Whistle Data Transformation Language language reference

Background

The Whistle Data Transformation Language expresses data mappings from one schema to another. It lets users transform complex, nested data formats into other complex and nested data formats. This document lists out the features available in Whistle.

Running your mappings

- Run your mapping using the mapping*engine binary, in mapping*engine/main. An example command might look like (run from mapping_engine/main):
  `go run . -input_file_spec=<PATH_TO_INPUT_FILE_OR_DIR> -output_dir=<PATH_TO_INPUT_DIR> -mapping_file_spec=<PATH_TO_MAPPING_FILE>`

All available flags

- input*file*spec: Input data file or directory (JSON)
- output_dir: Path to the directory where the output will be written to. Leave empty to print to stdout
- mapping*file*spec: Mapping file
- lib*dir*spec: Path to the directory where the library mapping files are
- harmonize*code*dir_spec: FHIR ConceptMaps used for code harmonization
- harmonize*unit*spec: Unit harmonization config (textproto)
- data*harmonization*config*file*spec: Data harmonization config (textproto).

Mapping

Mapping is the most basic operation of the mapping engine. There are several different types of mappings that can be expressed.

Field

You can map a field from the input JSON to a field in the output JSON. The output JSON field is on the left, seperated by a : and the input field is on the right.

E.g.

```
patient_name: input_json.name;
```

Any nested fields can be written to directly, without needing to explicitly create the JSON structure.

E.g.

```
patient.name: input_json.name;
```

> NOTE: By default, all fields are present in the output. For fields that should not be present in the output, use variables.

A field can contain spaces, dots or any special characters by escaping them with a `\` (or two backslashes for a literal backslash in the field).

A field can be composed of any characters (e.x. unicode), or use reserved Whistle keywords (like "var" or "required"), if it is quoted in single quotes. Beware that all delimiters inside the quotes are preserved literally and are not treated as delimiters in the path.

```
patient\.data.first\\name: input_json.first\ name
'patient.data'.'😊': true
keyword.'var': "keywords must be quoted"
```

outputs:

```
{
  "patient.data": {
    "first\\name": "...",
    "😊": true
  },
  "keyword": {
    "var": "keywords must be quoted"
  }
}
```

NOTE: In the above example, the double `\\` is the JSON escape for a single `\`. This means the above first name field, when parsed out of JSON, is `first\name`.

NOTE: Unicode characters are not currently escaped in the output JSON.

Constant

You can map various different types of constants to the output fields.

String constants

Directly map constant strings.

```
patient_name: "John Doe";
```

Numeric constants

Directly map constant numerics.

```
patient_age_years: 25;
patient_age: 25.5;
```

Boolean constants

Directly map constant booleans.

```
patient_deceased: false;
patient_not_deceased: true;
```

Variable

Use variables to map to temporary fields that are not desired in the output.

```
// Writing to a variable.
var input_name: input_json.name

// Reading from a variable.
patient_name: input_name;
```

> NOTE: Variables take precedence over input fields if they have the same name.

Nested mappings

Nested fields can also be described with blocks:

```
patient: {
  name: input_json.name
}
```

Variables inside blocks are scoped to that block for writes. E.g.,

```
var one: "one"
nested: {
  read: one
  var one: "two"
  // The above assignment declares a new variable called "one" for this block.
  inner: one
}
// The below "one" refers to the variable declared on the first line.
outter: one
```

outputs:

```
{
  "nested": {
    "read": "one",
    "inner": "two"
  },
  "outter": "one"
}
```

Functions

Mappings can be grouped together into a function, and this function can be called by other mappings.

> NOTE: All mappings defined inside of a function are present in the function's output object but not the mapping's output object. Use root or out for outputting an object to the mapping's output object.

Defining a function

Functions are defined by the `def` keyword followed by the function name and the list of parameters the function takes.

```
def PatientName(input) {
    patient_name: input.name
}
```

Arguments can have optional `required` keywords. Mappings in a function will only be executed if **all** arguments with `required` keyword are not nil. For example:

```
def PatientName(required requiredArg1, required requiredArg2, arg3) {
    ...mappings...
}
```

is equivalent to

```
def PatientName(requiredArg1, requiredArg2, arg3) {
    if $IsNotNil(requiredArg1) and $IsNotNil(requiredArg2) {
        ...mappings...
    }
}
```

Calling a function

Calling a function is similar to how you call functions in other programming languages (e.g. C, Java, Python).

```
patient: PatientName(input_json)
```

Values passed to functions can also be objects, as described above, E.g.

```
patient: PatientName({
  first: "Jane"
  var initial: "J. "
  last:  $ToUpper($StrCat(initial, "Doe"))
})

// Or, equivalently
patient: PatientName({first: "Jane"; last:  "Doe";})
```

Note that variables are not passed along to `PatientName`.

Builtin functions

There are a number of builtin functions provided out of the box. Builtin functions are prefixed with a `$`. The full documentation of the builtin functions is available here.

Null propagation

By default, null and missing values/fields are ignored in accordance with the following rules:

- If a field is written with a null or empty value, it is ignored (thus `null`, `{}`, and `[]` can never show up in the mapping output)
- If a non-existent field is accessed, it returns `null`

> NOTE: Functions are still executed even if its arguments are null.

Merge semantics

Assigning a value to the same field results in a merge rather than an overwrite, i.e:

- Arrays are concatenated
  - Even though both functions mapped to `colours[0]`, `"blue"` ended up in `colours[1]`
- New fields are added
- Similar fields produce a merge conflict. An overwrite can be forced (see overwrite operator `:!`)

Conditions

Mappings can be conditionally executed.

Conditional Mapping

Conditional mapping allows a single mapping to be executed if and only if a condition is true. Add the condition right before the `:`.

```
patient (if input_json.type = "Patient"): PatientName(input_json)
```

Conditional Blocks

Conditional blocks allow wrapping a set of mappings within a condition. An optional `else` allows executing mappings if the condition is not met.

```
def PatientName(input) {
  if input.type = "first" {
    first: input.value;
  } else {
    last: input.value;
  }
}
```

Arrays

Arrays

Iteration (`[]`)

To iterate an array, suffix the array with `[]`. For example:

- `Function(a[])` means "pass each element of `a` (one at a time) to `Function`"
- `Function(a[], b)` means "pass each element of `a` (one at a time), along with `b` to `Function`". If `b` is an array of the same length as `a` we can iterate them together
- `Function(a[], b[])` means "pass each element of `a` (one at a time), along with each element of `b` (at the same index) to `Function`"
    - An error is returned if `a` and `b` are of different lengths
- `[]` is also allowed after function calls
    - `Function2(Function(a)[])` means "pass each element from the result of `Function(a)` (one at a time) to `Function2`
- The result of an iterating function call is also an array

Appending (`[]`)

Suffixing a target with `[]` appends to the array. The array index can also be used to write to a specific index.

- Append to an array using `[]`
- `[]` in the middle of the path (e.g. types[].typeName: ...) is valid as well and creates `types: [{"typeName": ... }]`
- Hardcoded indexes can also be used (e.g. `types[0]: ...` and `types[1]: ...`)
- "Out of bounds" indexes (e.g. `types[153]: ...` generates all the missing elements as `null`

Wildcards (`[*]`)

In contrast to iteration, the `[*]` syntax works like specifying an index, except that it returns an array of values. Multiple arrays mapped through with `[*]`, for example `a[*].b.c[*].d`, in one long, non-nested array of the values of `d` with the same item order.

> NOTE: null values are included, through jagged traversal. E.g.: `a[*].b.c[*].d`, if some instance of a does not have b.c, then a single null value is returned for that instance.

Filtering (`[where ...]`)

Filters allow narrowing an array to items that match a condition using the following syntax:

- The `where` keyword indicates a filter, similar to `if` indicating a condition
- Each item from the array is loaded one at a time into a variable named `$` in the filter
    - all fields of `$` are accessable
- The filter produces a new array. To iterate over the results, use the `[]` operator
- Filters can only be the last element in a path, i.e. `a.b[where $.color = "red"].c` is invalid

```
// Set males to all patients whose gender = "MALE".
males: patients[where $.gender = "MALE"];
```

Post Processing (`post`)

Post processing allows running a function after the mapping is complete. The input to the post processing function is the output from the mapping such that the result becomes the new output. A `post` function must be the last thing in the file.

Inline:

```
post UpperCase(output) {
  name[] : $ToUpper(output[*].name[])
}
```

Existing functions:

```
def UpperCase(arr) {
  name[] : $ToUpper(arr[*].name[])
}

post UpperCase
```

Other Keywords

Whistle has various constructs to allow mapping from one JSON structure to another.

$root

`$root` is used to denote the input JSON object. `$root` can also be used inside functions, but is not recommended since it is a strong sign of messy, non-modular mappings.

root

`root` can be used inside a function in order to send data to the root of the output. If there exists an object with the same name, they are merged.

```
root name: input.patient.name
```

$this

`$this` is used to set the current object as the return value instead of creating a new object to return.

```
first_name : FirstName($root.name)

def FirstName(name) {
  $this: $StrSplit(name, " ")[0]
}
```

out

`out` is used to append output an object to the main output object instead of only as the return value of a mapping. This is different from `root` which sets data at the root of the output instead of appending. `out` can also be used inside a function.

> NOTE: The mapping engine returns an error if the field written to by `out` is not an array.

dest

`dest` is used to read data from the current function's output object instead of from the input.

Operators

There are built in arithmetic, logical and existential operators.

Arithmetic (`+` , `-` , `*` , `/`)

- num + num : Addition
- num - num : Subtraction
- num * num : Multiplication
- num / num : Division

Logical (`~` , `and` , `or`)

- bool and bool : Logical AND
- bool or bool : Logical OR
- ~bool : Logical NOT

Equality

- any = any : Equal
- any ~= any : Not Equal

Existential (`?`)

- any? : Value exists or array/string is not empty
- ~any? : Value does not exist or array/string is empty

Overwrite (`!`)

In order to prevent data loss and reduce mapping errors, Whistle allows a primitive (string, numeric, or boolean) field to only be written once. The `!` operator can be used to overwrite primitive fields.

> NOTE: Overwriting restrictions do not apply to variables.

Code Harmonization

Code Harmonization is the mechanism for mapping a code in one terminology to another. The mapping engine uses FHIR ConceptMaps to store lookup tables, and uses a subset of FHIR Translate mechanics to do code lookups. Unlike FHIR translate, lookups return an array of FHIR Codings on successful responses.

The lookups can be configured to read from remote servers as well as local files. Local files are read at startup, and cached for the duration of the mapping engine instance. When reading from remote servers, the returned codes are cached. The Time To Live (TTL) and cache clean up interval for this cache is configurable.

For unsuccessful responses, an error message is returned instead of an an array of FHIR Codings. In the case of remote servers, the http code and error message is returned.

Configuration

See the CodeHarmonization protobuf for more information on how to configure Code Harmonization.

Local code harmonization example

ConceptMap:

```
{
  "group":[
    {
      "element":[
        {
          "code": "red",
          "target":[
            {
              "code": "target-red",
              "equivalence": "EQUIVALENT"
            }
          ]
        },
        {
          "code": "blue",
          "target":[
            {
              "code": "target-blue",
              "equivalence": "EQUIVALENT"
            }
          ]
        }
      ],
      "source": "codelab-source",
      "target": "codelab-target"
    }
  ],
  "id": "codelab-conceptmap-id",
  "version": "v1",
  "resourceType":"ConceptMap"
}
```

Local file configuration:

```
code_lookup: {
  local_path: PATH_TO_CONCEPT_MAP
}
```

Google Cloud Storage (GCS) file configuration:

```
code_lookup: {
  gcs_path: gs://PATH_TO_CONCEPT_MAP
}
```

> NOTE: Applicaton default credentials are used when accessing GCS files.

Remote code harmonization example

> NOTE: Only Google Cloud FHIR Stores are supported currently.

Configuration:

```
code_lookup: {
  name: "fhir-store",
  url_path: https://healthcare.googleapis.com/PATH_TO_FHIR_STORE
},
cache_ttl_seconds: 100,
cleanup_interval_seconds: 500
```

Lookup syntax

$HarmonizeCode

```
$HarmonizeCode(lookupSourceName string, sourceCode string, sourceSystem string, conceptMapID string) array
```

Harmonize the provided code to the value specified by the ConceptMap.

Arguments:

- lookupSourceName: The name of the code lookup block in the CodeHarmonzation configuration if remote, or `$Local` for local concept maps.
- sourceCode: The code to lookup.
- sourceSystem: The system that the source code is in.
- conceptMapID: The ID of the ConceptMap to lookup against.

Return: An array of FHIR Codings that match.

$HarmonizeCodeBySearch

```
$HarmonizeCodeBySearch(lookupSourceName string, sourceCode string, sourceSystem string) array
```

Harmonize the provided code to the value specified by any ConceptMap. This is slower than `$HarmonizeCode` as it needs to look at all the provided ConceptMaps.

Arguments:

- lookupSourceName: The name of the code lookup block in the CodeHarmonzation configuration if remote, or `$Local` for local concept maps.
- sourceCode: The code to lookup.
- sourceSystem: The system that the source code is in.

Return: An array of FHIR Codings that match.

Unit Harmonization

Unit harmonization is the mechanism for converting a value in one unit to another. The mapping engine uses conversion tables defined in the UnitConfiguration syntax. Conversions return a object that contains:

- `originalQuantity` : the original quantity
- `originalUnit` : the original unit
- `quantity` : the converted quantity
- `unit` : the converted unit
- `system` : the unit config system
- `version` : the unit config version

Sample configurations

Conversion configuration:

```
version: "v1"
system: "http://unitsofmeasure.org"
conversion {
  source_unit: "LB"
  dest_unit: "KG"
  code: "weight"
  codesystem: "metric"
  constant: 0.0
  scalar: 0.453592
}
conversion {
  source_unit: "G"
  dest_unit: "KG"
  code: "weight"
  codesystem: "metric"
  constant: 0.0
  scalar: 0.001
}
```

Code harmonization configuration:

```
unit_conversion: {
  local_path: "PATH_TO_CONVERSION_CONFIG"
}
```

Lookup syntax

$HarmonizeUnit

```
$HarmonizeUnit(sourceValue string, sourceUnit string, sourceCodingSystemArray array) object
```

Harmonize the provided quantity and unit to the specified coding system.

Arguments:

- sourceValue: The value of the unit to convert.
- sourceUnit: The unit to convert.
- sourceCodingSystemArray: The coding system to convert to.

Return: An object with the original quantity and converted quantity as specfied above.

E.g: `$HarmonizeUnit("50", "KG", [{"system":"metric", "code":"weight"}])`

Output:

```
{
  "Units": {
    "converted_unit": {
      "originalQuantity": 50,
      "originalUnit": "LB",
      "quantity": 22.6796,
      "system": "http://unitsofmeasure.org",
      "unit": "KG",
      "version": "v1"
    }
  }
}
```

Comments

Similar to C/Java, lines prefixed with `//` are comments and not part of the mapping execution.