## An introduction to Greenfox, a schema language describing file system contents

Accompanies a tutorial held at: Declarative Amsterdam, 2020-10-08, 14:00 – 15:30

## What is file system validation?

The term **file system validation** is used for an evaluation of a file system tree, defined as a selected root folder and all files and folders directly or indirectly contained. The evaluation is controlled by a **Greenfox schema**, which is a set of **constraints**. The primary outcome of validation is a set of **validation results**, one result per validation of a single resource against a single constraint. The validation result is structured information which identifies the resource and the constraint, asserts **conformance** or violation and includes details about a possible violation. The validation results are mapped to a **validation report**, which is a list of results or some derived representation, namely statistical information.

## And why might you care?

We are used to and appreciate declarative validation of files, using well-known schema languages like XSD, RelaxNG, JSON Schema, SHACL. Our true interest, however, is often broader – we want systems to be valid, and an important aspect of systems is file system trees – trees of folders and files. Such trees may, for example, accommodate …

* applications in use
* a product to be shipped
* components of infrastructure
* data sources and assets
* test results
* observations, obtained by test execution, monitoring, sanity checks, etc.
* a mixture of the aforementioned

In all cases we cannot help caring about *whether everything is as expected* - important processes depend on conformance of file system contents to expectations.

Greenfox is a proposal how to validate file system contents declaratively. It is still in an early stage, but at the end of its first year of development I believe the conceptual framework to have reached a certain maturity, capable of guiding future development. If the current scope of functionality addresses at least some of your requirements, you may obtain within hours what otherwise would cost you large developmental effort leading to heaps of complex code. Using Greenfox you invest in an executable description of your expectations, rather than code. This characterization may be a slightly idealizing one, as the description may involve complex expressions which can be regarded as a sort of code; but by and large I think the statement is true. And I invite you to put it to a test.

The outline for this tutorial is this:

* A guided tour, to give you an impression of the scope, look and feel of Greenfox
* This is followed by a brief “big picture” - concepts & major features
* An overview of the available constraint types
* An outlook, especially important because important features have been postponed in the interest of a solid fundament

## Guided tour

Please see folder: $greenfox/declarative-amsterdam-2020/schema

## Big picture

A “big picture” is all important for an understanding of Greenfox. Consider this analogy – how to learn the XQuery language?

1. Learn concepts and principles; (2) Study the catalog of expressions (syntax + semantics)

A similar situation applies to Greenfox:

1. Learn concepts and principles; (2) Study the catalog of constraints (syntax + semantics)

Once you have understood concepts and principles, it is easy to extend your knowledge iteratively, familiarizing yourself with the various types of constraints, one at a time.

### Seven things

The big picture which I propose is a collection of seven concepts:

* Resources
* Constraints
* Shapes
* Target declarations
* Link definitions
* Results & Reports

We’ll deal with them one by one, and then we’re done.

### Resources

There are two kinds of resources – folders and files.

### Constraints

A constraint is a function mapping a single resource to a validation result. A constraint has parameters, syntax and semantics. The syntax describes the representation of the constraint and its parameters; the semantics define how the validation result is determined. We saw various examples, as a reminder here some further examples:

<fileSize eq="0"/>

<value exprXP="//@iata" length="3"/>

<value exprXP="//@iata" length="3" distinct="true"/>

<valuePair expr1XP="/project/@minDate" count1="1"

expr2XP="//milestoneDate" minCount2="3"

cmp="le"/>

<docSimilar linkName="referenceResponse">

<skipItem kind="attribute" localName="timestamp"/>

</docSimilar>

There are many kinds of constraints. The kind can be decomposed into a **type** and an optional **facet**. Examples: FileSizeEq, ValueLength, ValueDistinct, ValuePairCount1, ValuePairMinCount2. Constraints are represented by the content of **constraint elements**. A constraint element has a name equal to a constraint type, and attributes and child elements representing one or more constraints of this type and with different facets. Nodes can be shared by some or all constraints. In the following example, each attribute represents a constraint parameter used by one or more constraints:

<valuePair expr1XP="/project/@minDate" count1="1"

expr2XP="//milestoneDate" minCount2="3"

cmp="le" useDatatype="date"/>

Constraint element: <valuePair>

Constraint parameters

ValuePairCount1 constraint: @expr1XP, @expr2XQ, @count1

ValuePairMinCount2 constraint: @expr1XP, @expr2XQ, @minCount

ValuePairLe constraint: @expr1XP, @expr2XQ, @useDatatype

Parameters are usually atomic, but there are also complex parameters. Example: ignoreValue parameters of a docSimilar constraint, represented by a child element with attributes:

<docSimilar linkName="referenceDoc">  
 <ignoreValue kind="attribute" localName="timestamp"/>  
</docSimilar>

### Shapes

A shape is a container for two things: a set of constraints and a target declaration:

* The shape is represented by a <file> or <folder> element
* The constraints by child elements (or descendants) of the element
* The target declaration by attributes of the shape element. Example:

<file uri="airports.xml">

<fileDate gt="2020-01-01"/>

<links exprXP="//@href"/>

</file>

In this example, the shape is represented by the <file> element, the target declaration is given by the @uri attribute, and the constraints by the child elements of <file>.

### Target declaration

The target declaration may take several different forms – the most common ones being a relative URI (@uri) and a Foxpath expression (@navigateFOX). Here is a different example:

<file hrefXP="/\*/(xs:include, xs:import)/@schemaLocation"

recursive="true">

<value exprXP="/xs:schema/xs:redefine" empty="empty"/>

</file>

This target declaration is expressed by attribute @hrefXP and @recursive. It selects the documents obtained by recursively resolving URIs found at locations identified by an XPath expression. Independent of the kind of target declaration, the basic principle is that the *target declaration is evaluated repeatedly*, once for each resource selected by the parent shape and treating that resource as evaluation context. Consider:

<domain uri="/projects/abc-service">

<folder navigateFOX=".\\xsd-\*">

<file navigateFOX="\*.xsd">

<file hrefXP="/\*/(xs:include, xs:import)/@schemaLocation"

recursive="true">

<value exprXP="/xs:schema/xs:redefine" empty="empty"/>

</file>

</file>

</folder>

</domain>

Follow the trail of selection:

* The target declaration of the folder shape selects all folders xsd-\* under the domain folder.
* *In each of these* folders, all files \*.xsd are selected.
* *For each of these* files, all directly or indirectly imported or included XSDs are selected.

The innermost <file> shape is a child of another <file> shape. There is nothing surprising about that. A parent/child relationship between shapes does not mean that their target resources have a parent/child relationship; it means that the target declaration of the child shape is evaluated once for each resource from the target of the parent shape, treating that resource as evaluation context.

The analogy with path expression as defined by the XPath language should be noted. A path expression

E1/E2/E3

is evaluated as follows:

* Evaluate E1
* For each item in the value of E1: evaluate E2, treating that item as evaluation context
* For each item in the value of E2: evaluated E3, treating that item as evaluation context
* The items obtained are the value of the path expression

Similarly, nested shapes like

<folder TD1="…">

<file TD2="…">

<file TD3="…">…

…

</folder>

are evaluated as follows (note that attributes @TD\* are placeholder for other attributes expressing a target declaration, e.g. @uri or @navigateXP):

* Evaluate targetDeclaration TD1, treating the domain folder as evaluation context
* For each folder in the value of TD1: evaluate TD2, treating that folder as evaluation context
* For each file in the value of TD2: evaluate TD3, treating that file as evaluation context
* The files obtained are the target of the innermost file shape

### Link definitions

Target declarations are mappings – they map a resource from the target of the parent shape to a set of resources added to the target of the current shape. Such mappings are also required by **binary constraints** – constraint types designed to be applied to a pair of resources. An example is the DocSimilar constraint type. Consider:

<docSimilar linkName="referenceResponse">

<ignoreValue kind="attribute" localName="timestamp"/>

</docSimilar>

A DocSimilar constraint checks whether a given resource has content which is similar to the content of certain other resources. More precisely, the target resource is validated by comparing its contents to the contents of other resources selected by a **link definition**. What is a link definition? It specifies the mapping of a given resource called the **context resource** to a set of other resources called **link target resources**. (Distinguish between the terms *link target resource* and *shape target resource*, the latter often simply called *target resource*.)

Binary constraints use a link definition in order to determine the pairs of resources to which the checks must be applied. Resource shapes use a link definition in order to determine the resource target. Although the goals are different, the means are the same, which is supplying a link defintiion. As the syntactic means for specifying a target declaration (attributes and child elements) are independent of how the link targets are used, resource shapes and binary constraints support the same set of attributes and child elements available for link definition. For instance, attributes @uri, @navigateFOX and @uriXP are available in <file> and <folder> elements as well as in binary constraint elements like <docSimilar> or <valueCompared>.

A link definition may be **local** – defined by attributes and child elements on the element in need of link targets; or it can be referenced by name. The schema may contain **global link defintions** which can be referenced by name. Example:

<greenfox greenfoxURI="… "xmlns="http://www.greenfox.org/ns/schema">

<!-- \*\*\* Context variables -->  
 <context>…</context>  
   
 <!-- \*\*\* Named link definitions -->  
 <linkDef name="hrefLinks" hrefXP="//\*:href"/>

<!-- \*\*\* Domain and its shapes -->  
 <domain uri="…" name="dc2020">

…

<file linkName="hrefLinks">…</file>  
 </domain>  
  
</greenfox>

### Results and Reports

What is the outcome of Greenfox validation? The validation of a file system tree against a Greenfox schema is a processing which is composed of a fundamental building block - validation of a *single resource* against a *single constraint*. The execution of such a constraint validation produces an element called a **validation result**,:

resource + constraint = validation-result

In the typical case, the validation of a single resource against a single constraint produces a single validation result. In some well-defined cases, more results can be produced. This is always the case when a binary constraint has more than one link target resource.

The validation result is an element named after a **colour** which signals conformance – red, yellow, green. In the very special case that a validation is only performed in order to assist another validation –e.g. for checking a condition defined by a Conditional constraint – this subordinate role is signaled by a composite colour - whitered, whiteyellow and whitegreen results.

The primary outcome of a file system tree against a Greenfox schema is a collection of validation results, which is mapped to a validation report:

File-system-tree + Greenfox-schema = validation-result+

Validation-result+ + call-parameters = validation-report

Currently, the default report is statistical – it does not expose the validation results themselves. Example:

G r e e n f o x r e p o r t s u m m a r y

greenfox: C:/tt/greenfox/declarative-amsterdam-2020/schema/air03.gfox.xml

domain: C:/tt/greenfox/declarative-amsterdam-2020/data/air

#red: 2 (2 resources)

#green: 41 (4 resources)

--------------------------------------------

| Constraint Comp | #red | #green |

|--------------------------|------|--------

| FileSizeEq ............. | - | 1 |

| FolderContentClosed .... | - | 1 |

| FolderContentMemberFile | - | 1 |

| FolderContentMemberFiles | - | 6 |

| FolderContentMinCount .. | - | 1 |

| TargetCount ............ | - | 1 |

| TargetMinCount ......... | - | 2 |

| ValueDatatype .......... | - | 3 |

| ValueEq ................ | - | 3 |

| ValueItemsDistinct ..... | 1 | 2 |

| ValueLt ................ | 1 | 2 |

| ValueMatches ........... | - | 3 |

| ValueMinCount .......... | - | 15 |

--------------------------------------------

Red resources:

F C:/declarative-amsterdam-2020/data/air/airports/index/airports-denmark.xml (ValueItemsDistinct)

F C:/declarative-amsterdam-2020/data/air/airports/index/airports-ireland.xml (ValueLt)

There are two red results – in order to see these red elements, repeat the call with option –r (for “red”), a report type providing all red validation results, grouped by resource:

<gx:validationReport … reportType="red" reportMediatype="application/xml">

<gx:redResources count="2">

<!--

\*\*\* C:/tt/greenfox/declarative-amsterdam-2020/data/air/airports/index/airports-denmark.xml

-->

<gx:redResource file="C: /declarative-amsterdam-2020/data/air/airports/index/airports-denmark.xml">

<gx:red msg="IDs not distinct"

constraintComp="ValueItemsDistinct"

constraintPath="gx:values[1]/gx:value[5]/@distinct"

resourceShapePath="/gx:greenfox[1]/gx:domain[1]/gx:folder[1]/gx:file[1]"

resourceShapeID="file\_2" distinct="true" valueCount="31" exprLang="xpath"

expr="//airport/@id" quantifier="all">

<gx:value nodePath="/airportsForCountry[1]/airport[1]/@id">607</gx:value>

<gx:value nodePath="/airportsForCountry[1]/airport[2]/@id">607</gx:value>

</gx:red>

</gx:redResource>

<!--

\*\*\* C:/ declarative-amsterdam-2020/data/air/airports/index/airports-ireland.xml

-->

<gx:redResource file="C:/declarative-amsterdam-2020/data/air/airports/index/airports-ireland.xml">

<gx:red msg="Airport too high"

constraintComp="ValueLt"

constraintPath="gx:values[1]/gx:value[2]/@lt"

resourceShapePath="/gx:greenfox[1]/gx:domain[1]/gx:folder[1]/gx:file[1]"

resourceShapeID="file\_2" lt="1000"

useDatatype="integer"

valueCount="3"

exprLang="xpath"

expr="//altitude"

quantifier="all">

<gx:value nodePath="/airportsForCountry[1]/airport[6]/geo[1]/altitude[1]">1319</gx:value>

<gx:value nodePath="/airportsForCountry[1]/airport[16]/geo[1]/altitude[1]">1001</gx:value>

</gx:red>

</gx:redResource>

</gx:redResources>

</gx:validationReport>

A glance suffices to understand that validation results are very fine-grained structured information. A central goal of Greenfox is to ensure access to finest-grained information about the state of the system under investigation.

The use of validation reports may be facilitated by requesting filtered results: use options to filter by constraint type (-F) or by resource name (-R). Selection can be very fine-grained, using inclusive and exclusive name filters. The following invocation

gfox … -C "value\* ~\*count\*" -R "\*ireland\* \*finland\*"

filters validation results: include only results for particular constraints (matching \*value\*, but not matching \*count\*) and particular resources (matching \*ireland\* or \*finland\*).

## Further important topics

Striving for a basic understanding of Greenfox, you have two main tasks:

* Familiarize yourself with key concepts and principles
* Get a cursory overview of the available constraint types

You already got an overview of the basic building blocks of validation input and output: resources, constraints, shapes, target declarations, link definitions, results and reports. The next step is learning about a set of concepts also required for understanding the potential and limitation of Greenfox:

* Available **expression languages**
* The **evaluation context**
* The dealing with **non-XML mediatypes**
* The **context variables** of a Greenfox schema
* Usage, syntax and semantics of **link definitions**
* A few **syntax principles**

### Expression languages

These expression languages are supported:

* Foxpath
* XPath
* NodePath
* LinesPath

#### Foxpath

Foxpath is an extended version of XPath 3.0, supporting file system navigation, node tree navigation and mixing both within an expression. This makes it a tool for solving tasks of file system navigation with the ease and elegance you are used to from XPath. As Foxpath supports both, file system and node tree navigation, it uses two step separators, the slash (separating steps of conventional path expressions) and backslash – separating steps of file system navigation. A few examples give you an impression. You can try out the examples yourself using the fox command-line tool, found in the bin folder of the Greenfox project. Pass the Foxpath expression to the fox script which is included in the Greenfox project. In the expression text, $da2020 should be replaced with the absolute or relative path of the declarative-amsterdam-2020 folder. Any linefeeds in the examples below have been added for readability and must not be used on the command-line. Using option –b, the separators for file system and node tree navigation are backslash and slash, respectively. Without option –b, their roles are swapped.

fox -b "$da2020\data\air\airports\index\\*"

*Result: the file paths of all files and folders in the index folder*

fox -b "$da2020\data\\airport-\*.xml\ancestor~::\*[parent~::countries]\file-name()"

*Result: the names of folders containing airport XML files. Note the use of reverse navigation axes (ancestor~::, parent~::), the use of a predicate and a non-navigational path step.*

fox -b "$da2020\data\\airports\index\\*.xml

[/airportsForCountry[.//latitude[xs:decimal(.) lt 10]]]"

*Result: the file paths of XML files with a <airportsForCountry> root element and a latitude less than 10. Note the use of node tree navigation in a predicate of file system navigation.*

Foxpath can deal with non-XML formats (JSON, CSV, HTML) as if they were XML, parsing them into node trees:

fox -b "$da2020\data\\airport-\*.json[jdoc()//timezone = 0] => count()"

*Result: The number of JSON airport document containing a timezone equal 0.*

fox -b "$da2020\data\air\resources\openflights\\*.csv

[csv-doc()/csv/record[\*[4] = 'Papua New Guinea']]"

*Result: CSV documents with a record which holds in the fourth column the value “Papua New Guinea”*

Using CSV files, parameters are available for dealing with non-comma separators and headlines. Example:

fox -b "$da2020\data\resources\geo\cow.csv\csv-doc(., 'semicolon', 'yes')

/csv/record/ISOen\_name[. ne ../ISOen\_proper]

/concat(., ';', ../ISOen\_proper)"

*Result: A sorted list of all pairs – ISOen\_name ; ISOen\_proper – where the two are different (what is rare).*

In Greenfox, you use Foxpath for various purposes:

* As **target declaration**, selecting the target resources of a shape
* As **link definition**, selecting the link target resources of a link definition
* As **resource value** to be checked against constraints (e.g. Foxvalue constraints)

SYNTAX RULE:

Foxpath expressions are contained by attributes with the name suffix FOX.

Examples: @navigateFOX, @exprFOX, @expr1FOX, @reflector2FOX.

#### XPath

XPath (version 3.1) is used for the following purposes:

* As **resource value** to be checked against constraints (e.g. Value constraints)
* As **focus node**, shifting the evaluation context to selected inner nodes
* As **part of link definitions**, e.g. selecting link context nodes or constructing URIs

SYNTAX RULE:

XPath expressions are contained by attributes with the name suffix XP.

Examples: @contextXP, @targetXP, @exprXP.

#### LinePath

LinePath expressions are XPath expressions evaluated in the context of a document obtained by representing the lines of a text file by <line> elements wrapped in a <lines> element. This enables the use of XPath expressions for evaluating text content which cannot be parsed into “normal” node trees. For example, the following line path expression selects a version number:

/lines/line[matches(.,"^Version:")]/replace(.,"^Version:\s\*(.+)\s\*$", "$1 ")

SYNTAX RULE:

LinePath expressions are contained by attributes with the name suffix LP.

Examples: @exprLP, @expr1LP, @expr2LP

#### NodePath

NodePath is a deliberately simplistic navigation language which is used for describing document tree structure. The syntax is similar to XPath.

SYNTAX RULE:

NodePath expressions are contained by attributes with the name suffix NP.

Example: @locNP

### Evaluation context

When expressions are evaluated, it is crucial to have a clear understanding of the evaluation context. The evaluation context comprises the *initial context item* and *variable bindings*.

#### Foxpath context

In Greenfox, the **context item** of a Foxpath expression is always a *resource URI*, not a node. The resource is either a shape target resource (the resource which is currently validated), or a link target resource.

The choice of **context resource** (the resource providing the context item) depends on the attribute containing the expression. Ignoring a couple of exceptions (see [Table Evaluation Context](#table_evaluation_context)), the context resource is the …

* *Link target resource* - if the expression attribute is @targetXP or \*Compared/@expr2\*
* *Shape target resource* - otherwise

If existent, relevant nodes are made available via **variable bindings:**

* Nodes of the shape target resource ($doc, $focusNode, $linkContext, $lines)
* Nodes of the link target resource ($targetDoc, $targetNode)

The evaluation context also comprises further variables with atomic values: $fileName, $filePath, $domain, as well as the value of a “compagnon expression” ($value, $item). The evaluation context of Foxpath and XPath expressions is summarized by the following table.

**Table. Evaluation context of Foxpath and XPath expressions.** The first row (@\*FOX, @\*XP) applies to all expressions, remaining rows only to expressions in attributes with a name matching the first column and only if the conditions (last column) are satisfied.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Expression**  **attribute** | **Context resource** | **Variable name** | **Variable value** | **Conditions**  ***(or remark)*** |
| @\*FOX  @\*XP  (except field/@\*) | Shape or link target resource | $doc | Root node of the shape target resource | Shape target resource can be parsed into a node tree |
| $focusNode | Current focus node | Relevant constraint has a <focusNode> parent |
| $lines | Node representation of content lines of the shape target resource | - |
| $fileName | File name of the shape target resource | - |
| $filePath | URI of the shape target resource | - |
| $domain | URI of the domain folder | - |
|  |  | Names (@name) of all <field> elements in the <context> | Values (@value,m @valueFOX, @valueXP) of all <field> elements in the <context> | - |
| @expr2FOX  @expr2XP | Shape or link target resource | [see  @\*FOX, @\*XP] | [see  @\*FOX, @\*XP] | *(If the attribute belongs to a <\*Compared> constraint, the context resource is the link target, otherwise the shape target resource)* |
| $value | Value obtained from sibling attribute @expr1\* | *(The complete value, not a single item)* |
| $item | An item returned by @expr1\* (\*=FOX|XP|LP); may be a node or an atom | The attribute has a @expr2Context sibling with value item |
| $linkContext | Link context item (a content node, the root node or the URI) | The attribute belongs to a <\*Compared> constraint |
| $targetDoc | Root node of link target resource | The attribute belongs to a <\*Compared> constraint with a link target resource parsed as a node tree |
| $targetNode | A node from the link target resource | The attribute belongs to a <\*Compared> constraint with a link definition containing a target selector (@targetXP) |
| @targetXP | Link target resource | [see  @\*FOX, @\*XP] | [see  @\*FOX, @\*XP] |  |
| $linkContext | Link context item (content node, root node or URI) | The attribute belongs to a link definition with a link target resource parsed as a node tree |
| @reflector2FOX | Foxpath value of sibling attribute @reflector1FOX, or literal value of sibling attribute @reflector1URI | [see  @\*FOX, @\*XP] | [see  @\*FOX, @\*XP] | - |
| field/@valueFOX  field/@valueXP | Containing Greenfox schema | Variable names (@name) of preceding <field> sibling elements | Variable values (@value, @valueXP, @valueFOX) of preceding <field> sibling elements | - |

#### XPath context

In Greenfox, the **context item** of an XPath expression is usually a **document node** or a **focus node**. It is a focus node if the relevant constraint has a <focusNode> parent. A <focusNode> element selects nodes from the target documents of the containing shape. Consider this schematic example

<file foxpath=".\\geo.xml'">

<focusNode selectXP=".//continent"> <!-- Visit <continent> nodes -->

<!-- … continent check … --> <!-- Context item here: <continent> -->

<focusNode selectXP=".//country"> <!-- Visit <country> nodes -->

<!-- … country checks --> <!-- Context item here: <country> -->

<focusNode selectXP=".//province"> <!-- Visit <province> nodes -->

<!-- … province checks --> <!-- Context item here: <province> -->

</focusNode>

</focusNode>

</focusNode>

</file>

If the expression is the value of attribute @expr2XP and the constraint parameter @expr2Context has the value item, the context item is an item returned by the corresponding first expression (@expr1\*). In all other cases, the context item is the document node of the context resource.

The choice of **context resource** (shape target or link target) depends on the attribute containing the expression. The context resource is the …

* Link target resource – if expression attribute: \*Compared/@expr2XP or @targetXP
* Shape target resource - otherwise

Similar **variable bindings** are available as for [Foxpath](#_Foxpath) expressions. See x.x for details.

#### Linepath context

The name **Linepath** is used for XPath expressions evaluated in the context of a “lines document”, which is a <lines> element with <line> child elements representing the lines of the file text content. The expressions are contained by attributes with a name \*LP.

The evaluation context of expressions in most attributes @\*LP is equal to the context of the XPath expression in an attribute with a matching name (trailing “LP” replaced by “XP”):

* @exprLP – same context as @exprXP
* @expr1LP – same context as @expr1XP
* @expr2LP – same context as @expr2XP
* @contextLP – same context as @contextXP
* @targetXP – same context as @targetXP

The context resource (shape target resource or link target resource) is however *represented by a lines document*, not by the node tree obtained by parsing the resource according to its mediatype.

Special rules apply to expressions in attributes @filter\*LP and @map\*LP:

* expressions in @filter\*LP are re-evaluated in the context of each <line> element from the <lines> document
* expressions in @map\*LP are re-evaluated in the context of each <line> element selected by the accompanying @filter\*LP

### Dealing with non-XML mediatypes

Greenfox supports XPath-based evaluation of non-XML resources.

#### Shape target resource

The first approach is to “annotate” a shape as targeting resources of a particular mediatype, using attribute @mediatype with one of the values json, csv, html:

<file mediatype="json" …>…</file>

<file mediatype="csv" …>…</file>

<file mediatype="html" …>…</file>

Greenfox will automatically parse the target resources of the shape into a node tree representation. It is the implicit *evaluation context* for any XPath expression used in a constraint of the shape (excluding @expr2XP in binary constraints). Here comes an example where a shape targets JSON files and submits them to a value constraint based on an XPath expression:

<file foxpath="airport.json" mediatype="json">

<value exprXP="//iata" length="3"/>

</file>

In the case of mediatype CSV, further attributes are available for controlling the parsing of the CSV file into a node tree:

* @csv.separator - the separator, identified by one of the tokens comma, semicolon, colon, tab,space, or a single character
* @csv.header – a switch indicating whether the first line of the file contains column headers (value yes or no)
* @csv.format – controls the representation of JSON names, if JSON names are used as XML element names (value direct) or provided by XML @name attributes (value attributes)

#### Link target resource

Like shape elements, also link definition elements have an optional attribute @mediatype, which may be one of the values xml, json, csv, html. When the attribute is used, link resolution includes a final step of parsing the link target into a node tree representation. If the parsing fails, link resolution as a whole is considered a failure.

A link definition has additional optional attributes controlling the parsing of CSV targets. These are the same attributes as allowed on a shape element(@csv.separator, @csv.header, @csv.format).

#### Expression context

The node tree representation of the current shape target resource is bound to a variable $doc, available in any XPath or Foxpath expression used in a constraint of the shape, or in a link definition referenced by such a constraint (see [Expression context](#_Expression_context)). In the following example, the $doc variable is referenced by the “second” expression of a ValueCompared constraint:

<linkDef linkName="projects" navigateFOX="…" mediatype="csv">

<file navigateFOX="…" mediatype"json">

<valuesCompared linkName="projects">

<valueCompared expr1XP="//startDate"

expr2XP="//project[@id eq $doc//projectId]/startDate" …/>

</valuesCompared>

</file>

Note that @expr2XP is evaluated in the context of a *link target resource*, not the shape target resource, so that $doc is required for access to data from the shape target resource. Also note that the link target resource is a CSV document, whereas the shape target resource is a JSON document. The constraint thus uses an expression evaluating JSON contents (@expr1XP) and a second expression evaluating CSV contents, yet also accessing JSON contents (@expr2XP).

#### Foxpath function calls

Independently of the current resource from the shape target, Foxpath expressions may contain function calls for parsing arbitrary non-XML resources into node trees:

* jdoc($uri) - parses a JSON resource into a node tree
* hdoc($uri) - parses an HTML resource into a node tree
* cdoc($uri) - parses a CSV resource into a node tree

The function cdoc() has further optional parameters corresponding to the shape attributes controlling CSV parsing (@csv.separator, @csv.header and @csv.format):

* cdoc($uri, $separator)
* cdoc($uri, $separator, $headerFlag,
* cdoc($uri, $separator, $headerFlag, $format)

### The context element of a Greenfox schema

The <context> element can be used in order to define name/value pairs receiving their values from call parameters. Before validation, the schema is modified by replacing variable references (syntax: ${varname}) with the variable values. References may be used in any attribute value. While this replacement is a purely textual operation, the evaluation context of expressions is also extended by corresponding variable bindings. As an example, when invoking the following schema

<greenfox …>

<context>

<field name="maxDate"/>

</context

<domain …>

<file …>

<fileDate lt="${maxDate}"/>

</file>

</domain>

</greenfox>

the caller controls validation by supplying a value for context variable maxDate:

gfox myschema.xml /a/b/c -v maxDate=2019-12-31

The variable is also added to the evaluation context of expressions. Here is a shape which targets files with a file date greater than the user supplied maxDate:

<file navigateFOX=".\\\*[file-date() gt $maxDate]">…</file>

Note the syntactical difference between variable references evaluated as part of expressions ($foo) and variable references indicating textual substitution (${foo}).

Variable names must be NCNames (names which might be used as XML names) and must not start with an underscore. A call supplying values for undefined context variables is rejected with an error message. Variable default values can be defined using a @value attribute on the <field> element:

<field name="maxDate" value="2019-06-30"/>

Values assignment can use literals (@value), Foxpath expressions (@valueFOX) or XPath expressions (@valueXP). The evaluation context of a Foxpath expresssion is the URI of the schema document, the evaluation context of an XPath expression is the root node of the schema document. For example, in the following context

<context>

<field name="domain" valueFOX="ancestor~::decl\*\data"/>

</context>

the variable value is the file path of the data folder contained by a folder matching name pattern decl\* and reached by upward navigation starting at the schema document.

A call *must* provide values for all context variables without a default value. A call failing to do so is rejected with an error message. Variable references can also be used within the context. Example:

<context>

<field name="extension" value="json"/>

<field name="logFileName" value="log.msgs-${extension}.txt"

</context>

Several *built-in context variables* may be referenced as if they had been declarared by the user:

* schemaLocation – the path of the schema file, in Foxpath syntax (using backslashes)
* currentDate – the current date
* currentDateTime – the current date time

### Link resolution

Speaking generally, a link definition is a function mapping a resource to other resources. This definition conveys the basic idea, but it leaves important details unclear, in particular the use of resource fragments. Greenfox is therefore based on a detailed link model, which should be regarded as an elaboration of the basic idea, not as a replacement.

#### Abstract link model

A **link** is a directed association between two resources: a resource – called the link context resource – is mapped to another resource – called the link target resource. Details:

* The **link context resource** is either a complete resource or a resource fragment, understood as a single node from a node tree representation of resource contents.
* The **link target resource** is either a complete resource or a fragment, understood as a set of nodes from a node tree representation of resource contents.

A **link definition** is a mapping of input to output:

* The input of a link definition is a resource URI, called the context resource URI.
* The output of a link definition is a set of links, described by a set of information items:
  + Context resource URI
  + Optionally: context resource tree (node tree representation of the context resource)
  + Optionally: context resource fragment (a single node from the context resource tree)
  + Target resource URI
  + Optionally: target resource tree (node tree representation of the target resource)
  + Optionally: target resource fragment (set of nodes from the target resource tree)

Note that a context resource fragment is a single node, whereas a target resource fragment is a set of nodes, constrained to belong to the node tree representation of a single resource.

A link definition can be divided into three components:

* Context selector – selection of link context fragments (optional)
* Resource connector – selection of target resources
* Target selector – selection of link target fragment (optional)

The optional **context selector** selects nodes from the link context resource. Each selected node is mapped to a distinct link, using that node as context resource fragment. The resource connector is therefore evaluated repeatedly, once for each node obtained from the context selector. The node is the *initial context item* when evaluating XPath expressions used by the connector, and it can be accessed by all expressions (XPath or not) of the link definition via a context variable ($linkContext). If the link definition does not contain a context selector, the resource connector is evaluated only once. In this case, XPath expressions used by the connector are evaluated in the context of the root node of the context resource tree. If the context resource cannot be parsed into a resource tree (e.g. because it is a folder), the resource connector must not use any XPath expressions requiring a context resource tree (@uriXP, @hrefXP, templateVar/@valueXP).

In the following example, the context selector is given by an XPath expression selecting <airport> elements from the context resource. The resource connector is an XPath expression constructing a target URI (@uriXP). The connector expression is evaluated repeatedly, once for each item returned by the context selector and using that item as a context item:

<linkDef name="fooLink"

contextXP="//airport"

uriXP="concat('http://example.com/airport/', @iata)"/>

Thus we get one link for each airport returned by the context selector, and each link identifies a different <airport> element as its context resource fragment. The IATA code used to construct the link target URI is provided by the @iata attribute on the context resource fragment (a particular <airport> element). By contrast, when applying this link definition

<linkDef name= "barLink"

uriXP="//airport/@iata/concat('http://example.com/airport/', .)"/>

the resource connector will be evaluated only once. While also here one link is created for each <airport> element, these links do not identify a context resource fragment, thus do not contain information about the individual <airport> element used in order to construct the target URI. When validating the target resources, dependencies on the triggering <airport> element can only be checked when using the first link definition, although both link definitions yield the same set of target resources.

The optional **target selector** specifies a final step of evaluation, which is applied to each resource obtained from the resource connector. Currently, the target selector can only be an XPath expression (@targetXP). The initial context item for its evaluation is the root node of a node tree representation of the target resource. Use of a target selector is therefore only possible if the target resource can be parsed into a node tree. As long as this is the case, a target selector can be added, changed or removed independently of the resource connector and the context selector, as the selection of a target fragment is a final step of evaluation without impact on any preceding step.

The mandatory **resource connector** has a connector type and parameters dependent on the type.

#### Resource connectors

The resource connector is the key piece of a link defintion, as it maps the context resource to other resources. Such mappings may be defined in very different ways, e.g. locating target URIs in the content of the context resource, constructing target URIs from pieces of data found in the context resource contents, or evaluating a Foxpath expression. These different approaches are modeled as **connector types**: each connector type has a specific set of parameters. The current version of the Greenfox language supports six different connector types. Each connector type has a specific set of parameters, summarized in the following table. Future versions of the Greenfox language may support additional connector types, as well as additional parameters for the types currently included.

**Table. Resource connector types and their parameters.**

|  |  |  |  |
| --- | --- | --- | --- |
| **Connector Type** | **Parameter** | **Meaning** | **Note** |
| uri | @uri | The URI of the link target. | The URI may be relative or absolute. When it is relative, it is evaluated relative to the context resource URI, however treating this context URI *as if* it had a trailing slash – e.g. uri="foo" selects the foo child of the context resource, not the foo sibling |
| uri-expression | @uriXP | XPath expression returning the URIs of the link targets. | When evaluating the XPath expression, the context item is a node from the context resource tree: a node returned by the context selector, if the link definition contains a context selector, or the root node, otherwise. Relative URIs are resolved against the context resource URI in the standard way (not assuming a trailing slash). |
| href-expression | @hrefXP | XPath expression returning nodes containing the URIs of the link targets. | Evaluation context item and resolving of relative URIs as in the case of @uriXP. |
| uri-template | @uriTemplate | A template to be resolved to the URIs of the link targets; with placeholders for template variables defined by templateVar child elements | Template variable references are replaced by single items from the template variable values; each combination of value items yields a URI.  Resolving of relative URIs as in the case of @uriXP |
| templateVar/  @name | Name of a template variable | Must be an NCName. |
| templateVar/  @valueXP | XPath expression returning the variable value | Evaluation context item as in the case of @uriXP. |
| mirror | @reflector1URI  |  @reflector1FOX | URI of reflector 1, or a Foxpath expression returning that URI; reflector 1 is a folder containing the link context resource, which is mapped to a link target resource found at the same relative path under folder reflector 2 as the context resource is found under reflector 1. | When specified as a URI, it may be relative or absolute; resolving of a relative URI as in the case of @uri; when specified as a Foxpath expression, the context item is the context resource URI and the expression value must contain at most one item; URI and FOX variants may be combined with URI and FOX variants for reflector 2. |
| @reflector2URI  |  @reflector2FOX | URI of reflector 2, or a Foxpath expression returning that URI | Resolving of a relative URI against the URI of reflector 1; when specified as a Foxpath expression, the context item is the URI of reflector 1; see parameter @reflector1URI or @reflector1FOX for more information. |
| @reflected-  -ReplaceSubstring | The URI obtained from the reflectors is modified by replacing this substring with a string supplied by @reflectedReplaceWith | Describes an optional “final” editing of the preliminary target URI obtained from the reflectors |
| @reflected-  -ReplaceWith | The URI obtained from the reflectors is modified by replacing the substring supplied by @reflected-ReplaceSubstring with this string | Describes an optional “final” editing of the preliminary target URI obtained from the reflectors |
| foxpath | @navigateFOX | A Foxpath expression returning the link target resources. | Atomic value items are interpreted as link target URIs, and node items are interpreted as nodes from the node tree representation of a target resource |

#### Resolving link definitions – pseudo code

The evaluation of a link definition can be described by pseudo-code (see listing below). The pseudo-code relies on four pseudo functions:

* node-tree(uri) - parses a URI into a node tree;
* apply-expression(expr, context-item) - evaluates an expression
* apply-connector(connector, context-point) - evaluates a resource connector
* LRO(target-uri, target-tree, target-nodes, context-uri,

context-tree, context-node) – constructs a Link Resolution Object

**Listing. Pseudo-code of the evaluation of a link definition.** The link definition is represented by a Link Definition Object ($LDO); evaluation output is a sequence of Link Resolution Objects (LROs).

LROS($context-uri, $LDO):

let $context-tree = null

let $context-points :=

if (exists($LDO.context-selector)):

let $context-tree = node-tree($context-uri)

apply-expression($LDO.context-selector, $context-tree)

else:

if ($LDO.resource-connector.requires-node-tree):

node-tree($context-uri)

else:

$context-uri

for each $context-point in $context-points:

let $target-uris := apply-connector($LDO.connector, $context-point)

for each $target-uri in $target-uris:

if (exists($LDO.target-selector)):

let $target-tree := node-tree($target-uri)

let $target-fragment :=

apply-expression($LDO.target-selector, $target-tree)

LRO($target-uri, $target-tree, $target-fragment,

$context-uri, $context-tree, $context-point)

else if ($LDO.expects-target-tree):

let $target-tree := node-tree($target-uri)

LRO($target-uri, $target-tree, null,

$context-uri, $context-tree, $context-point)

else:

LRO($target-uri, null, null,

$context-uri, $context-tree, $context-point)

#### Recursive links

A link definition may be recursive (@recursive="true"). In this case the mapping of link context resource to link target resource is recursively applied to every link target resourced, and the link definition maps a context resource to all target resources discovered. Example:

<linkDef name="xsd-import">

hrefXP="//xs:import/@schemaLocation"

recursive="true "/>

#### Link errors

The description of a link may include an error condition, as summarized by the following table.

**Table. Link errors and their meaning.**

|  |  |
| --- | --- |
| **Error code** | **Meaning** |
| no\_resource | No resource found at the link target URI. |
| no\_text | Link target resource is binary, but a text resource was expected. |
| not\_json | Link target resource not a well-formed JSON document. |
| not\_xml | Link target resource not a well-formed XML document. |
| href\_selection\_not\_nodes | A href selector expression yields non-node items. |
| no\_uri | Failed to determine a link target URI. |

#### Link constraints

A link definition may include constraints which define successful resolution of the link definition as a whole, rather than on the level of the individual links. Constraints are expressed by attributes of a <targetSize> child element, which is a child element of the link defining element. Example:

<linkDef

name="airportLink"

contextXP="//airport"

uriXP="concat('http://example.com/airport/', @iata)">

<targetSize

resolvable="true"

minCountTargetResources="10"

countTargetResourcesPerContextPoint="1"/>

</linkDef>

Constraints built into a link definition are validated whenever the link definition is used. Available constraints are summarized by the following table.

**Table. Available constraints which may be included in a link definition.** For each @count\* constraint, there is also a corresponding @minCount\* and a corresponding @maxCount\* constraint. When no context selector is used (link definition without @contextXP), there is only one context point and therefore the constraints with suffix PerContextPoint are treated like the corresponding constraint without suffix.

|  |  |
| --- | --- |
| **Constraint attribute** | **Meaning** |
| @resolvable | If true, the target URI must point to an existing resource |
| @countTargetResources | The number of target resources, per context resource |
| @countTargetResourcesPerContextPoint | The number of target resources, per context resource fragment (or per context resource, if no context selector has been defined) |
| @countTargetDocs | The number of target resources successfully parsed into a node tree, per context resource |
| @countTargetDocsPerContextPoint | The number of target resources successfully parsed into a node tree, per context resource fragment (or per context resource, if no context selector has been defined) |
| @countTargetNodes | The number of nodes contained in the target fragment, per context resource |
| @countTargetNodesPerContextPoint | The number of nodes contained in the target fragment, per context resource fragment (or per context resource, if no context selector has been defined) |

### A syntax rule to remember

The Greenfox language supports several expression languages, and in some cases a particular detail can be expressed in alternative ways, using different languages. For example, the expression underlying a Value constraint may be expressed as an XPath expression (@exprXP), as a Foxpath expression (@exprFOX), as a line path expression (@exprLP) or as a pair of line path expressions (@filterLP, @mapLP). In other cases, increased flexibility by allowing a choice of expression languages is planned (e.g. for link context selectors). A simple naming rule should help you avoid confusion: attributes expecting an expression have a name suffix indicating the expression language:

* suffix XP – value is an XPath expression
* suffix FOX – value is a Foxpath expression
* suffix LP – value is a Linepath expression
* suffix NP – value is a node path expression

The following table lists for each attribute name suffix the names of all attributes.

|  |  |  |
| --- | --- | --- |
| **Attribute name suffix** | **Meaning** | **Attribute names** |
| XP | Value is an XPath expression | contextXP  expr1XP, expr2XP, exprXP  hrefXP  ifXP  itemXP  targetXP  uriXP  valueXP |
| FOX | Value is a Foxpath expression | expr1FOX, expr2FOX, exprFOX  reflector1FOX, reflector2FOX  valueFOX  xsdFOX |
| LP | Value is a Linepath expression | expr1LP, expr2LP, exprLP  filter1LP, filter2LP, filterLP  map1LP, map2LP, mapLP |
| NP | Value is a Nodepath expression | locNP |

## Constraint types

Having acquired an idea of the basic concepts of Greenfox, the next thing to do is familiarize yourself with the major constraint types available (currently 19).

### Overview

The following table summarizes these types, including information whether the constraint can be used for folders or files only, whether it considers the resource in isolation or in the context of other resources, and whether the constraint is concerned with resource properties or resource contents.

**Table.** **The constraint types supported by Greenfox.** For each constraint type, a varying number of constraint facets is available. A unary constraint is applied to single resources, a binary constraint is applied to pairs of resources. An open constraint allows impact by other resources than the one being validated or the second resource of a pair.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Constraint type** | **Element** | **File (F)**  **or**  **Folder (D)** | **Unary/Binary (U|B) /**  **Closed/Open (C|O)** | **Resource properties (P) or content (C)** |
| FileDate | <fileDate> | F, D | U/C | P |
| FileName | <fileName> | F, D | U/C | P |
| FileSize | <fileSize> | F, D | U/C | P |
| FolderContent | <folderContent> | D | U/C | C |
| Mediatype | <mediatype> | F | U/C | C |
| DocTree | <docTree> | F | U/C | C |
| HyperdocTree | <hyperdocTree> | F, D | U/**O** | C |
| XsdValid | <xsdValid> | F | U/C | C |
| Value | <value> | F | U/C | C |
| ValuePair | <valuePair> | F | U/C | C |
| Foxvalue | <foxvalue> | F, D | U/**O** | C |
| FoxvaluePair | <foxvaluePair> | F, D | U/**O** | C |
| ValueCompared | <valueCompared> | F | **B**/C | C |
| FoxvalueCompared | <foxvalueCompared> | F, D | **B**/**O** | C |
| DocSimilar | <docSimilar> | F | **B**/C | C |
| FolderSimilar | <folderSimilar> | D | **B**/C | C |
| Link | <links> | F, D | U/**O** | *(depends)* |
| TargetSize | <targetSize> | F, D | U/**O** | *(depends)* |
| Conditional | <conditional> | F, D | *(depends)* | *(depends)* |

### Constraint types - summaries

The section offers for each constraint type a short summary.

#### FileDate

FileDate constraints check the last-modified property of the target resource. Checks are comparisons with literal strings (greater than, equal to, etc.) or attempts to match the timestamp against string patterns or regular expressions.

Facets: *eq, ne, lt, le, gt, ge, like, notLike, matches, notMatches.*

Examples:

<fileDate le="2020-07-08" leMsg="File from 2020-07-08 or later"/>   
<fileDate ne="2020-04-11T00:11:17.142Z" neMsg="This file version must not be used"/>

<fileDate like="2020\*" likeMsg="File not from 2020"/>

<fileDate matches="T04|T05" matchesMsg="File not updated between 04:00 and 06:00"/>

#### FileName

FileName constraints check the name of the target resource. Checks are comparisons with literal strings (greater than, equal to, etc.) or attempts to match the name against string patterns or regular expressions.

Facets: *eq, ne, like, notLike, matches, notMatches.*

Examples:

<fileName like="airport\*.json" likeMsg="JSON files must match pattern 'airport\*.json"  
 notLike="\*test\*" notLikeMsg="JSON files must not match pattern '\*test\*'"/>

<fileName notMatches="\s"

likeMsg="Billions of dollars are wasted by using whitespace in file names"/>

#### FileSize

FileSize constraints check the file size of the target resource, measured as number of bytes. Checks are comparisons with integer numbers.

Facets: *eq, ne, lt, le, gt, ge.*

Examples:

<fileSize gt="0" gtMsg="Empty files not allowed"  
 le="1000000" leMsg="Files larger 1MB not allowed"/>

<fileSize ne="1029920" neMsg="File size unchanged"/>

#### FolderContent

FolderContent constraints check the content of a folder – presence and absence of files and the number of files matching a name pattern. Checks may include file hash keys.

Facets: *closed, minCount, maxCount, count, excludedMember*

Example:

<folderContent closed="true" ignoredMembers="xml-in\* xml-out\*">  
 <excludedMemberFile

name="airport-xxx.xml"

excludedMemberFileMsg="xxx code in production"/>  
 <memberFolder name="log" minCount="0"/>  
 <memberFile name="airport-\*.xml" maxCount="unbounded"/>  
 <memberFile name="ONLINE.FLAG"/>  
 <memberFile name="STATUS.txt"/>  
</folderContent>

#### Mediatype

Mediatype constraints check the mediatype of the target resource. It is checked if the resource has one of the mediatypes XML, JSON or CSV. In case of CSV, optional additional constraints refer to the number of rows and the numbers of columns.

Facets: *eq, csv.columnCount, csv.columnMinCount, csv.columnMaxCount,*

*csv.rowCount, csv.rowMinCount, csv.rowMaxCount.*

Examples:

<mediatype eq="json"/>

<mediatype eq="xml json"/>

<mediatype eq="csv"/>

<mediatype eq="csv" csv.separator="semicolon" csv.withHeader="true"/>  
<mediatype eq="csv" csv.separator="semicolon" csv.header="yes"   
 csv.columnCount="71" csv.columnCountMsg="Not the expected number of columns"   
 csv.rowCount="249" csv.rowCountMsg="Not the expected number of rows"/>

#### DocTree

A DocTree constraint checks the tree structure of the target resource. It may describe the complete document tree, or any number of arbitrary sub trees. The following example describes two subtrees, rooted in <temporal> and <geo> nodes found in the document tree:

<docTree>  
 <node locNP="//temporal" maxCount="unbounded" closed="true">  
 <node locNP="timezone"/>  
 <node locNP="timezoneTz"/>  
 <node locNP="dst"/>   
 </node>  
 <node locNP="//geo" maxCount="unbounded">  
 <node locNP="latitude"/>  
 <node locNP="longitude"/>  
 </node>   
</docTree>

A tree descriptor is a tree of model nodes representing:

* an instance node - <node>
* a choice between alternatives- <oneOf>
* or a group of nodes - <nodeGroup>

Note that the order of sibling nodes is irrelevant – tree structure is treated as unordered, irrespective of the mediatype (XML, JSON, …). Every node descriptor (<node>) has a navigation path (@locNP) describing how instance nodes are reached from the instance nodes described by the parent node descriptor, or from the root node if there is no parent node descriptor. By default, every node must occur exactly once. Different cardinality constraints can be expressed using attributes @count, @minCount and @maxCount.

Parent child relationships in the tree descriptions are logical, not necessarily physical, as the navigation path leading from instances of the parent node descriptor to instances of the child node descriptor is not necessarily a single step along the child axis:

<docTree>  
 <node locNP="//geo" maxCount="unbounded">  
 <node locNP="../@icao"/>  
 <node locNP="latitude"/>  
 <node locNP="longitude"/>   
 <node locNP="../temporal/timezone"/>   
 </node>   
</docTree>

Tree descriptions may stop at any point, representing a complex node without its child nodes. For example, the node descriptor

<node locNP="temporal"/>

may describe a leaf node, but it may also describe an intermediate node the child nodes of which are left unspecified.

Any node descriptor is by default open, meaning that it may contain further child nodes unrelated to the tree description. A node descriptor may be closed (@closed="true"), meaning that all child nodes of the instance nodes are described by the child node descriptors and their navigation paths. More precisely, each child node of an instance node must be matched by an initial (or only) navigation step along the child axis, occurring in the navigation path of a child node descriptor. In this example:

<node locNP="//airport" closed="true">  
 <node locNP="country"/>  
 <node locNP="geo/latitude"/>  
 <node locNP="geo/longitude"/>  
</nodeGroup>

the instance nodes are expected to have two child nodes, <country> and <geo>.

By default, the node names used in the path expressions are matched against the local names of the instance nodes. Names are interpreted as lexical QNames if <docTree> has attribute @withNamespaces equal true.

Example:

<docTree>  
 <node locNP="//temporal" maxCount="unbounded">  
 <node locNP="timezone" closed="true"/>  
 <node locNP="timezoneTz"/>  
 <node locNP="dst"/>   
 </node>  
 <node locNP="//airport" maxCount="unbounded" closed="true">  
 <node locNP="@id"/>   
 <node locNP="@icao"/>  
 <node locNP="@createdAt"/>  
 <oneOf>  
 <nodeGroup>   
 <node locNP="@iata"/>  
 <node locNP="@latitude"/>   
 <node locNP="@longitude"/>  
 <node locNP="@href"/>  
 <node locNP="@comment" minCount="0"/>  
 </nodeGroup>  
 <nodeGroup>  
 <node locNP="name"/>  
 <node locNP="city"/>  
 <node locNP="country"/>  
 <node locNP="geo/latitude"/>  
 <node locNP="geo/longitude"/>  
 <node locNP="geo/altitude"/>  
 <node locNP="temporal"/>  
 <node locNP="type"/>  
 <node locNP="source"/>   
 <node locNP="addInfo" minCount="0"/>  
 </nodeGroup>  
 </oneOf>  
 </node>   
</docTree>

#### HyperdocTree

bla

#### XsdValid

bla

#### Value

A Value constraint evaluates an expression and checks the result against expectations. Checks are concerned with the number of items, their datatype, string value and string length and whether they are distinct. Options control whether all or at least one value item must conform to expectations and whether checks are applied to raw values or edited values, e.g. cast to a type or set to lowercase.

The expression is either an [XPath](#_XPath) expression or a [LinePath](#_LinePath) expression. If the use of a [Foxpath](#_Foxpath) expression is desired, a different constraint type must be used (a [Foxvalue](#_Foxvalue) constraint).

The following table compiles the facets supported by Value constraints.

**Table. Check nodes available in Value constraints.**

|  |  |  |  |
| --- | --- | --- | --- |
| **Check node** | **Expectation** | **Example** | **Notes** |
| @count, @minCount, @maxCount | Number of value items is eq/ge/le the attribute value | count="1"  minCount="1"  maxCount="3" | There is no default – absence of count attributes means that the number of items is not constrained |
| @eq, @ne,  @lt, @le,  @gt, @ge | Value items eq/ne/lt/le/gt/ge the attribute value | eq="airport"  lt="1010" | Compares as strings, unless option @useDatatype is set |
| @like | Value items match the text pattern | like="\*/\*" | Wildcards are \* and ? |
| @notLike | Value items do not match the text pattern | notLike="airport-\*" | Wildcards are \* and ? |
| @matches | Value items match the regex | matches="^Z\d+$"  matches="-" | The regex is not anchored – it may describe a substring |
| @notMatches | Value items do not match the regex | notMatches="\s"  notMatches=",\S" | The regex is not anchored – it may describe a substring |
| @length, @minLength, @maxLength | Value items have a string length eq/ge/le the attribute value | @length="1"  @minLength="1"  @maxLength="4" | Length is the number of characters. |
| @datatype | Value items can be cast to the datatype identified by the attribute value | datatype="integer"  datatype="date"  datatype="boolean" | The attribute value must be the local name of a datatype defined by the XSD specification |
| @distinct | Value items must be distinct | distinct="true" | Compares as strings, unless option @useDatatype is set |
| in/eq  in/ne  in/matches  in/notMatches  in/like  in/notLike | Value items must match at least one of the elements | <in>  <eq>Active</eq>  <eq>NonActive</eq>  <like>Custom-\*</like>  </in> | Each child element of <in> defines an alternative; the semantics of the elements is equal to the semantics of an attribute with the same name and the same value |
| notin/eq  notin/ne  notin/matches  notin/notMatches  notin/like  notin/notLike | Value items must not match any of the elements | <notin>  <like>Test-\*</like>  <like>Debug-\*</like>  <matches>\s</matches>  </notin> | Each child element of <notin> defines a case which must not apply |
| contains/term | For every term there must be a value item with that value | <contains>  <term>Summary</term  <term>References</term>  </contains> | The expression value may contain also other items |
| sameTerms/term | For every item there is an equal term, and for every term there is at least one equal item | <sameTerms>  <term>FRA</term>  <term>CGN</term>  <term>DUS</term>  </sameTerms> | The order of terms and items may be different, and there may be different numbers of repetition |
| deepEqual/term | The n-th item is equal to the n-th term | <deepEqual>  <term>Entrance</term>  <term>Exit></term>  <term>Exit></term>  </deepEqual> | Corresponds to the XPath function deep-equal() |

The evaluation may be modified by several options – see following table.

**Table. Check nodes available in Value constraints.**

|  |  |  |  |
| --- | --- | --- | --- |
| **Option node** | **Semantics** | **Example** | **Notes** |
| @quant | If used with the value some, conformance requires only at least one item to meet the expectation, not all | quant="some"  quant="all" | Not evaluated in the case of the following checks: distinct, sameTerms, deepEqual |
| @useDatatype | Before comparing, value items are cast to the schema type identified by the attribute value | useDatatype="integer" | The attribute value must be the local name of a datatype defined by the XSD specification |
| @useString | Before comparing, value items are edited; available manipulations:  lc – set lower-case,  uc – set upper-case,  ns – normalize space,  tr – trim leading/trailing WS  sv – take string value | useString="lc"  useString="lc ns" | The attribute value is a whitespace separated list of tokens identifying manipulations |
| @flags | The attribute value supplies flags used when evaluating regular expressions or string patterns | flags="i"  flags="x" | Flag semantics as described in the XPath functions spec ([XP flags](https://www.w3.org/TR/xpath-functions-31/#flags)); “x” can be useful when ignorable whitespace makes the regex more readible |

The expression can be an [XPath](#_XPath) expression (@exprXP) or a [LinePath](#_LinePath) expressions(@exprLP; @exprLP + @filterLP).

A few examples illustrate the use of Value constraints.

Example - check items using a text pattern:

<value exprXP="//temporal/timezoneTz"   
 count="1" countMsg="Missing value: timezoneZt"  
 like="\*/\*" likeMsg="timezoneTz should have \*/\*."/>

Example – check items using a numerical comparison:

<value exprXP="//altitude"   
 count="1" countMsg="Missing value: timezoneZt"  
 lt="1100" likeMsg="Altitude expected to be lt 1000."  
 useDatatype="integer"/>

Example – check items as edited values:

<value exprXP="//type"   
 count="1" countMsg="Missing value: timezoneZt"  
 eq="AIRPORT" eqMsg="Type must be AIRPORT."  
 useString="uc"/>

Example – check items as edited values:

<value exprXP="//type"   
 count="1" countMsg="Missing value: timezoneZt"  
 eq="AIRPORT" eqMsg="Type must be AIRPORT."  
 useString="uc"/>

Example – at least one value item must conform, rather than all items:

<value exprXP="//altitude"   
 minCount="1" minCountMsg="Missing values: altitude"  
 lt="10" ltMsg="Airport at altitude lt 10 expected."  
 quant="some"  
 useDatatype="integer"/>

Example – comparing value items with alternatives:

<value exprXP="//dst"   
 count="1" countMsg="Missing value: dst">  
 <in>  
 <eq>E</eq>  
 <eq>N</eq>  
 <eq>U</eq>  
 <like>X-\*</like>  
 </in>  
</value>

#### ValuePair

A ValuePair constraint evaluates two expressions and checks their relationship against expectations. Available checks include pair-wise and aggregated comparisons between the value items of both expressions, as well as comparisons between the value item counts. The following table compiles the facets supported by Value constraints.

**Table. Check nodes available in ValuePair constraints.**

|  |  |  |  |
| --- | --- | --- | --- |
| Check node | Expectation | Example | Notes |
| @cmp=  eq|ne|lt|le|gt|ge | The items of the first expression value are eq/ne/lt/le/gt/ge the items of the second second expression value;  (3) When the attitems of the first expression value are | cmp="eq"  cmp="ne"  cmp="lt"  cmp="le"  cmp="gt"  cmp="ge" | Compares as strings, unless option @useDatatype is set |
| @cmp=in | Every item from the first expression value is equal to some item of the second expression value | cmp="in" | The value of the second expression may contain items not  equal to any item from the first expression value |
| @cmp=notin | The expression values are disjunct – no item is found in both values | cmp="notin" |  |
| @cmp=contains | Every item from the second expression value is equal to some item of the first expression value | cmp="contains" | The value of the first expression may contain items not equal to any item from the second expression value |
| @cmp=sameTerms | All items from the first expression value are found among the items of the second expression value, and the other way around | cmp="sameTerms" | Every item is found in both values, but order and the numbers of repetition may be different |
| @cmp=deepEqual | The n-th item of the first expression value is equal to the n-th item of the second expression value | cmp="deepEqual" | Corresponds to the XPath function deep-equal() |
| @cmpCount=  eq|ne|lt|le|gt|ge | The number of items of the first expression is eq/ne/lt/le/gt/ge the number of items of the second expression | cmpCount="eq"  cmpCount="lt"  cmpCount="ge" | Compares the item counts, not the items themselves |
| @count1, @minCount1, @maxCount2 | The number of value items of the first expression is eq/ge/le the attribute value | count1="1"  minCount1="1"  maxCount1="99" | There is no default – absence of count attributes means that the number of items of the first expression is not constrained |
| @count2, @minCount2, @maxCount2 | The number of value items of the second expression is eq/ge/le the attribute value | count2="1"  minCount2="1"  maxCount2="99" | There is no default – absence of count attributes means that the number of items of the second expression is not constrained |

The evaluation may be modified by several options – see following table.

**Table. Check nodes available in ValuePair constraints.**

|  |  |  |  |
| --- | --- | --- | --- |
| **Option node** | **Semantics** | **Example** | **Notes** |
| @quant | If used with the value some, conformance requires only at least one item of the first expression value to meet the expectation, not all.  If used with the value someForEach, conformance requires for each item of the first expression at least one item of the second expression to meet the expectation. This variant can be useful when the second expression is re-evaluated for each item of the first expresion (see below, option expr2Context) | quant="some"  quant="someForEach"  quant="all" | Not evaluated in the case of the following @cmp values: sameTerms, deepEqual |
| @useDatatype | Before comparing, value items are cast to the schema type identified by the attribute value | useDatatype="integer" | The attribute value must be the local name of a datatype defined by the XSD specification |
| @useString | Before comparing, value items are edited; available manipulations:  lc – set lower-case,  uc – set upper-case,  ns – normalize space,  tr – trim leading/trailing WS  sv – take string value | useString="lc"  useString="lc ns" | The attribute value is a whitespace separated list of tokens identifying manipulations |
| @expr2Context | If used with the value item, the second expression is evaluated repeatedly, once per item of the first expression value and using that item as context item | expr2Context="item"  expr2Context="context1" | If the value is context1, the second expression is evaluated only once, using the same context item as used for the first expression |

The expressions can be [XPath](#_XPath) expressions (@expr1XP, @expr2XP), [LinePath](#_LinePath) expressions (@expr1LP, @expr2LP; @filter1LP + @map1LP, @filter2LP + @map2LP) or a combination of both – e.g. @expr1XP, @expr2LP.

A few examples illustrate the use of Value constraints.

Example – the value items must all be equal:

<valuePair expr1XP="/airportsForCountry/@country" count1="1"  
 expr2XP="//airport/country" minCount2="1"  
 cmp="eq" cmpMsg="Inconsistent country names"/>

Example – referenes must be a subset of IDs:

<valuePair expr1XP="//@country" minCount1="1"  
 expr2XP="//country/@id" minCount2="1"  
 cmp="in" cmpMsg="Country references not a subset of country IDs"/>

Example – the second expression is evaluated for each item of the first expression value:

<valuePair expr1XP="//country/@name" minCount1="1"  
 expr2XP="../name" minCount2="1"  
 expr2Context="item"  
 cmp="eq" cmpMsg="Country name attribute and child different"/>

#### Foxvalue

A Foxvalue constraint is very similar to a Value constraint: a resource is mapped to a value which is checked against expectations. The only difference is the kind of expression, which here is a Foxpath expression, rather than an XPath or LinePath expression. The constraints are nevertheless distinguished in order to emphasize a difference of meaning: while a Value constraint checks resource contents and is independent of other resources, a Foxvalue constraint checks a resource in terms of its environment, usually involving file system navigation and the inspection of other resources. A Foxvalue constraint is often used for checking folder contents, but it may also be applied to files, e.g. verifying that a resource is accompanied by another resource with particular content.

The check nodes and option nodes of a Foxvalue constraint are identical to the check nodes and option nodes of a [Value constraint](#_Value).

Example: check the distinctness of items found in folder contents.

<folder uri="data/air/airports/countries">   
 <foxvalue   
 exprFOX="\*\xml\airport-\*.xml/airport/@id" minCount="10"  
 distinct="true" distinctMsg="XML Airport IDs not distinct"/>   
</folder>

Example: check integer values found in folder contents for a limit value:

<folder uri="data/air/airports/countries">   
 <foxvalue   
 exprFOX="\*\json\airport-\*.json\jdoc()//altitude" minCount="10"  
 useDatatype="integer"  
 lt="1100" ltMsg="Altitude lt 1100 expected"/>   
</folder>

#### FoxvaluePair

bla

#### ValueCompared

A ValueCompared constraint is similar to a ValuePair constraint, as two expressions are evaluated and the relationship between their values is checked. But while the expressions of a ValuePair constraint are both evaluated in the context of the shape target, a ValueCompared constraint has one expression evaluated in the shape target and the other expression evaluated in the context of a second resource. A ValueCompared constraint thus checks a pair of resources, rather than a single resource.

The second resource is determined by evaluating a [Link definition](#_Link_resolution). The expression values are checked in exactly the same way as in a ValuePair constraint – see [checknodes of a ValuePair constraint](#table_check_nodes_valuepair).

The link definition can be referenced by name (@linkName) or specified locally by [link defining attributes](#table_connector_types).

Example: a resource pair is validated by checking different values for equality:

<valuesCompared   
 navigateFOX="..\..\json\fox-child($fileName, '.xml$', '.json')"  
 mediatype="json"  
 countTargetResources="1">   
 <valueCompared   
 expr1XP="//latitude" count1="1"  
 expr2XP="//latitude" count2="1"  
 cmp="eq" cmpMsg="Latitude different in XML and JSON airports"/>   
 <valueCompared   
 expr1XP="/\*/@icao" count1="1"  
 expr2XP="//icao" count2="1"  
 cmp="eq" cmpMsg="ICAO codes different in XML and JSON airports"/>   
</valuesCompared>

The second resource is determined by a local link definition, defined by attributes (@navigateFOX, @mediatype) on <valuesCompared>. Alternatively, the link definition may be referenced by name (@linkName):

<linkDef name="myJSON"  
 navigateFOX="..\..\json\fox-child($fileName, '.xml$', '.json')"  
 mediatype="json"/>  
…

<valuesCompared   
 linkName="myJSON"  
 countTargetResources="1">   
 …

</valuesCompared>

When the link definition yields multiple link target, a binary constraint like ValueCompared is evaluated for each pair consisting of the shape target and one of the link targets:

<valuesCompared hrefXP="//@href"  
 minCountTargetResources="1">   
 <valueCompared   
 expr1XP="/airportsForCountry/@country" count1="1"  
 expr2XP="//country" count2="1"  
 cmp="eq" cmpMsg="Countries inconsistent"/>   
</valuesCompared>

#### FoxvalueCompared

bla

#### DocSimilar

bla

#### FolderSimilar

bla

#### Link

bla

#### TargetSize

bla

#### Conditional

bla

### Constraint types - examples

The folder demo-constraints contains for each constraint type one or several example schemas. Note that the examples are not meant to give a comprehensive overview of the possibilites. Rather, they should give a feeling what can be achieved using that constraint type, and how using the constraint type looks.

## Outlook

[under construction]