaceAll(find: '-', replaceWith: '')
TokenSeparators:   [space]
NgramConcatSize:   2

Input string:  'A 12 x-yZ'

Transform:     'a 12 xyz'

Tokenization:  a, 12, xyz

Ngram:         a12, a, 12xyz, 12, xyz
```

Pattern Matching

This step processes the token stream and returns the highest ranking candidate pattern.

The pattern file defines a pattern set. All patterns in the pattern set are evaluated to find the candidates.

Each pattern has 2 main attributes, its pattern type and its pattern rank. The pattern type defines how the pattern is supposed to be matched against the token stream. The pattern rank defines how the pattern ranks against other patterns.

All the pattern types in 2.0 are prefixed with 'Simple'. This means that each pattern token is matched using a plain byte or string comparison. No regex or other syntax is allowed in Simple patterns. This allows the algorithm to use simple byte or string hashing for matching. This gives maximum performance and scaling complexity equal to a hashtable implementation. A Simple{{`Hash}}`Count attribute can be optionally defined which hints the classifier as to how many unique hashes it would need to generate to support the pattern set.

Pattern attributes:

PatternId::

- *

${renderedContent}

- *

${renderedContent}

RankType::

- *

${renderedContent}

- *

${renderedContent}

RankValue::

- *

${renderedContent}

- *

  ${renderedContent}

PatternType::

- *

  ${renderedContent}

- *

  ${renderedContent}

PatternTokens::

- *

  ${renderedContent}

- *

  ${renderedContent}

Pattern set attributes:

DefaultId::

- *

  ${renderedContent}

- *

  ${renderedContent}

SimpleHashCount::

- *

  ${renderedContent}

- *

  ${renderedContent}

PatternType

The following pattern types are defined:

SimpleOrderedAnd::

- *

  ${renderedContent}

SimpleAnd::

- *

  ${renderedContent}

Simple::

- *

  ${renderedContent}

RankType

The following rank types are defined:

Strong::

- *

  ${renderedContent}

Weak::

- *

  ${renderedContent}

None::

- *

  ${renderedContent}

In the case where 2 or more candidates have the same Rank{{`Type and Rank}}`Value resulting in a tie, the candidate with the longest concatenated matched pattern length is used. If that results in another tie, the candidate with the first matched token found is returned.

Notes

If no candidate patterns are found, the DefaultId is returned. If no DefaultId is defined, a null pattern is returned.

2 or more patterns may share the same PatternId. These patterns function completely independent of each other.

2 or more patterns cannot have identical Rank{{`Type, Rank}}`Value, and pattern tokens. This results in undefined behavior when the patterns are candidates since they have identical rank. The classifier is free to choose any one candidate in this situation.

New pattern types and ranks can be introduced in future specifications. If a classifier encounters a definition it cannot support, it must immediately return an initialization error.

Examples

```
Pattern:
  PatternId: p1
  RankType: Strong
  PatternType: Simple
  PatternTokens: bingo, jackpot

Pattern:
  PatternId: p2
  RankType: Weak
  RankValue: 100
  PatternType: SimpleOrderedAnd
  PatternTokens: two, four, six

Pattern:
  PatternId: p3
  RankType: None
  RankValue: 1000
  PatternType: Simple
  PatternTokens: two, four, six

Token stream: one, two, three, four, five, six, seven
Pattern: p2

Token stream: one, two, three, six, five, four, seven
Pattern: p3

Token stream: one, two, three, four, five, six, bingo, seven
Pattern: p1
```

Attribute Retrieval

This step processes the result of the Pattern Matching step. The PatternId is used to look up the corresponding attribute map. The PatternId and the attribute map are returned.

Attributes can inherit other attribute maps by using a parent field. Attributes defined by the child will override matching parent attributes.

Attribute Transformers

An attribute map can contain attributes values which are parsed out of the input string. This is done by configuring the attribute as a set of transformers. The attribute can also have a default value if the transformers return an error.

Notes

If no attribute map is found, an empty map is used. The patternId must always be returned with the attributeMap. patternId is a reserved attribute name for this purpose.

If a null pattern is returned from the previous step, this must be properly returned to the user. A null pattern must be discernible from a user defined pattern.

Transformers

Transformers accept a string, apply an action, and then return a string. If multiple transformers are defined in a set, the outputs and inputs are linked together. Transformers are used in the input parsing phase and the attribute retrieval phase.

Transformers can cause errors. Errors in input parsing are fatal, input parsing is immediately stopped and an error is returned to the user. Errors in attribute retrieval are okay. The error is written to [attribute]_error and the attribute is set to the default value, if configured, or a blank value. [attribute] _error is a reserved attribute name.

The following transformer functions are supported:

Lowercase::

- *

${renderedContent}

- *

${renderedContent}

Uppercase::

- *

${renderedContent}

- *

${renderedContent}

ReplaceFirst::

- *

${renderedContent}

- *

${renderedContent}

- *

${renderedContent}

- *

${renderedContent}

ReplaceAll::

- *

${renderedContent}

- *

${renderedContent}

- *

${renderedContent}

- *

${renderedContent}

Substring::

- *

${renderedContent}

- *

${renderedContent}

- *

${renderedContent}

- *

${renderedContent}

- *

${renderedContent}

SplitAndGet::

- *

${renderedContent}

- *

${renderedContent}

- *

${renderedContent}

- *

${renderedContent}

- *

${renderedContent}

IsNumber::

- *

${renderedContent}

- *

${renderedContent}

- *

${renderedContent}

Notes

New transformers can be introduced in future specifications. If a classifier encounters a definition it cannot support, it must immediately return an initialization error.

Examples

```
Input string: 'aaa bbb 123 ccc'

Transformers:

SplitAndGet(delimiter: 'ccc', get: 0)
Result: 'aaa bbb 123 '

SplitAndGet(delimiter: ' ', get: -1)
Result: '123'

IsNumber()
Result: '123'
```

Patch Files

The pattern and attribute files can be patched with a user created pattern and attribute file. In this case, parsing configurations override, the pattern sets get appended (you can override using pattern ranking), and attributes override using the PatternId.

Patch files must be passed in during startup initialization. After startup, a domain cannot be patched.

Test Suites

Test suites can be configured to test input against a client on a specific domain.

Test suite attributes:

Domain::

- *

  ${renderedContent}

- *

  ${renderedContent}

DomainVersion::

- *

  ${renderedContent}

- *

  ${renderedContent}

Tests::

- *

  ${renderedContent}

- *

  ${renderedContent}

Test attributes:

Input::

- *

  ${renderedContent}

- *

  ${renderedContent}

ResultPatternId::

- *

  ${renderedContent}

- *

  ${renderedContent}

ResultAttributes::

- *

  ${renderedContent}

- *

  ${renderedContent}

When a client runs a test suite, it needs to load the associated domain and then iterate thru each test, running the input string against the domain and then checking the resulting PatternId and attributes against the expected results.

A client must report several metrics on completion of the test suite:

- Number of tests completed (attempted)
- Number of tests failed
- Domain loadtime in ms (including JSON parsing)
- Tests runtime in ms (excluding JSON parsing)
- [Optional] Amount of memory required to run the test suite in bytes

A client passes a test suite if all tests are completed and there are no failed tests.

A client must also show failure when:

- A domain with a specVersion greater than what is supported is used
- A domain, type, or domainVersion mismatch between files
- An unsupported Pattern{{`Type or Rank}}`Type is used
- An unsupported transformer is used
- Failed tests

The DeviceMap project will provide reference domains with test suites and a reference client. All clients must pass the reference test suites to be considered a valid client. All domains must provide a suitable test suite which passes on the reference client to be considered a valid domain. Therefor, any valid client can run any valid domain.

Client

Clients must do the following:

- Allow domains to be passed in and initialized.
- Allow user text to be classified against the initialized domain and return the proper result.
- Pass all of the reference domain test suites.
- Fail as described in the specification (see Test Suites).

Note that a client will only be required to initialize a domain (and possible patches) at startup time. After a client has initialized itself, the client is allowed to put the configured domain in a read only mode and reject anything which would change the domain functionality.

Also, detecting errors outside of the required failure states is at the discretion of the client. A client can assume that only well formed domains which pass on the reference client will be used.

A client must also:

- Be written using the best practices, idioms, and patterns of the given implementation language.
- Achieve the highest levels of performance while maintaining correctness.
- Have a maintainer who can support the client.
- Be released using the proper release channels.
- Include detailed documentation and examples for usage and integration.
- Include any relevant domains.

Clients can be contributed to the DeviceMap project. They will be reviewed, tested, and then voted on. There will only be 1 official client per language /platform. If a client does not exist for a given language/platform, then contributing one is highly encouraged. If a client exists, then bugs, improvements, features, and/or rewrites should be directed to said client.

Format

The pattern, attribute, and test files are JSON objects. All files will contain:

- Specification version
- Type (pattern, attribute, test)
- Domain name
- Domain version
- Description
- Publish date

Pattern File

```
{
  "specVersion": 2.0,
  "type": "pattern|patternPatch",
  "domain": "example",
  "domainVersion": "1.0",
  "description": "this is an example pattern file",
  "publishDate": "2015-06-19T09:17:37Z",
  "inputParser": { InputParser },
  "patternSet": { PatternSet },
  "attributes": [ { Attribute }, { Attribute }, ... ]
}
```

Note that defining attributes in a pattern file is optional. Attributes defined in the actual attribute file will override attributes defined in the pattern file.

InputParser

```
{
  "transformers": [ { Transformer }, { Transformer }, ... ],
  "tokenSeperators": [ "1", "two", "3" ],
  "ngramConcatSize": 2
}
```

PatternSet

```
{
  "defaultId": "someid",
  "simpleHashCount": 1400,
  "patterns" [ { Pattern }, { Pattern }, ... ]
}
```

Note that Simple{{`Hash}}`Count must be defined before Patterns to facilitate efficient JSON stream parsing.

Pattern

```
{
  "patternId": "someid",
  "rankType": "Strong|Weak|None",
  "rankValue": 150,
  "patternType": "SimpleOrderedAnd|SimpleAnd|Simple",
  "patternTokens" [ "token1", "token2", "token3", ... ]
}
```

Transformer

```
{
  "type": "UpperCase|LowerCase|IsNumber|ReplaceAll|...",
  "parameters":
  {
    "param1": "value1",
    "param2": "value2",
    ...
  }
}
```

Attribute File

```
{
  "specVersion": 2.0,
  "type": "attribute|attributePatch",
  "domain": "example",
  "domainVersion": "1.0",
  "description": "this is an example attribute file",
  "publishDate": "2015-06-19T09:17:37Z",
  "attributes": [ { Attribute }, { Attribute }, ... ]
}
```

Attribute

```
{
  "patternId": "somePatternId",
  "parentId": "somePatternId",
  "attributes":
  {
    "attribute1": "value1",
    "attribute2": "value2",
    ...
  },
  "attributeTransformers":
  {
    "transformedAttribute1": { TransformedAttribute },
    "transformedAttribute2": { TransformedAttribute },
    ...
  }
}
```

TransformedAttribute

```
{
  "defaultValue": "some default value",
  "transformers": [ { Transformer }, { Transformer }, ... ]
}
```

Test File

```
{
  "specVersion": 2.0,
  "type": "test",
  "domain": "example",
  "domainVersion": "1.0",
  "description": "this is an example test file",
  "publishDate": "2015-06-19T09:17:37Z",
  "tests": [ { Test }, { Test }, ... ]
}
```

Test

```
{
  "input": "some input string",
  "resultPatternId": "somePatternId",
  "resultAttributes":
  {
    "attribute1": "value1",
    "attribute2": "value2",
    ...
  }
}
```