## Getting started with greenfox

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 an example 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, as a testcase folder has exactly three members: folders input, output and config with contents as described by the table. So the use of a testcases folder wrapping all testcase folders - and the use of usecase-\* folders adding additional substructure - is possible, 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 must be exactly one XSD containing an element declaration governing the message root element
* Every response message in XML format can be validated against the appropriate XSD successfully
* In response messages (XML or JSON format), items with name fooValue must be found in the codelist with name foo-article
* In response messages (XML or JSON format), the return code must be as configured by the corresponding row in msg-config.csv

Now we create a greenfox schema. The first 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 file path as specified by the @path attribute.

A <folder> element represents the set of folders matching the foxpath expression given by its @foxpath attribute. 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. 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 the containing folder (or domain, if there is no containing <folder>), so that 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 matching the foxpath expression given by its @foxpath element.

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.

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.

The number of folders or files belonging to the target of a shape can be constrained by a <targetSize> child element, using attributes @minCount, @maxCount and @count. If the constraint is violated, the validation result includes a <gx:red> element which contains the message specified by @msg on the constraint element. The following snippet is from a validation report using a report format which groups errors by the resource on which they occur:

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

navigationPath="\*.xsd"   
 constraintComp="targetCount"   
 constraintID="targetSize\_2"   
 minCount="1"   
 actCount="0"   
 msg="No XSDs found"/>

A key principle of greenfox is that every constraint belongs to a shape and is applied to each resource in the target of that shape, referred to as the **focus resource**. The <targetSize> constraint is an uncommon case where the focus resource is *not* from the target of the containing <folder> or <file> shape, but it is the folder in whose context the foxpath expression was evaluated. Any constraint checking produces a **validation result**, which identifies the focus resource by its file path (@folder or @file) and associates it with information about the constraint and, in case of a violation, further information related to the violation.

The constraintID is an ID assigned by the greenfox processor. It can be used to locate the definition of the violated constraint in an augmented copy of the original schema, which is part of the validation results. This means that any information included in the definition of a constraint can be associated with the focus resource of the constraint violation.

In a second step we extend our schema with a folder shape whose target consists of all test case 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"/>  
 <memberFile name="log-\*" occ="\*"/>  
 </folderContent>  
 …  
</folder>

The target includes all folders found at any depth under the current context (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-\*.

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