Declarative Stream Mapping (DSM) Documentation

Release 1

Mehmet Fatih Ercik

Jun 13, 2023

QUICK START GUIDE:

1 Introduction 1

2 Simple Example 3

3 Features 7

4 Installation 9

5 Sample Config File 11

6 Introduction 13

7 Definitions 15

8 Format 17

9 Document Structure 19

10 Schema 21
- 10.1 DSM Object . . . 21
  - 10.1.1 params . . . 22
  - 10.1.2 transformations . . . 22
  - 10.1.3 functions . . . 23
  - 10.1.4 fragments . . . 23
  - 10.1.5 result . . . 24
  - 10.1.6 $extends . . . 25
- 10.2 Parsing Element Object . . . 26
  - 10.2.1 fieldName . . . 28
  - 10.2.2 dataType . . . 28
  - 10.2.3 dataTypeParams . . . 30
  - 10.2.4 type . . . 30
    - 10.2.4.1 std . . . 31
    - 10.2.4.2 object . . . 31
    - 10.2.4.3 array . . . 32
    - 10.2.4.4 sum . . . 33
    - 10.2.4.5 multiply . . . 34
    - 10.2.4.6 divide . . . 36
    - 10.2.4.7 join . . . 37
  - 10.2.5 typeParams . . . 38
  - 10.2.6 path and parentPath . . . 39
  - 10.2.7 fields . . . 42

10.2.7.1 Type of Value Definition . . . 42
10.2.8 filter . . . 44
10.2.9 default . . . 46
10.2.10 transformationCode . . . 47
10.2.11 function . . . 48
10.2.12 uniqueName . . . 49
10.2.13 normalize . . . 51
10.2.14 xml . . . 51
10.2.15 $ref . . . 51
10.3 Transformation Element Object . . . 54
10.4 Default Object . . . 55
10.4.1 value . . . 55
10.4.2 force . . . 55
10.4.3 atStart . . . 55
10.5 XML Object . . . 56
10.5.1 attribute . . . 56

11 Absolute Path **59**

12 Expressions and Scripting **63**
12.1 Loading Time Expression . . . 63
12.1.1 **Example** . . . 64
12.2 Parsing Time Expression . . . 65
12.2.1 all . . . 66
12.2.2 self . . . 68
12.2.3 **Example** . . . 68

13 Merge of DSM Document **71**

14 Property Assignment Order **75**

15 Indices and tables **85**

CHAPTER

ONE

INTRODUCTION

Declarative Stream Mapping(DSM) is a *stream* deserializer library that makes parsing of **XML and JSON** easy. DSM allows you to make custom parsing, filtering, transforming, aggregating, grouping on any JSON or XML document at stream time(read only once). DSM uses yaml or json for configuration definitions

If you parsing a complex, huge file and want to have high performance and low memory usage then DSM is for you.

CHAPTER

TWO

SIMPLE EXAMPLE

Lets Parse below simple JSON and XML file with DSM

File contents are taken from Swagger Petstore example. Slightly changed.

Source file

JSON

```
    "id"  1
    "name"  "Van Kedisi"
    "status"  "sold"
    "createDate"  "01/24/2019"
    "category"   "id"  1 "name"  "Cats"
    "tags"
         "id"  1 "name"  "Cute"
         "id"  2 "name"  "Popular"

    "photoUrls"   "url1" "url2"
```

XML

```
<?xml version="1.0" encoding="UTF-8" ?>

<Pet id="1">
    <name>Van Kedisi</name>
    <status>sold</status>
    <createDate>01/24/2019</createDate>
    <category>
        <id>1</id>
        <name>Cats</name>
    </category>
    <tags>
        <tag>
            <id>1</id>
            <name>Cute</name>
        </tag>
        <tag>
            <id>2</id>
            <name>Popular</name>
        </tag>
```

(continues on next page)

(continued from previous page)

```
    </tags>
    <photoUrls>
        <photoUrl>url1</photoUrl>
        <photoUrl>url2</photoUrl>
     </photoUrls>
</Pet>
```

Those are rules that we want to apply during parsing.

- exclude "photoUrls" tag.
- read only "name" field of "tags" tag.
- read only "name" field of "category" tag.
- **add new the "isPopular" field that it's value is true, if "tag.name" has "Popular" value**

DSM config file

[YAML]

```
params
    dateFormat
result


   xml
      path
   filter
   fields
       id
         dataType
         xml
            attribute
       name
       status
       createDate
       category
           path
       isPopular
            default
       tags


# this is a regex expression. works for both JSON and XML
```

Class to deserialize

[JAVA]

```
public class
        private int
        private
        private boolean
        private
        private
```

(continues on next page)

(continued from previous page)

```
        private
        private

        // getter/setter
```

Read Data

```
                    new            "dsm-config-file.yaml"
                setType             XML  create
            toObject new      "path/to/xmlFile.xml"      class    ␣
↪// read data from xml file

            setType             JSON  create
        toObject new      "path/to/jsonFile.json"      class    ␣
↪// read data from json file
```

CHAPTER

THREE

FEATURES

- **Work** for both **XML** and **JSON**
- **Custom stream parsing**
- **Filtering** by value on any field with very **low cognitive complexity**
- Flexible value **transformation**.
- **Default value assignment**
- Custom **function calling** during parsing
- **Powerful Scripting**(Apache JEXL, Groovy, Javascript and other jsr223 implementations are supported)
- **Multiple inheritance** between DSM config file (DSM file can **extends to another config file**)
- **Reusable fragments support**
- Very **short learning curve**
- **Memory** and **CPU** efficient
- **Partial data extraction** from JSON or XML
- **String manipulation** with expression

CHAPTER

FOUR

INSTALLATION

Maven

Jackson

```
<dependency>
  <groupId>com.github.mfatihercik</groupId>
  <artifactId>dsm</artifactId>
  <version>1.0.4</version>
</dependency>
```

Gradle

Jackson

```
compile ('com.github.mfatihercik:dsm:1.0.4')
```

CHAPTER

FIVE

SAMPLE CONFIG FILE

Detailed documentation and all option is here.

This config file contains some possible option and their short description.

[header.yaml]

```
params
    dateFormat                              # define date format for "date" data type
transformations
    SOLD_STATUS
  # value transformation for "isAvailable" property
      map
         sold
         pending
         available
         DEFAULT
    SOLD_STATUS_SKIP
       $ref                                   # extends to "SOLD_STATUS" transformation.
       map
          DEFAULT                           # exclude default value
       onlyIfExist
  # make transformation only source value exist in transformation map other wise return as it is
functions
    insertPet
  # declare a function to declare at Parsing Element.

fragments                                   # create reusable fragment
    category


         id
         name
         type
```

[main.yaml]

```
$extends                                    # extends to header.yaml config.
result
                                        # result is an array
    path
  # start reading form beginning for json. path is a regex. we can define both for xml and json same ti
    xml
```

(continues on next page)

(continued from previous page)

```
    path                                # start reading from /Pets/Pet for xml

  filter                                ␣
↪# filter by "isAvailable" property. "self" key word refers to current Node. self.parent refers to par

  function                              ␣
↪# call "insertPet" function for every element of "result" array
  fields
    name                                # read name as string.
    id
      dataType                              # read id as int
      xml
        attribute                       # id is an attribute on /Pets/Pet tag.
    createDate                          ␣
↪# use dateFormat in params then convert string to date
    isAvailable
        path                            # read isAvailable as string from "status" tag
        dataType
        transformationCode              ␣
↪# user "SOLD_STATUS" transformation to map from "status" to "isAvailable"
    category
        $ref                                # extends to "fragment.category"
        fields
           type                         ␣
↪# exclude "type" field from "category" fragment
           name
               default  'Animal'        ␣
↪#set default value to 'Animal' if "category/name" tag not exist in source document
    isPopular
          default                                   ␣
↪# set default value of "isPopular" property

    tags


        filter                          # filter by length of value.
        xml
          path
```

CHAPTER

SIX

INTRODUCTION

Declarative Stream Mapping(DSM) is a stream deserializer library that works for both XML and JSON. DSM allows you to make custom parsing, filtering, transforming, aggregating, grouping on any JSON or XML document at stream time(read once). There is no need to writing custom parser. DSM use yaml or json configuration file to parse data. Processed data can be deserialized to java classes.

CHAPTER

SEVEN

DEFINITIONS

DSM Document:
A document (or set of documents) that defines or describes parsing element definition uses and conforms to the DSM Specification.

Source Document:
Document(File, Stream, String, HTTP Request Payload) contains JSON or XML data.

CHAPTER

EIGHT

FORMAT

DSM document is a JSON object, which maybe represented either in JSON or YAML format. All field names in the specification are case sensitive. This includes all fields that are used as keys in a map, except where explicitly noted that keys are case insensitive.

CHAPTER

NINE

DOCUMENT STRUCTURE

DSM document may be made up single document or divided into multiple connected parts at the discretion of the user. In later case, *$extends* fields must be used to reference those parts.

CHAPTER

TEN

SCHEMA

In the following description, if a field is not explicitly REQUIRED or described with a MUST or SHALL, it can be considered OPTIONAL.

10.1 DSM Object

This is the root document object of the DSM document.

Fields:

Field Name	Type	Description
version	string	**REQUIRED**. This string MUST be the semantic version number of the DSM Specification version that the DSM document uses. The DSM field SHOULD be used by tooling specifications and clients to interpret the DSM document
params	Map[string,any]	**params**_ field is a map that contains parameter definition to configure DSM document and `Parsing Element`_s.
transformations	Map[string,`Transformation Element`_]	Declaration of Map contains *transformationCode* as key, and *Transformation Element* as value. *Transformation Element* holds lookup table to transform value from *Source Document* to destination document.
functions	Map[string,Function_]	Declaration of Map contains *function* name as key, and *function deceleration* as value. functions are used for custom parsing or calling services with parsed data. Functions implements FunctionExecutor interface.
fragments	Map [String, *Parsing Element*]	A map contains declaration of reusable *Parsing Element*. The fragment definition can be referenced with $ref_ field while defining *Parsing Element*.
result	*Parsing Element*	**REQUIRED**. The entry point of *Parsing Element* declarations. The result field defines structure of the output.
$extends	string	`**$extends**`_ field is used for import external DSM document.

10.1.1 params

params field is a map that contains parameter definition to configure DSM document and *Parsing Element*. The key of params map is string and case sensitive. The value of params map can be any type of json object(int, boolean, object, array) accepted by JSON and YAML specification.

Example DSM document that contains params.

YAML

```
version
params
    dateFormat
    rootPath
    category
        foo
    acceptedCountryCode    TR US FR
```

JSON

```
    "version"  1.0
    "params"
          "dateFormat" "dd.MM.yyyy"
          "rootPath" "fooBar/foo"
          "category"
            "foo" "bar"

          "acceptedCountryCode"    "TR" "US" "FR"
```

10.1.2 transformations

Deceleration of Map contains *transformationCode* as key, and *Transformation Element* as value. *Transformation Element* holds lookup table to transform value from *Source Document* to destination document.

Example CF document that contains transformations

YAML

```
version
transformations
    COUNTRY_CODE_TO_NAME
        map
          DEFAULT
          TR
          US
```

JSON

```
"version"  1.0
"transformations"
   "COUNTRY_CODE_TO_NAME"
    "map"
      "TR" "Turkey"
      "US" "United States"
      "DEFAULT" "Other"
```

10.1.3 functions

Deceleration of Map contains *function* name as key, and *function deceleration* as value. functions are used for custom parsing or calling services with parsed data. Functions implements FunctionExecutor interface.

Example CF document that contains functions

YAML

```
version
functions
   insertProduct
   approveOrder
```

JSON

```
"version"  1.0
"functions"
   "insertProduct" "com.example.InsertProduct"
   "approveOrder" "com.example.ApproveOrder"
```

10.1.4 fragments

A map contains declaration of reusable *Parsing Element*. The fragment definition can be referenced with *$ref* field while defining *Parsing Element*.

Example CF document that contains functions

YAML

```
version
fragments
   product
     fields
        id
        name
        price
```

JSON

```
{
   "version"  1.0
   "fragments"
      "product"
         "fields"
            "id" "string"
            "name" "double"
            "price" "double"
}
```

10.1.5 result

REQUIRED. The entry point of *Parsing Element* declarations. The result field defines structure of the output.

Example CF document that contains result

YAML

```
version
result
  type
  path
  fields
     id
     name
     price
```

JSON

```
{
  "version"  1.0
  "result"
     "type" "object"
     "path" "/"
     "fields"
        "id" "string"
```

(continues on next page)

(continued from previous page)

```
        "name" "double"
        "price" "double"
```

10.1.6 $extends

$extends field is used for import external DSM document to current DSM document. it's value is basically relative path or URI definition of external DSM document. if it's value start with "$" sing, it is accepted as an *expression* and resolved by expression resolver. External DSM document will *merged* into current DSM document. $extends can also be list of path or URI. *Merge* process start from first element to last element. Firstly current DSM document *merged* with first element then result of *merge* process extended to second element etc..

Example CF document that contains extends

YAML

```
version
$extends
```

or

```
version
params
   rootPath
$extends
```

JSON

```
   "version"  1.0
   "$extends"  "/foo/bar/external.json"
```

or

```
   "version"  1.0
   "params"
   "rootPath" "/bar/foo/"

   "$extends"   "/foo/bar/external.json" "$params.rootPath.concat(
↪'externalWithExpression.json')"
```

10.2 Parsing Element Object

Parsing Element is basic object of DSM. Parsing Element contains set of rules for parsing specific tag of *Source Document*

Fields:

Field Name	Type	Description
fieldName	string	**REQUIRED** *fieldName* define the name of the property to expose by current object. the fieldName is unique in object.
dataType	string	**REQUIRED**. The data type of exposed field. it may have extra parameter provided with *dataTypeParams*
dataTypeParams	Map[string,any]	extra parameters for dataType field need to convert. for example. dateFormat for dataType
type	string	the name of the parsing strategy for the current field. the default is STD.
typeParams	Map[string,any]	it is used for passing extra parameter to *dataType* converter. *typeParams* field *extended* to *params* field.
path	string	The path field specifies the location of a tag in the *source document* relative to the path field of the higher-level Parsing Element definition. The default value is the value in the fieldName field.
parentPath	string	The *parentPath* field is used in a slightly more complex parsing definitions. it holds path to parent tag of the tag specified in the "*path*" field.
default	string,`Default Object`_	default value of the field if *path* not exist in the source document. if default value starts with "$" character it is accepted as *expression* and it is resolved by expression resolver.
filter	string	The Filter field determines whether the value of a *Parsing Element Object* (complex or simple *type* does not matter) is added to the object tree. The filter field is an *expression* that returns true or false.
transformationCode	string	this field refers to the definition of the **transformation_** to be used to transform the tag value.
function	string	name of the function in *functions* map.
normalize	string	this field is used to normalize the value of **tag_**. It is an expression.
uniqueName	string	When "fieldName" fields of complex *Parsing Element* definitions are the same in the DSM document, these definitions are differentiated by using the "uniqueKey" field.
xml	Map[string,any]	XML related configuration goes under this tag.
attribute	boolean	it is indicates that the current *Parsing Element Object* is an attribute on the tag pointed to by the parentPath field in the xml.
overwriteByDefault_	boolean	force using *default* field. Mostly used with filter field.
fields	Map[string,String - Parsing Element Object - [Parsing Element Object]]	fields of the current object. its only valid for object and array *type*
$ref	string	$ref_ field is used to *extends* current config to given **fragment_**. it's value is an expression.

10.2.1 fieldName

REQUIRED The *fieldName* define the name of the property to expose by current object. the fieldName is unique in object.. However, a fieldName may have multiple *Parsing Element*. The fieldName is not explicitly defined. it is defined with *fields* property. The keys of *fields* map are the fieldName of the Parsing Element.

In blow DSM document, id, name, and price are *fieldName* of the result object. The *result* object exposes id, name and price property

YAML

```yaml
result                # fieldName is result
version


                      # fieldName is id
  fields
     price
     name             # fieldName is name
```

JSON

```json
  "version"  1.0
  "result"
     "type" "object"
     "path" "/"
     "fields"
        "id" "string"
        "name" "double"
        "price" "double"
```

10.2.2 dataType

The dataType field defines data type (string, int, boolean etc.) of the exposed property. it is basicity a converter from string to given dataType type. it may need extra parameters to convert a string to given data dataType. Extra parameters may be provided with *params* or *dataTypeParams*.

Supported dataType name and their corresponding java class:

Type Name	Java Type	Extra Parameters
int	int	
float	float	
short	short	
double	double	
long	long	
date	date	dateFormat(required)
boolean	boolean	
char	char	
BigDecimal	BigDecimal	
BigInteger	BigInteger	

YAML

```
version
params
  dateFormat              # default 'dateFormat' for all 'date' dataType
result


  fields
     id
     name                 # implicitly defined dataType. dataType is string
     price
        dataType             # explicitly defined dataType dataType. type is double
     createDate
# implicitly defined the date dataType. and 'dateFormat' is defined in params field.
     modifiedTime
       dataType
# explicitly defined the date dataType. and 'dateFormat' is defined in dataTypeParams field.
       dataTypeParams
          dateFormat  "yyyy-MM-dd'T'HH:mm:ss'Z'"
```

JSON

```
  "version"  1.0
  "params"
     "dateFormat"  "dd.MM.yyyy"

  "result"
     "type" "object"
     "path" "/"
     "fields"
        "id" "string"
        "name" "double"
        "price"
           "dataType"  "double"

        "createDate"  "date"
        "modifiedTime"
           "dataType"  "date"
```

(continues on next page)

(continued from previous page)

```
            "dataTypeParams"
               "dateFormat" "yyyy-MM-dd'T'HH:mm:ss'Z'"
```

10.2.3 dataTypeParams

dataTypeParams is used for passing extra parameters to a *dataType* convert. dataTypeParams field *extended* to *params* field.

Check *example* here.

10.2.4 type

type defines how tags in the source document are parsed. it also defines the structure of the output object and hierarchy of the object tree. it may need extra parameters. Extra parameters are provided with *typeParams* field that *extended* to *params* field. Basically, there are two main "type" categories which are "complex", and "simple". The complex category includes "tagTypes" which exposes complex data dataType such as object or arrays. the simple category includes tagTypes which expose data type in the *dataType* field.

Supported *type*'s:

Type Name	Category	Java Type	Extra Parameters
std	Simple	*dataType*	default *type* if not defined explicitly
object	Complex	Map	
array	Complex	List<Map> - List<*dataType*>	
sum	Simple	*dataType*	fields: array of fieldName of current object.
multiply	Simple	*dataType*	**fields**: array of fieldName of current object.
divide	Simple	*dataType*	**fields**: array of fieldName of current object.
join	Simple	*dataType*	**fields**: array of fieldName of current object, **separator**: separator string. default is comma(,)

10.2.4.1 std

std is basic *type* which copy the value of the tag in the source document to the current object. std is the default value of the *type* field. Data *dataType* is defined in **dataType_ype_** field.

YAML

```
version
result
  type
  path
  fields
     foo          # tag type is STD
     bar          # tag type is STD
```

JSON

```
  "version"  1.0
  "result"
     "type" "object"
     "path" "/"
     "fields"
        "foo" "string"
        "bar" "int"
```

10.2.4.2 object

object *type* is used to expose an object. *Parsing Element* which has "object" *type* must have '*fields*' field.

YAML

```
version
result
  type
  path
  fields
     id
     name
     price
```

JSON

```
  "version"  1.0
  "result"
     "type" "object"
```

(continues on next page)

(continued from previous page)

```
    "path" "/"
    "fields"
      "id" "string"
      "name" "string"
      "price" "double"
```

Above DSM document generate following output(values are only example) :

```
  "id" "11111"
  "name" "foo"
  "price" 1111.111
```

10.2.4.3 array

array *type* is used to expose an array. Items of the array may be a object or simple *dataType*. if *Parsing Element* has "*fields*" field then the array *type* exposes List<Object>. if *fields* field is not defined, the data type of array item decided *dataType* field.

YAML

```
version
result
  type          # EXPOSE [Object] array of Object
  path
  fields
     id
     name
     price
     tags                  # EXPOSE [string] array of string
       type
       path
       type
```

JSON

```
  "version"  1.0
  "result"
     "type" "object"
     "path" "/"
     "fields"
        "id" "string"
        "name" "string"
        "price" "double"
        "tags"
```

(continues on next page)

(continued from previous page)

```
            "type"  "array"
            "path"  "tag"
            "type"  "string"
```

Above DSM document generate following output(values are only example) :

```
  "id" "11111"
  "name" "foo"
  "price" 1111.111
  "tags"  "foo" "bar"
```

10.2.4.4 sum

sum type is used to sum properties defined with "fields" in **typeParams**_. if one of the properties that defined in *fields* does not exist in the current object, it is accepted as ZERO.

if current property(*Parsing Element* that "*sum*" *type* is defined on) is defined in "fields" in **typeParams**_, current property value is added to total result.(sum with self)

(Explained with example below)

typeParams:

Name	Type	Description
fields	array	**REQUIRED** list of *fieldName* of the properties in parent *Parsing Element* to sum.

YAML

```
version
result
  type
  path
  fields
     foo
     bar
     fooAndBar
         path       # when current object closed
         type          # declare sum type to sum foo and bar field

         typeParams

# sum foo,and bar fields then set to fooAndBar property of current object.
     sumWithSelf
         type          # declare sum type to sum foo and bar field
```

(continues on next page)

(continued from previous page)

```
        typeParams
↪# sum foo, bar and sumWithSelf(current field) fields then set sumWithSelf to total property of curren
```

JSON

```
"version"  1.0
"result"
   "type" "object"
   "path" "/"
   "fields"
      "foo" "int"
      "bar" "int"
      "fooAndBar"
          "path" "\."
          "type"  "sum"
          "typeParams"
              "fields"  "foo" "bar"


      "sumWithSelf"
         "type"  "sum"
         "type" "int"
         "typeParams"
            "fields"  "foo" "bar" "sumWithSelf"
```

10.2.4.5 multiply

multiply type is used to multiply properties defined with "fields" in **typeParams_**. if one of the properties that defined in *fields* does not exist in the current object, it is accepted as ONE.

if current property(*Parsing Element* that "*multiply*" *type* is defined on) is defined in "fields" in **typeParams_**, current property value is multiplied with total result. (multiply with self)

(Explained with example below)

typeParams:

Name	Type	Description
fields	array	**REQUIRED** list of field name of the properties in **current object** to multiply.

YAML

```
version
result
  type
  path
  fields
     foo
     bar
     fooAndBar
         path       # when current object closed
         type               # declare multiply type to sum foo and bar field

         typeParams

# multiply foo,and bar fields then set it to fooAndBar property of current object.
     multiplyWithSelf
         type               # declare multiply type to sum foo and bar field

         typeParams

# multiply foo, bar and multiplyWithSelf(current field) fields then set it to multiplyWithSelf proper
```

JSON

```
  "version"  1.0
  "result"
     "type" "object"
     "path" "/"
     "fields"
        "foo" "int"
        "bar" "int"
        "fooAndBar"
            "path" "\."
            "type"  "multiply"
            "typeParams"
                "fields"  "foo" "bar"


        "multiplyWithSelf"
           "type"  "multiply"
           "type" "int"
           "typeParams"
              "fields"  "foo" "bar" "multiplyWithSelf"
```

10.2.4.6 divide

---

divide type is used to divide properties defined with "fields" in **typeParams_**. if one of the properties that defined in *fields* does not exist in the current object, it is accepted as ONE.

if current property(*Parsing Element* that "*divide*" *type* is defined on) is defined in "fields" in **typeParams_**, current property value is divided with total result. (divide with self)

(Explained with example below)

typeParams:

Name	Type	Description
fields	array	**REQUIRED** list of field name of the properties in **current object** to divide.

YAML

```
version
result
  type
  path
  fields
     foo
     bar
     fooAndBar
         path       # when current object closed
         type             # declare divide type to sum foo and bar field

         typeParams

# divide foo with bar (foo/bar) fields then set it to fooAndBar property of current object.
     divideWithSelf
         type             # declare divide type to sum foo and bar field

         typeParams

# divide foo with bar then divide with divideWithSelf(current field) (foo/bar/divideWithSelf) fields
```

JSON

```
  "version"  1.0
  "result"
     "type" "object"
     "path" "/"
     "fields"
        "foo" "int"
        "bar" "int"
        "fooAndBar"
            "path" "\."
            "type"  "divide"
            "typeParams"
                 "fields"  "foo" "bar"
```

(continues on next page)

(continued from previous page)

```
        "divideWithSelf"
           "type"  "divide"
           "type" "int"
           "typeParams"
              "fields"  "foo" "bar" "divideWithSelf"
```

10.2.4.7 join

join type is used to join properties defined with "fields" in **typeParams_**. if one of the properties that defined in *fields* does not exist in the current object, it is skipped.

if current property(*Parsing Element* that "*join*" *type* is defined on) is defined in "fields" in **typeParams_**, current property value is included in to joining (join with self) Values are separated by *separator* defined in "*typeParams*". The default separator is a comma(,)

typeParams:

Name	Type	Description
fields	array	**REQUIRED** list of field name of the properties in **current object** to join.
separator	string	separator string. default is comma(i)

YAML

```
version: 1.0
result:
  type: object
  path: /
  fields:
     foo: string
     bar: string
     fooAndBar:
         path: \.  # when current object closed
         type: join    # declare join type to sum foo and bar field
         type:int
         typeParams:
            fields:[foo,bar]  # join foo and bar (foo,bar) fields then set it to␣
↪fooAndBar property of current object.
     joinWithSelf:
         type: join    # declare join type to sum foo and bar field
         type:int
         typeParams:
            separator: &
            fields:[foo,bar,sumWithSelf]  # join foo,bar, and joinWithSelf(current␣
```

(continues on next page)

(continued from previous page)

```
↪field) (foo&bar&joinWithSelf) fields then set it to joinWithSelf property of current␣
↪object.
```

JSON

```
{
  "version"  1.0
  "result"
     "type" "object"
     "path" "/"
     "fields"
        "foo" "string"
        "bar" "string"
        "fooAndBar"
            "path" "\."
            "type"  "join"
            "typeParams"
                 "fields"  "foo" "bar"


        "joinWithSelf"
           "type"  "join"
           "type" "int"
           "typeParams"
              "separator"  "&"
              "fields"  "foo" "bar" "joinWithSelf"
```

10.2.5 typeParams

typeParams is used for passing extra parameters to *type* field. The typeParams field is *extended* to *params* field.

Examples:

YAML

```
version
result
  type
  path
  fields
     foo
     bar
     fooAndBar
         path      # when current object closed
         type          # declare join type to concat foo and bar field
```

(continues on next page)

(continued from previous page)

```
        typeParams          # typeParams is used to pass fields parameter to type
```

JSON

```
"version"  1.0
"result"
   "type" "object"
   "path" "/"
   "fields"
      "foo" "string"
      "bar" "string"
      "fooAndBar"
          "path" "\."
          "type"  "join"
          "typeParams"
               "fields"  "foo" "bar"
```

10.2.6 path and parentPath

The path and parentPath fields indicate which tags are used in the *source document* during parsing. The value of those fields are regular expressions.

The **path** field specifies the location of a tag in the *source document* relative to the path field of the higher-level *Parsing Element* definition. The default value is the value in the *fieldName* field.

The **parentPath** field is used in a slightly more complex parsing definitions. it holds path to parent tag of the tag specified in the "path" field.

The *path* and *parentPath* fields can be defined as the relative path as in unix. Relative paths are resolved according to the structure in the DSM document, not by the structure in the source file.

Some example of relative path:

current
: ./

parent
: ../

parentOfParent
: ../../

categoryInParent
: ../category

categoryInCurrent
: ./category

To find the exact tag path for the current *Parsing Element*, starting from the result field, all the *parentPath* and *path* fields from top to bottom are merged with the "/" character.

tagParentAbsolutePath and **tagAbsolutePath** evaluated as follow:

> **currentObject** mean is *Parsing Element* that current path and parentPath is defined **parentObject** mean is parent *Parsing Element* of currentObject
>
> - parentObject = if (**parentPath+path**) is relative path then find parentObject by resolving relative path else currentObject.parent
> - **tagParentAbsolutePath** = parentObject.path+"/"+currentObject.parentPath
> - **tagAbsolutePath** = absoluteParentPath+"/"+currentObject.path

if **tagAbsolutePath** regex match any *absolute* path of tag, value of this tag evaluated by *type*

To explain with example:

```
result
   path                                                    #path = orders/order
                                                             #parentPath = ""
                                                             #tagAbsolutePath= /orders/order
                                                             #tagParentAbsolutePath= /
   fields
      defaultName
↪#path =defaultName (default value is fieldName)
                                                             #parentPath = ""
↪#tagParentAbsolutePath =/orders/order (parent(result) absoluteTagPath))
↪#tagAbsolutePath =/orders/order/defaultName
      tagPathDefined
         path                                              #path =status
                                                             #parentPath = ""
↪#tagParentAbsolutePath =/orders/order (parent(result) absoluteTagPath))
↪#tagAbsolutePath =/orders/order/status
      tagPathAndParentPath
            path                                           #path = name
            parentPath                                     #parentPath  = category
↪#tagParentAbsolutePath =/orders/order/category
↪#tagAbsolutePath =/orders/order/category/name
      innerObject
            type
            path                                           #path = "innerObject"
                                                             #parentPath = ""
↪#tagParentAbsolutePath =/orders/order
↪#tagAbsolutePath =/orders/order/innerObject
            fields
               normalPathInInnerObject                     #path = "normalPathInInnerObject"
```

(continues on next page)

(continued from previous page)

```
                                                        #parentPath = ""
                                                        ␣
↪#tagParentAbsolutePath =/orders/order/innerObject
                                                        ␣
↪#tagAbsolutePath =/orders/order/innerObjcet/normalPathInInnerObject
            relativeTagPath
                                                    ␣
↪#path = ../defaultName    (only level up)
                                                        #parentPath = ""
                                                        ␣
↪#tagParentAbsolutePath =/orders/order
                                                        ␣
↪#tagAbsolutePath =/orders/order/defaultName
            relativeParentPath
                                                    #path = defaultName
                                                    #parentPath = "../" (only level up)
                                                        ␣
↪#tagParentAbsolutePath =/orders/order
                                                        ␣
↪#tagAbsolutePath =/orders/order/defaultName                                    relativeTagPathA
                                                    #path = defaultName
                                                    #parentPath = "../" (only level up)
                                                        ␣
↪#tagParentAbsolutePath =/orders/order
                                                        ␣
↪#tagAbsolutePath =/orders/order/../defaultName (relative path of parentPath important. path considere

            relativePathFromResult
                                                    #path = /orders/order/defaultName
                                                        #parentPath = ""
                                                        ␣
↪#tagParentAbsolutePath =/orders/order/defaultName
                                                        #tagAbsolutePath =/orders/order
```

tagAbsolutePath and tagParentAbsolutePath:

fieldName	path	parentPath	tagAbsoLutePath	tagParentAbsoLutePath
result	orders/order		order/simpleOrder	/
result	orders/order		order/simpleOrder	/
defaultName	defaultName(default value is fieldName)	order/simpleOrder/defaultName	order/simpleOrder	
tagPathDefined	status	order/simpleOrder/status	order/simpleOrder	
tagPathAndParentPath	status	order/simpleOrder/category/name	order/simpleOrder/category	
innerObject	innerObject	order/simpleOrder/innerObject	order/simpleOrder	
normalPathInInnerObject	normalPathInInnerObject	order/simpleOrder/innerObject/normalPathInInnerObject	order/simpleOrder/innerObject	
relativeTagPath	defaultName	order/simpleOrder/defaultName	order/simpleOrder	
relativeParentPath	defaultName	order/simpleOrder/defaultName	order/simpleOrder	
relativeTagPathAndParentPath	defaultName	order/simpleOrder/..defaultName (relative path of parentPath important. path considered as regex)	order/simpleOrder	
relativePathFromResult	/orders/order/defaultName	order/simpleOrder/defaultName	order/simpleOrder	

10.2.7 fields

The fields field is used to define the properties of complex objects. Only *Parsing Element* that has complex *type* can have the "fields" field.

The fields field is a map that keys are fieldName of *Parsing Element* , values are string, *Parsing Element* or list of *Parsing Element*

10.2.7.1 Type of Value Definition

The value of the map can be empty, string, *Parsing Element* or array of *Parsing Element*.

Different value definitions create *Parsing Element* with some default values.

Below explain type of value definition and the default values of the *Parsing Element* that it creates.

empty:

fieldName
key of the map.

dataType
string

path
key of the map (fieldName)

parentPath
null

string:

fieldName
key of the map.

dataType
value of the map

path
key of the map (fieldName)

parentPath
null

***Parsing Element*:**

fieldName
key of the map.

other fields are can be defined or initialized with default values.

Array of *Parsing Element*:
some fields of objects can be read from the different tag in the source document. By making multiple definitions for one field, the value of different tags can be read.

Example of different type of value definition:

YAML

```yaml
version
result
  type
  path
  fields
     name       # fieldName is "name" and dataType is  string and the path is "/name"
     category
     price           ␣
↪# fieldName is "price" and dataType is  "long"  and the path is "/price"
     categoryType    ␣
↪# fieldName is "categoryType" and it is  "string" value and the path is "/categoryType" it has extra
            default  "foo"    # default value a is a string.
     productUnit  ␣
↪# this field contains two definition. one  of that  will win depending on the structure of source doc
             path                      ␣
↪# fieldName is "productUnit" and dataType is  "long"  and the path is "/unit/unit_name"
             default                                    'KG'

             path                      ␣
↪# fieldName is "productUnit" and dataType is  "long"  and the path is "/mainUnit/unit_name"
```

JSON

```
{
   "version": 1.0,
   "result":{
      "type"object",
      "path":"/"  ,
      "fields":{
         "name":"",
         "category":"",
         "price":"long",
         "categoryType":{
            "default": "foo"
         },
         "productUnit":[
         {  "path": "unit/unit_name",
            "default": " $self.data.categoryType=='foo'? 'LT': 'KG'"
         },
         {
            "path": "mainUnit/unit_name",
         },
         ],
       }
   }
 }
```

10.2.8 filter

The filter field determines whether the value of a *Parsing Element* (complex or simple *type* does not matter) is added to the object tree. The filter field is an *expression* that returns true or false.

The following objects are available in Expression Context.

See also:

expression

Name	Data Type	Description	Example
params	Map<string,any>	*params* object.	**params.dateFormat** ==’dd.MM.yyyy’
self	**Node_**	current node object that hold data of current complex *type*	**self.data.foo** => foo field of current node, **self.parent.data.foo** => foo field of parent node, **self.data.bar.foo** => foo field of bar object in current node.
all	Map<string,Node_>	A map that stores all nodes by the “*uniqueName*” of *Parsing Element*	**all.bar.data.foo** => foo field of **bar** node, **all.barList.data[0].foo** => *foo* field of first item of *barList* node
value	string	value of the current tag in *source document*	**value==’Computer’**,**value.startWith(‘bar’)**

Examples:

Example 1

YAML

```
version


# collect all data that category field is 'Computer'
  fields
   name
   category
```

JSON

```
  "version"  1.0
  "result"
     "type" "array"
     "path" "/"
     "filter"  "$self.data.category=='Computer'"
     "fields"
        "name" "string"
        "category" "string"
```

Example 2

YAML

```
version


   name
   category
     filter                    # only assign "category" if "category" is "computer".
```

JSON

```
  "version"  1.0
  "result"
     "type" "array"
     "path" "/"
     "fields"
        "name" "string"
        "category"
           "filter"  "$value=='Computer'"
```

Possible Output Of Example 2

```
  "name" "foo"
  "category" "Computer"


  "name" "foo"
```

10.2.9 default

The *default* field holds the value to be assigned to a property by default. The default value is assigned when the *path* does not match the *absolute* path of any tag in the *source document*. If the value of the default field is a string, this value is accepted as the *value* field of the *Default Object*.

See also:

Default Object

Expression

assignment order of the default is from the bottom to up in an object.

Examples

YAML

```
version
result
  type
  path
  fields
     name
           filter
           default
              value       # force set name to foo with filter
              force
           path
     category
     productUnit
           default                                          'KG'
# default value is expression.  this default value is assigned after "categoryType" field assigned.
     categoryType
            default  "foo"    # default value a is a string.
```

JSON

```
  "version"  1.0
  "result"
     "type" "object"
     "path" "/"
     "fields"
        "name" "string"
```

(continues on next page)

(continued from previous page)

```
        "category" "string"
        "productUnit"
           "default"  " $self.data.categoryType=='foo'? 'LT': 'KG'"

        "categoryType"
           "default"  "foo"
```

10.2.10 transformationCode

transformationCode field refers to the definition of the **transformation_** to be used to transform the tag value.

Below definition work as follows:

- value of tag "/country_code" is read from *source document*
- if this value exist in "COUNTRY_CODE_TO_NAME" **transformation_** definition, get value that match.
- if not exist, get "DEFAULT" value of "COUNTRY_CODE_TO_NAME" **transformation_** definition

YAML

```
version
transformations
    COUNTRY_CODE_TO_NAME
        map
          DEFAULT
          TR
          US

 result
   type
   path
   fields
        country
            path
            transformationCode
```

JSON

```
  "version"  1.0
  "transformations"
     "COUNTRY_CODE_TO_NAME"
      "map"
        "TR" "Turkey"
        "US" "United States"
        "DEFAULT" "Other"
```

(continues on next page)

(continued from previous page)

```
"result"
  "type"  "object"
  "path"  "/"
  "fields"
       "country"
           "path"  "country_code"
           "transformationCode"  "COUNTRY_CODE_TO_NAME"
```

See also:

Transformations

10.2.11 function

The *function* field refers to the definition of *functions* field to be used for the custom operation. For more detail about how *functions* works, look at *functions* sections.

Below definition work as follows:

- all fields of product are read from *source document*
- When the "/product" tag is closed, the "com.example.InsertProduct" function in the "insertProduct" definition is called.

YAML

```
version
functions
   insertProduct

 result
   type
   path
   function
   fields
        name
        price
        image
```

JSON

```
   "version"  1.0
   "functions"
      "insertProduct" "com.example.InsertProduct"
```

(continues on next page)

(continued from previous page)

```
   "result"
      "type"  "object"
      "path"  "/"
      "function"  "insertProduct"
      "fields"
           "name"  "string"
           "price"  "long"
           "image"  "string"
```

See also:

functions

10.2.12 uniqueName

When "fieldName" fields of complex *Parsing Element* definitions are the same in the DSM document, these definitions are differentiated by using the "uniqueKey" field. This field is optional. The default value is the value of the "fieldName" field. The uniqueName field may need in very complex document parsing.

Example Case:

In the following DSM document, both the users and the orders objects have a category field and the category field is an object. The *uniqueName* field is used to differentiate the category objects.

YAML

```
version
result
   type
   path
   fields
        users
            type
            fields
                name
                email
                category
                     type
                     uniqueName
                     fields
                        categoryName
        order
            type
            fields
                id
                category
                  type
                  uniqueName
```

(continues on next page)

(continued from previous page)

```
                fields
                      categoryName
```

JSON

```
"version"  1
"result"
   "type"  "object"
   "path"  "/"
   "fields"
      "users"
         "type"  "array"
         "fields"
            "name"  "string"
            "email"  "string"
            "category"
               "type"  "object"
               "uniqueName"  "userCategory"
               "fields"
                  "categoryName"  "string"



      "order"
         "type"  "object"
         "fields"
            "id"  "string"
            "category"
               "type"  "object"
               "uniqueName"  "orderCategory"
               "fields"
                  "categoryName"  "string"
```

10.2.13 normalize

The normalize is used to normalize the value of the tag being read. Changes can be made to the raw string value of the tag by using normalize field. The value of this field is an expression.

The following objects are available in Expression Context.

Name	Data Type	Description	Example
params	Map<string,any>	*params* object.	**params.dateFormat** ==’dd.MM.yyyy’
self	**Node_**	current node object that hold data of current complex *type*	**self.data.foo** => foo field of current node, **self.parent.data.foo** => foo field of parent node, **self.data.bar.foo** => foo field of bar object in current node.
all	Map<string,Node_>	node that stores all nodes by the “*uniqueName*” of *Parsing Element Object*	**all.bar.data.foo** => foo field of **bar** node, **all.barList.data[0].foo** => *foo* field of first item of *barList* node
value	string	raw string value of the current tag in *source document*	**value==’Computer’**,**value.startWith(‘bar’)**

See also:

Expression

10.2.14 xml

The xml field is used to make extra definitions and to change “path” and “type” fields for XML format.

check *XML Object* for more detail

See also:

XML Object

10.2.15 $ref

$ref field is used to *extends Parsing Element* to given *fragments*. it’s value is a Load Time Expression. *fragments* can be *extends* another *fragments* but can not extends itself. Sometimes we don’t need parent properties. To exclude parent properties, define *dataType* as “exclude”. In example bellow category property is excluded.

YAML

```
version
result
  type
  path
  xml
    path  "/Pets/Pet"
  $ref
```

(continues on next page)

(continued from previous page)

```
  fields
     category
# import all properties of fragments.pet except category property.
     isPopular

fragments
   tag
      type
      fields
         id
         name
   category
      type
      fields
         id
         name
   pet
     type
     fields
         id
         name
         status
         category
            $ref
         photoUrls
              type
              path
              xml
                 path
         tags
             type
             path
             xml
               path
             $ref
```

JSON

```
  "version"  1
  "result"
     "type"  "array"
     "path"  "/"
     "xml"
        "path"  "/Pets/Pet"

     "$ref"  "$fragments.pet"
     "fields"
        "category"  "exclude"
        "isPopular"  "default $self.data.tags.stream().anyMatch(s->s.name=='Popular')"
```

(continues on next page)

(continued from previous page)

```
"fragments"
   "tag"
      "type"  "object"
      "fields"
         "id"  "int"
         "name"  "string"


   "category"
      "type"  "object"
      "fields"
         "id"  "int"
         "name"  "string"


   "pet"
      "type"  "object"
      "fields"
         "id"  "long"
         "name"  "string"
         "status"  "string"
         "category"
            "$ref"  "$fragments.category"

         "photoUrls"
            "type"  "array"
            "path"  "photoUrls"
            "xml"
               "path"  "photoUrls/photoUrls"


         "tags"
            "type"  "array"
            "path"  "tags"
            "xml"
               "path"  "tags/tag"

            "$ref"  "$fragments.tag"
```

10.3 Transformation Element Object

Transformation is a very powerful feature that used to map value of a tag from the *source document* to destination document. Transformation Element holds the mapping and how the mapping will be used with `Parsing Element`_s. We can consider, transformation as switch-case in programing language. Every record in the mapping table is a case and DEFAULT record is a default case fo switch-case statement.

Fields:

Field Name	Type	Description
map	Map<String,Objec	**REQUIRED** mapping table from source to destination
only-IfExist	boolean	transform source value only if exist in mapping table. if not exist use as is.
$ref	string	ref field is used to *extends* current Transformation Element to another Transformation Element. it is an expression.

YAML

```
version
transformations
    COUNTRY_CODE_TO_NAME
        map
          DEFAULT
          TR
          US
    COUNTRY_CODe_TO_NAME_IF_EXIST
         $ref
         onlyIfExist
```

JSON

```
        "version"  1
        "transformations"
            "COUNTRY_CODE_TO_NAME"
                    "map"
                            "DEFAULT"  "Other"
                            "TR"  "Turkey"
                            "US"  "United States"


            "COUNTRY_CODe_TO_NAME_IF_EXIST"
                    "$ref"  "$transformations.COUNTRY_CODE_TO_NAME"
                    "onlyIfExist"  true
```

10.4 Default Object

Default Object determines how the *default* field is assigned.

Fields:

Field Name	Type	Description
value	string	**REQUIRED** default value that is assigned to current field
force	string	Use the default value, even if the tag specified in the "*path*" field is in the source file.
atStart	string	assign default value at start of tag.

10.4.1 value

REQUIRED it holds default value that is assigned to current field

if the value starts with the "$" character, it is treated as "expression" and is resolved by expression resolver.

The following objects are available in Expression Context.

Name	Data Type	Description	Example
params	Map<string, any>	*params* object.	**params.dateFormat** ==’dd.MM.yyyy’
self	**Node_**	current node object that hold data of current complex *type*	**self.data.foo** => foo field of current node, **self.parent.data.foo** => foo field of parent node, **self.data.bar.foo** => foo field of bar object in current node.
all	Map<string, Node_>	Map that stores all nodes by the "fieldName" of *Parsing Element Object*	**all.bar.data.foo** => foo field of **bar** node, **all.barList.data[0].foo** => *foo* field of first item of *barList* node

10.4.2 force

if it's value is true, it means Use the default value, even if the tag specified in the "*path*" field is in the source file. if force value is true, default value is assigned both start and end of *parentPath*. It is mostly used with filter field or with value in *params*. The default value is false.

10.4.3 atStart

if *atStart* filed is true, default value is assigned at start of the tag. other wise default value is assigned at the end of the tag.

See also:

default

10.5 XML Object

XML Object is used to make extra definitions and to change "*path*" and "*parentPath*" fields for xml format

Fields:

Field Name	Type	Description
path	string	xml specific *path* definition default value is *path* field of *Parsing Element Object*
parentPath	string	xml specific *parentPath* definition. default value is *parentPath* field of *Parsing Element Object*
attribute	boolean	attribute field is indicates that the current *Parsing Element Object* is an attribute on the tag pointed to by the parentPath field in the xml.

10.5.1 attribute

The attribute field is indicates that the current *Parsing Element* is an attribute on the tag pointed to by the *parentPath* field in the xml.

Examples:

YAML

```
version
result
   type
   path
   xml
      path                # xml specific path definition
   fields
        id
           dataType
           xml
             attribute        ␣
↪# id field is an attribute that is located at /Pets/Pet tag.
        name
        price
        image
```

JSON

```
   "version"  1.0
   "result"
      "type"  "object"
      "path"  "/"
      "xml"
         "path"  "/Pets/Pet"

      "fields"
```

(continues on next page)

(continued from previous page)

```
            "id"
                "dataType"  "long"
                "xml"
                   "attribute"  "true"


            "name"  "string"
            "price"  "long"
            "image"  "string"
```

See also:

xml

CHAPTER

ELEVEN

ABSOLUTE PATH

Absolute Path is found by **joining all tag name from top to bottom with "/"** character until specified tag.

Below example is json representation of array of Pet object.

Absolute tag paths are listed below.

```
"id"  1
"category"
  "id"  1
  "name"  "Cats"

"name"  "PetNameForm"
"photoUrls"
  "url1"
  "url2"
  "url3"

"tags"

    "id"  2
    "name"  "New"


    "id"  2
    "name"  "Cute"


"status"  "sold"
```

Absolute tag paths of all field in above json document are listed below.

Field Name	Absolute Path
	/
Pet:id	/id
Pet:category	/category
Pet:category.id	/category/id
Pet:category.name	/category/name
Pet:name	/name
Pet:photoUrls	/photoUrls
Pet:photoUrls.(item)	/photoUrls
Pet:tags	/tags
Pet:tags.id	/tags/id
Pet.tags.name	/tags/name
Pet.status	/status

Same Example for XML:

```
<Pets>
   <Pet>
      <category>
        <id>1589257917030308320</id>
        <name>Cats</name>
      </category>
      <id>6598053714149410844</id>
      <name>PetNameForm</name>
      <photoUrls>
        <photoUrl>url1</photoUrl>
        <photoUrl>url2</photoUrl>
        <photoUrl>url3</photoUrl>
      </photoUrls>
      <status>sold</status>
      <tags>
        <tag>
          <id>4250197027829930927</id>
          <name>New</name>
        </tag>
        <tag>
          <id>8271965854563266871</id>
          <name>Cute</name>
        </tag>
        <tag>
          <id>3487705188883980239</id>
          <name>Popular</name>
        </tag>
      </tags>
    </Pet>
</Pets>
```

Absolute tag paths of all field in above XML document are listed below.

Field Name	Absolute Path
	/
Pets array	/Pets
Pets array item	/Pets/Pet
Pet:id	//Pets/Pet/id
Pet:category	/Pets/Pet/category
Pet:category.id	/Pets/Pet/category/id
Pet:category.name	/Pets/Pet/category/name
Pet:name	/name
Pet:photoUrls	/Pets/Pet/photoUrls
Pet:photoUrls.(item)	/Pets/Pet/photoUrls
Pet:tags	/Pets/Pet/tags
Pet:id	/Pets/Pet/tags/id
Pet:tags.name	/Pets/Pet/tags/name
Pet.status	/Pets/Pet/status

See also:

path

parentPath

CHAPTER

TWELVE

3EXPRESSIONS AND SCRIPTING3EXPRESSIONS AND SCRIPTING

Expressions makes DSM very flexible. Expression allows a value to be filtered, manipulated, modified etc. Expressions can access objects in the expression context and do operations by using these objects.

Expressions can be used at both *source document* parsing time and *DSM document* loading time.

Expressions are resolved by one of Scripting language such a Javascript, Groovy, Apache JEXL or other JSR223 implementations. Expressions must be written with scripting language syntax. Default scripting language is Apache JEXL

See also:

Apache JEXL

There are two type of expression, Loading Time and Parsing Time expressions.

12.1 Loading Time Expression

Loading Time Expression is expressions that is only used during loading of `DSM document`. It allows you to modify structure of DSM document.

Loading Time Expressions are defined in the *$extends* and or *$ref* fields. For more detail check *$extends* and or *$ref* field.

See also:

$extends

$ref

The following fields are available in the expression context.

Name	Data Type	Description	Example
params	Map<string,Object>	*params* object.	params.dateFormat =='dd.MM.yyyy'

12.1.1 Example

YAML

```
version
params
   rootPath
$extends


# use "params" object in expression context to get "rootPath" property
```

JSON

```
  "version"  1.0
  "params"
  "rootPath" "/bar/foo/"

  "$extends"   "/foo/bar/external.json" "$params.rootPath.concat('externalWithExpression.json')"
```

12.2 Parsing Time Expression

Parsing Time Expression is expressions that is only used during parsing of `source document`. It allows you to change the structure of the output, change the property value, import a specific part of the *source document*, filter by property , transform a property, and almost all operations that can be done with custom coding.

The Parsing Time Expressions can be defined in the *filter*, *default*, and *normalize* fields.

The following objects are available in the expression context.

Name	Data Type	Description	Example
params	Map<string, any>	*params* object.	params.dateFormat =='dd.MM.yyyy'
all	Map<string, Node_>	A map that stores all nodes by the "*uniqueName*" of *Parsing Element Object*	all.bar.data.foo => foo field of bar node, all.barList.data[0].foo => *foo* field of first item of *barList* node
self	Node_	current node object that hold data of current complex *type*	self.data.foo => foo field of current node, self.parent.data.foo => foo field of parent node, self.data.bar.foo => foo field of bar object in current node.
value	string	(not available in *default* field) The raw string value of the current tag in *source document*	value=='Computer',**value.startWith('bar')**

See also:

Node_ , *default*, *filter*, *normalize*, *$extends*, *$ref*

12.2.1 all

Each complex *type* creates a node_. The created nodes can be accessed using the "all" object in the expressions. Each node is stored in *all* map with the *uniqueName* of the *Parsing Element* that creates the node.

```
result
     type
     fields
          order
               type
               fields
                  state
                  createDate
                  saleLines
                       type
                       fields
                          product
                                type
                                fields
                                    id
                                    name
                                    price
                          quantity
                          unit

          company
               type
               fields
                   id
                   name
                   price
```

for configuration at above following *all* map is created.

```
result
      parent
      data
         order                          ␣
  ↪# contains data of the order node
         company
order
        parent
        data
           orderLines
company
        parent
        data
orderLines
     parent
     data     product product.data      ␣
  ↪# data is array. each item contains product data
product
      parent
      data
```

Example usages:

product.parent
is equals orderLine node

product.parent.data
is equals orderLine.data

product.parent.parent
is equals order

product.parent.parent.data
is equals order.data

product.parent.parent.parent
is equals result

order.data.orderLine
is equals orderLine.data

order.data.orderLine[lastIndex].product
is equals product.data

12.2.2 self

---

current node object that hold data of current complex *type*.

Example usages:

self.parent
parent node

self.data.foo
foo field of current object

self.data[0]
First element of current array

12.2.3 Example

YAML

```
version
result
  type
  path
  fields
     name
           filter                                                 ␣
↪# filter expression.
           default
              value       # force set name to foo with filter
              force
           path
     category
        type
        fields
           id
           name
             default
↪                # default value is expression.
           categoryType
               default
↪       ␣
↪# default value is expression. and its is equivalent of expression in
     productUnit
           default                                              'KG
↪'     # default value is expression.
     categoryType
            default  "foo"    # default value a is a string.
```

JSON

```
{
   "version": 1.0,
   "result":{
      "type":object",
      "path":"/"  ,
      "fields":{
         "name":"string",
         "category":{
            "id": "int",
            "name": {
               "default": "self.parent.data.
↪categoryType=='foo'? 'Foo':'Bar'",
            },
            "categoryType": {
               "default": "default: all.data.
↪categoryType=='foo'? 'Foo':'Bar'",
            }
         },
         "productUnit":{
            "default": " $self.data.categoryType=='foo
↪'? 'LT': 'KG'"
         },
         "categoryType":{
            "default": "foo"
         }
       }
   }
}
```

CHAPTER

THIRTEEN

MERGE OF DSM DOCUMENT

DSM document can extends to another DSM document by using *$extends* field.

Parsing Element can extends to another Parsing Element by using *$ref* field.

DSM documents and Parsing Elements are merged with each other.

Before going to explain merge process lets make some definition.

source:

DSM Document or Parsing Element that we want to extend to another.

target:

DSM Document or Parsing Element that we want to extend to.

Merge process work as follows:

for every field of target do followings:

1. if field not exist in source, copy value of target to source.
2. if field exist in source do followings
   - 2.1. if dataType of fields is different then skip this field.
   - 2.2. if dataType of fields is same then do followings
     - 2.2.1. if dataType is simpleDataType (string,number) then skip this field (do not copy target to source)
     - 2.2.2. if dataType is array then add target values to start of the source values

2.2.3. if dataType is map then start Merge process for those two map.

Example merge process of DSM documents:

external DSM Document:(external.yaml)

```
version
params
  dateFormat
  rootPath
  acceptedCountryCode    TR US FR
transformations
    COUNTRY_CODE_TO_NAME
        map
          DEFAULT
          TR
          US
result
  fields
     id
     name
     price
```

current DSM Document:

```
version
$extends                                                    ␣
  ↪# resolve expression
params
  rootPath
  acceptedCountryCode   UK
transformations
    COUNTRY_CODE_TO_NAME
        map
          UK
result
  type
  path
  fields
     category
```

After merge process following configuration will take effect:

```
version
$extends
params
  dateFormat              ␣
  ↪# (rule 1) imported from external document
  rootPath                ␣
  ↪# (rule 2.2.1)  overwritten by current DSM document
  acceptedCountryCode    TR US FR UK ␣
  ↪#(rule 2.2.2)  external list element added to start off current list e
transformations     ␣
  ↪#(rule 2.2.3)  transformations field exist in both source and target a
    COUNTRY_CODE_TO_NAME     #(rule 2.2.3)
        map
          UK                      ␣
  ↪# only exist in current DSM document
          DEFAULT                 ␣
  ↪# (rule 1) imported from external document
          TR                ␣
  ↪# (rule 1) imported from external document
          US                      ␣
  ↪# (rule 1) imported from external document

result
  type           # exist only current DSM document
  path
  fields
     category           # exist only current DSM document
     id
     name               # imported from external document
     price
```

CHAPTER

FOURTEEN

PROPERTY ASSIGNMENT ORDER

The property assignment order is very important for the correct operation of the expressions in the filter field and in the default field Referencing a not existing field in "self.data" can cause NullPointerException.

DSM reads *source document* top to bottom in one pass as a stream. Once it reads a tag *source document*, it checks whether *absolute* path of the tag match with tagAbsolutePath_ or taParentAbsolutePath_ of any of *Parsing Element* if Parsing Element founds, value of tag is assigned according to *Parsing Element* definitions.

The Property assignment work as follows:

let's name the tag that is pointed by *path* as current tag and the tag that is pointed by parentPath as parent tag

The property is assigned when current tag is closed except *attribute* properties for the XML document. The *attribute* properties is assigned at start of parent tag by reading attribute value of parent tag

Order of the property assignment as follows:

- **the closing of `current tag`_ is near to the document header(starting of parent tag" for attribute)**
- **deeper `current tag`_**
- **Parsing Element definition close to the document header.(assignment start from top to bottom)**

The *default* value of a property is assigned when current tag not exist in source document and parent tag" is closed(for all property, include *attribute*).

***default* value is assigned only once except *force* field is true. if *force* field is true default value is assigned at both start and close of parent tag**

Order of the default value of property assignment as follows:

- **assure the property is not assigned or force field is true**
- **the closing of `parent tag`_is near to the document header.**
- **deeper *parent tag***
- **Parsing Element definition far to the document header.(assignment start bottom to top)**

Example:

```
<Pets>
   <Pet>
      <category>
        <id>1</id>
        <name>Cats</name>
      </category>
      <id>6598053714149410844</id>
      <name>Van Kedisi</name>
      <photoUrls>
        <photoUrl>url1</photoUrl>
        <photoUrl>url2</photoUrl>
        <photoUrl>url3</photoUrl>
      </photoUrls>
      <status>sold</status>
      <tags>
        <tag>
          <id>1</id>
          <name>New</name>
        </tag>
        <tag>
          <id>2</id>
          <name>Cute</name>
        </tag>
        <tag>
          <id>3</id>
          <name>Popular</name>
        </tag>
      </tags>
    </Pet>
</Pets>
```

```
result
  type
  path
  xml
    path  "/Pets/Pet"
  fields


   status
   isPopular


↪
   category
       type
       fields
         name
         id
   photoUrls
        type
        path
        xml
           path
   tags
       type
       path
       xml
         path
       fields
```

DSM read document top to bottom.

- **it founds /Pets/Pet *absolute* path that match with result Parsing Element. Then create a array and put first item into the array.**

```
result=[{}]
```

- **it founds /Pets/Pet/category match with category Parsing Element. then it create a object and assign it to category property**

```
result=[{
 "category":{}
 }]
```

- **it founds /Pets/Pet/category/id match with category.id Parsing Element. then it assign it to id property of category object.**

```
result=[{
 "category":{
   "id": 3
 }
 }]
```

- **it founds /Pets/Pet/category/name match with category.name Parsing Element. then the value is assigned**

```
result=[{
 "category":{
   "id": 3,
   "name": "Cats"
 }
 }]
```

- **it founds /Pets/Pet/id match with id then the value is assigned**

```
result=[{
 "category":{
   "id": 3,
   "name": "Cats"
 }
 "id":1
 }]
```

- **it founds /Pets/Pet/name match with name then the value is assigned**

```
result=[{
 "category":{
   "id": 3,
   "name": "Cats"
 },
 "id":1,
 "name":"Van Kedisi",
 }]
```

- **it founds /Pets/Pet/photoUrls/photoUrl match with photoUrls Parsing Element then the new array is created and assigned**

```
result=[{
 "category":{
   "id": 3,
   "name": "Cats"
 },
 "id":1,
 "name":"Van Kedisi",
 "photoUrls":[]
 }]
```

- **it founds /Pets/Pet/photoUrls/photoUrl match with photoUrls then the value of photoUrls is assigned**

```
result=[{
 "category":{
   "id": 3,
   "name": "Cats"
 },
 "id":1,
 "name":"Van Kedisi",
 "photoUrls":["url1","url2","url3"]
 }]
```

after reading all fields under /Pets/Pet path following result generated.

```
result=[{
 "category":{
   "id": 3,
   "name": "Cats"
 },
 "id":1,
 "name":"Van Kedisi",
 "photoUrls":["url1","url2","url3"],
 "status":"sold",
 "tags":[
       {
          "id":1,
          "name": "New"
       },
       {
          "id":1,
          "name": "Cute"
       },
       {
          "id":1,
          "name": "Popular"
       }
 ]

 }]
```

- **it can't find /Pets/Pet/isPopular but isPopular property has default value. When /Pets/Pet (parent tag) tag is closed then it's expression is evaluated. The result of expression is assigned to isPopular property.**

```
result=[{
  "category":{
    "id": 3,
    "name": "Cats"
  },
  "id":1,
  "name":"Van Kedisi",
  "photoUrls":["url1","url2",url3"],
  "status":"sold",
  "tags":[
        {
           "id":1,
           "name": "New"
        },
        {
           "id":1,
           "name": "Cute"
        },
        {
           "id":1,
           "name": "Popular"
        }
  ],
  "isPopular": true
  }]
```

CHAPTER

FIFTEEN

INDICES AND TABLES

- genindex
- modindex
- search