## Getting started with greenfox

This section illustrates the development of a greenfox schema designed for validating a file system tree against a set of expections. Such a validation can also be viewed as validation of the system “behind” the file system tree, represented by its contents.

### The system – system S

Consider **system S** – an imaginary system which is a collection of web services. We are going to validate a *file system representation* which is essentially a set of test results, accompanied by resources supporting validation (XSDs, codelists and data about expected response messages). The following listing shows a file system tree which is a representation of system S, as observed at a certain point in time:

**system-s**

. resources

. . **codelists**

. . . *codelist-foo-article.xml*

. . **xsd**

. . . *schema-foo-article.xsd*

. testcases

. . **test-t1**

. . . config

. . . . *msg-config.xml*

. . . input

. . . . *getFooRQ\*.xml*

. . . output

. . . . *getFooRS\*.xml*

. . **+test-t2 (contents: see test-t1)**

. . usecases

. . . usecase-u1

. . . . usecase-u1a

. . . . . **+test-t3 (contents: see test-t1)**

The concrete file system tree must be distinguished from the *expected file system tree*, which is described by the following rules.

|  |  |  |
| --- | --- | --- |
| **File or folder** | **Name or pattern** | **Expectation** |
| folder | codelists | Contains one or more codelist files |
| folder | codelists/\* | A codelist file; name not constrained; must be an XML document containing <codelist> elements with a @name attribute and <entry> children |
| folder | xsd | Contains one or more XSDs describing services messages |
| file | xsd/\* | An XSD schema file; name not constrained |
| folder | test-\* | A test case folder, containing input, output and config folders; apart from these only optional log-\* files are allowed |
| folder | config | Test case config folder, containing file msg.config.csv |
| file | msg.config.csv | A CSV file with three columns: request file name, response file name, expected return code |
| folder | input | Test case input folder, containg request messages |
| file | input/\* | A file representing a request message; name extension .xml or .json; mediatype corresponding to name extension |
| folder | output | A test case output folder, containing files representing response messages |
| file | output/\* | A file representing a response message; name extension .xml or .json; mediatype corresponding to name extension |

The number and location of testcase folders (test-\*) are unconstrained. This means that the testcase folders may be grouped and wrapped in any way, although they must not be nested. So the use of a testcases folder wrapping all testcase folders - and the use of usecase-\* folders adding additional substructure - is allowed, but must not be expected. The placing of XSDs in folder resources/xsd, on the other hand, is obligatory, and likewise the placing of codelist documents in folder resources/codelists. The names of XSD and codelist files are not constrained.

Structural expectations include also a conditional constraint:

* For every request message, there must be a response message with a name obtained by replacing in the request file name RQ with RS (e.g. getFooRQ and getFooRS)

Besides the structural expectations, there are also content-related expectations:

* For every response message in XML format, there is exactly one XSD against which it can be validated
* Every response message in XML format is valid against the appropriate XSD
* Response message items with name fooValue must be found in the codelist with name foo-article (applies to XML and JSON responses alike)
* Response message return codes must be as configured by the corresponding row in msg-config.csv (applies to XML and JSON responses alike)

### Building greenfox schema “system S”

Now we create a greenfox schema which enables us to validate the file system against these expectations. An initial version only checks the existence of non-empty XSD and codelists folders:

<greenfox greenfoxURI="http://www.greenfox.org/ns/schema-examples/system-s"  
 xmlns="http://www.greenfox.org/ns/schema">  
   
 <domain path="\tt\greenfox\resources\example-system\system-s" label="system-s">   
   
 <!-- \*\*\* System root folder shape \*\*\* -->  
 <folder foxpath="." id="systemRootFolderShape">  
   
 <!-- \*\*\* XSD folder shape -->  
 <folder foxpath=".\\resources\xsd" id="xsdFolderShape">  
 <targetSize msg="No XSD folder found" count="1"/>  
 <file foxpath="\*.xsd" id="xsdFileShape">  
 <targetSize msg="No XSDs found" minCount="1"/>  
 </file>  
 </folder>   
  
 <!-- \*\*\* Codelist folder shape -->  
 <folder foxpath=".\\resources\codelists" id="codelistFolderShape">  
 <targetSize msg="No codelist folder found" count="1"/>  
 <file foxpath="\*[is-xml(.)]"id="codelistFileShape">  
 <targetSize msg="No codelist files found" minCount="1"/>  
 </file>  
 </folder>   
   
 </folder>  
 </domain>   
</greenfox>

The <domain> element represents the root folder of a file system tree to be validated, which has a filepath as specified by the @path attribute.

A <folder> element represents the set of folders matching the foxpath expression given by its @foxpath attribute, which is its *target declaration*. Foxpath [1] is an extended version of XPath 3.0 which supports file system navigation, node tree navigation and a mixing of file system and node tree navigation within a single path expression. Note that file system navigaton steps are preceded by a backslash operator, rather than a slash, which is used for node tree navigation steps. The foxpath expression is evaluated in the context of a folder selected by the target declaration of the containing <folder> element (or <domain>, if there is no containing <folder>). Evaluation “in the context of a folder” means that the initial context item is the filepath of that folder, so that relative file system path expressions are resolved in this context (see x.x for details). For example, the expression

.\\resources\xsd

resolves to the xsd folders contained by a resources folder found at any depth under the context folder, which is …\system-s. Similarly, a <file> element represents the set of files selected by the foxpath expression given by its @foxpath attribute and resolved in the context of a folder selected by the parent <folder>’s target declaration.

A <folder> element represents a **folder shape**, which is a set of **constraints** which apply to a **target**, which is a (possibly empty) set of folders. When a <folder> has a @foxpath attribute, the target is the set of folders selected by the expression. The constraints are declared by child elements of the shape element.

Likewise, a <file> element represents a **file shape**, defining a set of constraints which apply to a target which is a set of files. When a <file> has a @foxpath attribute, the target is the set of files selected by the expression. Folder shapes and file shapes are collectively called **resource shapes**.

The expected number of folders or files belonging to the target of a shape can be expressed by declaring a **constraint**. A constraint has a kind (called the **constraint component IRI**) and a set of arguments passed to the **constraint parameters**. For every kind of constraint, a characteristic set of mandatory and optional constraint parameters is defined in terms of name, type and cardinality. In a schema document, a constraint is either declared by a *constraint element* or by *constraint attributes* attached to an element representing a shape. Here, we declare a TargetSize constraint, which is represented by a <targetSize> child element of a file or folder shape. The element has three optional attributes, @minCount, @maxCount and @count, representing three optional constraint parameters. A constraint can be thought of as a function which consumes constraint parameter values and a resource value – a value representing the resource being validated; and which returns a validation result. Here, the resource value is the number of target resources selected, and the constraint parameter minCount is set to the value “1”. If the constraint is violated, the validation result is a <gx:red> element which contains the message specified by @msg on the constraint element, along with a set of information items identifying the violating resource (@folder), the constraint (@constraintComp and @constraintID) and its parameter values (@minCount). Example result:

<gx:red folder="C:/tt/greenfox/resources/example-system/system-s/resources/xsd"

shapeID="xsdFolderShape"

childShapeID="xsdFileShape"

childShapeTargetFoxpath="\*.xsd"   
 constraintComp="TargetSize"   
 constraintID="TargetSize\_2"   
 minCount="1"   
 value="0"   
 msg="No XSDs found"/>

A key principle of greenfox is that every constraint belongs to a resource shape and is applied to each resource in the target of that shape, referred to as the **focus resource**. In the common case, the focus resource is in the target of the nearest ancestor resource shape (<file> or <folder>). The <targetSize> constraint is an exception of the rule where the focus resource is *not* from the target of the containing <folder> or <file> shape, but the context folder used when evaluating the target declaration of that shape; usually this is a folder from the target of the “grand parent folder element”, selected by the XPath $targetSize/parent::\*/parent::folder).

In a second step we extend our schema with a folder shape whose target consists of all *testcase folders*:

<!-- \*\*\* Testcase folder shape \*\*\* -->

<folder foxpath=".\\test-\*[input][output][config]" id="testcaseFolderShape">  
 <targetSize msg="No testcase folders found" minCount="1"/>  
 <folderContent msg="Testcase contains member other than input, output, config, log-\*."   
 closed="true">  
 <memberFolders names="input, output, config"/>  
 <memberFiles names="log-\*" occ="\*"/>  
 </folderContent>  
 …  
</folder>

The target includes all folders found at any depth under the current context folder (system-s), matching the name pattern test-\* and having (at least) three members input, output and config. The <targetSize> constraint checks that the system contains at least one such folder. The <folderContent> constraint is checked for each folder in the target of the containing <folder> shape – in other words, for each testcase folder. The constraint disallows any additional members except for *optional* files with a name matching log-\* (of which any number is allowed, note the @occ attribute). The folderContent constraint is an example for a constraint component defining *complex* constraint parameters: for example, values supplied to the memberFolders parameter (which can accept any number of values) have a names and an (optional) occ field.

We proceed with a file shape whose target is the msg-config.csv file in the config folder of the test case:

<!-- \*\*\* msg config file shape -->  
 <file foxpath="config\msg-config.csv" ... id="msgConfigFileShape">  
 <targetSize msg="Config file missing" count="1"/>

...  
 </file>

As explained above, the <targetSize> constrains the focus resource from the file shape’s parent shape, which is the shape of testcase folders. For any testcase folder which does not (indirectly) contain a msg-config.csv file, a constraint violation will be reported.

We want to be more specific: the file must be a CSV file, and the third column (which according to the header row is called returnCode) must contain a value which is OK or NOFIND or matches the pattern ERROR\_\*. We add attributes to the <file> element which specify how to parse the CSV file into an XML representation (@mediatype, @csv.separator, @csv.header). As with other non-XML mediatypes (e.g. JSON), an XML view enables us to leverage XPath and *express* a selection of content items. An <xpath> constraint associates the selection of items (@expr) with a description of expected values (in this case: <in> child element):

<!-- \*\*\* msg config file shape -->  
 <file foxpath="config\msg-config.csv" id="msgConfigFileShape"

**mediatype="csv" csv.separator="," csv.withHeader="yes"**>  
 <targetSize msg="Config file missing" count="1"/>  
   
 <!-- Check - configured return codes ok? -->  
 <xpath msg="Config file contains unknown return code" expr="**//returnCode**">  
 <in>  
 <eq>OK</eq>  
 <eq>NOFIND</eq>  
 <like>ERROR\_\*</like>  
 </in>   
 </xpath>   
 </file>

Note that the XPath expression (given by @expr) is evaluated in the context of the document node of the document obtained by parsing the file. Here comes an example of a conformant message definition file:

request,response,returnCode  
getFooRQ1.xml,getFooRS1.xml,OK  
getFooRQ2.xml,getFooRS2.xml,NOFIND

getFooRQ3.xml,getFooRS3.xml,ERROR\_SYSTEM  
  
while this example violates the constraint:

request,response,returnCode  
getFooRQ1.xml,getFooRS1.xml,OK

getFooRQ2.xml,getFooRS2.xml,NOFIND  
getFooRQ3.xml,getFooRS3.xml,ERROR-SYSTEM

We proceed to check request message files: for each such file, there must be a response file in the sibling <output> folder, with a name obtained by replacing in the requests file name the last substring RQ with RS. The constraint is expressed by the following file shape containing a foxpath constraint:

<!-- \*\*\* Request file shape \*\*\* -->

<file foxpath="input\\*" id="requestFileShape">  
 <targetSize msg="Input folder without request msgs" minCount="1"/>  
   
 <!-- Check - request with response ? -->  
 <foxpath msg="Request without response" count="1" expr="  
 let $expFileNameRS := file-name(.) ! replace(., '(.\*)RQ(.\*)\.(xml|json)$', '$1RS$2.$3')   
 return ..\..\output\\*[file-name(.) eq $expFileNameRS]" />  
 </file>

A foxpath constraint comprises a foxpath expression which maps the focus resource to a value and a set of constraint facets to be applied to the value. Here we have a single facet, which constrains the expression value to consist of exactly one item (@count=1). A closer inspection of the foxpath expression reveals a navigation starting at the context item (..\..\output…). When evaluating the foxpath expression of a foxpath constraint, the initial context item is the file path of the focus resource (here: the file path of a request file under investigation).

A foxpath constraint can also be defined by a folder shape. We apply this approach in order to constrain the codelists folder to contain <codelist> elements with a @name attribute and at least one non-empty <entry> child:

<!-- \*\*\* Codelist folder -->  
 <folder foxpath=".\\resources\codelists" id="codelistFolderShape">  
 <targetSize msg="No codelist folder found" count="1"/>

<**foxpath** expr="**\*.xml/codelist[entry/@code/string()]/@name**" minCount="1"/>

...  
 </folder>

Note the aggregative view enable by the foxpath language: we do not bother with individual files but perform a “mixed” navigation, starting with file system navigation to all \*.xml files, continuing within their collected content, arriving at <codelist> elements.

Now we turn to the files representing response messages. They must be “fresh”, that is, have a timestamp of last modification which is after a timestamp specified by a call parameter of the validation operation. This is accomplised by a <lastModified> constraint:

<!-- \*\*\* Response file shape \*\*\* -->   
 <file foxpath="output\(\*.xml, \*.json)" mediatype="xml-or-json">  
 <targetSize msg="Output folder without request msgs" minCount="1"/>  
   
 <!-- \*\*\* Check - response fresh? \*\*\* -->  
 <lastModified msg="Stale output file" ge="${lastModified}"/>  
 ...

</file>

The placeholder ${lastModified} is substituted by the value supplied to the greenfox processor as input parameter and declared (and defaulted) in the schema as a context parameter:

<greenfox ... >

<!-- \*\*\* External context \*\*\* -->  
 <context>  
 <field name="**lastModified**" value="2019-12-07"/>  
 </context>   
 ...

</greenfox>

We have numerous expecations related to the contents of response files. If the response is an XML document (rather than JSON), it must be valid valid against some XSD found in the XSD folder. XSD validation is triggered by a <xsdValid> constraint, with a foxpath expression locating the XSD(s) to be used:

<!-- \*\*\* Response file shape \*\*\* -->   
<file foxpath="output\(\*.xml, \*.json)" mediatype="xml-or-json">  
 ...

<!-- \*\*\* Check - schema valid? (only if XML) -->  
 <ifMediatype eq="xml">  
 <**xsdValid** msg="Response msg not XSD valid"   
 **xsdFoxpath**="$domain\resources\xsd\\\*.xsd"/>   
 </ifMediatype>  
</file>

It is not necessary to specify an individual XSD – the greenfox processor inspects all XSDs matching the expression and selects for each document to be validated the appropriate XSD element declaration. (If not exactly one element declaration is found, an error is reported.) Note the variable reference $domain, which can be referenced in any XPath or foxpath expression and which points to the domain folder.

The next constraint checks if certain values from the response are found in a particular codelist.

<!-- \*\*\* Check - known article number? -->  
<xpath msg="Unknown foo article number"  
 expr="//\*:fooValue"   
 inFoxpath="$domain\\codelists\\*.xml/codelist[@name eq 'foo-article']/entry/@code"/>

Note the @inFoxpath attribute: the value of @expr is not compared with a literal value, but with the value retrieved by an *expression*. The value is the content of a codelist, retrieved by a foxpath expression navigating across the file system into collected folder contents:

$domain\\codelists\\*.xml/codelist[@name eq 'foo-article']/entry/@code

In order to check the return code, we must first read it from the document being checked, then navigate to the message config of the current test case, which is a CSV file, and retrieve the expected return code as the value of the third column (named returnCode) in the row where the second column (named response) matches the file name of the response. Such an operation can be expressed using foxpath:

<!-- \*\*\* Check - return code ok? \*\*\* -->  
<foxpath msg="Return code not the configured value" eq="true" expr="  
 let $actReturnCode := $doc//\*:returnCode  
 let $expReturnCode := ..\..\config\msg-config.csv\csv-doc(., ',', 'yes')   
 //record[response eq $fileName]/returnCode   
 return $actReturnCode = $expReturnCode"/>

<!-- \*\*\* Check - return code ok? \*\*\* -->   
<foxpath msg="Return code not the configured value" eq="true" expr="  
 let $actReturnCode := doc(.)//\*:returnCode  
 let $expReturnCode := ..\..\config\msg-config.csv\csv-doc(., ',', 'yes')   
 //record[response eq $fileName]/returnCode   
 return $actReturnCode = $expReturnCode"/>

The complete schema is shown in the appendix A2. To summarize, we have developed a schema which constrains the presence and contents of folders, the presence and contents of files, and in particular relationships between contents of different files, in some cases belonging to different mediatypes.

It demonstrates several basic features of the greenfox language:

* Folders and files are validated against resource shapes, which are folder and file shapes, respectively
* The resources validated against a shape are called its focus nodes
* A resource shape may have a target declaration which selects a set of focus resources, called a target of the shape
* A target declaration may be a foxpath expression
* The focus nodes of a shape may include resource which are not in the target of the shape, but whose validation against the shape is prescribed by a constraint (not shown in the example)
* The constraints are usually represented by child elements of the shape element
* An exception is the targetSize constraint, which is a child element of a child element of the shape)
* Constraints can apply to resource properties like the last update date
* Constraints can apply to a “resource value”, which is a value to which the resource has been mapped by an expression (XPath or foxpath)
* A value shape comprises an expression mapping the focus resource to a value, and a set of constraints which apply to the value
* The heterogeneity of mediatypes can be hidden by a unified representation as XDM node tree
* Navigation within resources (their node tree representations) and navigation among resources (file system navigation) can be unified by the use of foxpath expressions