Scraping HTML with XPath

Stéphane Ducasse and Peter Kenny

Square Bracket tutorials
September 28, 2017
master @ a0267b2

Copyright 2017 by Stéphane Ducasse and Peter Kenny.

The contents of this book are protected under the Creative Commons Attribution-ShareAlike 3.0 Unported license.

You are **free**:

- to **Share**: to copy, distribute and transmit the work,
- to **Remix**: to adapt the work,

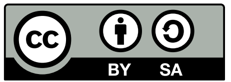
Under the following conditions:

Attribution. You must attribute the work in the manner specified by the author or licensor (but not in any way that suggests that they endorse you or your use of the work).

Share Alike. If you alter, transform, or build upon this work, you may distribute the resulting work only under the same, similar or a compatible license.

For any reuse or distribution, you must make clear to others the license terms of this work. The best way to do this is with a link to this web page:
http://creativecommons.org/licenses/by-sa/3.0/

Any of the above conditions can be waived if you get permission from the copyright holder. Nothing in this license impairs or restricts the author's moral rights.



Your fair dealing and other rights are in no way affected by the above. This is a human-readable summary of the Legal Code (the full license):
http://creativecommons.org/licenses/by-sa/3.0/legalcode

Layout and typography based on the `sbabook` LaTeX class by Damien Pollet.

Contents

Illustrations **ii**

1 Little Journey into XPath **3**
1.1 Getting started 3
1.2 An example 3
1.3 Accessing a tree object 4
1.4 Nodes and atomic values 5
1.5 Basic tree relationships 5
1.6 A large example 6
1.7 Node selection 6
1.8 Predicates 8
1.9 Selecting Unknown Nodes 10
1.10 Handling multiple queries 10
1.11 XPath axes 10
1.12 Conclusion 11

2 Scraping HTML **13**
2.1 Getting started 13
2.2 Define the Problem 13
2.3 First find the required data 15
2.4 Going back to our problem 16
2.5 Turning the pages 18
2.6 Conclusion 19

3 Scraping Magic **21**
3.1 Getting a tree 21
3.2 First the card visual 21
3.3 Revisiting it 23
3.4 Getting data 24
3.5 Conclusion 26

Illustrations

1-1 http://gatherer.wizards.com/Pages/Card/Details.aspx?multiverseid=389430. 4
1-2 Grabbing and playing with a tree. 5
1-3 Select the raw tab and click on self in the inspector. 7

2-1 Food list. 14
2-2 Food details - Salted Butter. 15
2-3 Navigating the XML document inside the inspector. 16
2-4 Sample of JSON output. 18

3-1 http://gatherer.wizards.com/Pages/Card/Details.aspx?multiverseid=389430. 22
3-2 Exploring images. 22
3-3 Exploring images. 23
3-4 Narrowing the node. 23
3-5 Exploring the class API on the spot: looking to see if there is a attribute something method. 24
3-6 Getting the card visual inside Pharo. 24
3-7 Getting the card information. 25

I came with the idea of this booklet thank to Peter that kindly answered a question on the Pharo mailing-list. To help Peter showed to a Pharoer how to scrap the web site mentioned in Chapter 2 using XPath. In addition, some years ago I was maintaining Soup a scraping framework because I want to write an application to manage my magic cards. Since then I always wanted to try XPath and in addition I wanted to offer this booklet to Peter. Why because I asked Peter if he would like to write something and he told that he was at a great age where he would not take any commitment. I realised that I would like to get as old as him and be able to hack like a mad in Pharo with new technology. So this booklet is a gift to Peter, a great and gentle Pharoer.

Stef

CHAPTER 1

Little Journey into XPath

XPath is the de factor standard language to represent queries to identify nodes in an xml structure. In this chapter we will go through the main concepts and show some of the way we can access nodes in a xml document. All the expressions can be executed on the spot so do not hesitate to experiment with them.

1.1 Getting started

You should load the XML parser and XPath library as follows:

```
Gofer it
   smalltalkhubUser: 'PharoExtras' project: 'XMLParserHTML';
   configurationOf: 'XMLParserHTML';
   loadStable.
```

```
Gofer it
   smalltalkhubUser: 'PharoExtras' project: 'XPath';
   configurationOf: 'XPath';
   loadStable.
```

1.2 An example

As an example we will take the possible representation of Magic cards. Here is for example how we can represent Arcane Lighthouse that you can see at http://gatherer.wizards.com/Pages/Card/Details.aspx?multiverseid=389430 and is shown in Figure 1-1.

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

<cardset>
  <card>
    <cardname lang="en">Arcane Lighthouse</cardname>
    <types>Land</types>
    <year>2014</year>
  <rarity>Uncommon</rarity>
  <expansion>Commander 2014</expansion>
    <cardtext>Tap: Add 1 uncolor to you mana pool.
  1 uncolor + Tap: Until end of turn, creatures your opponents
   control lose hexproof and shroud and can't have
   hexproof or shroud.</cardtext>
  </card>
</cardset>
```

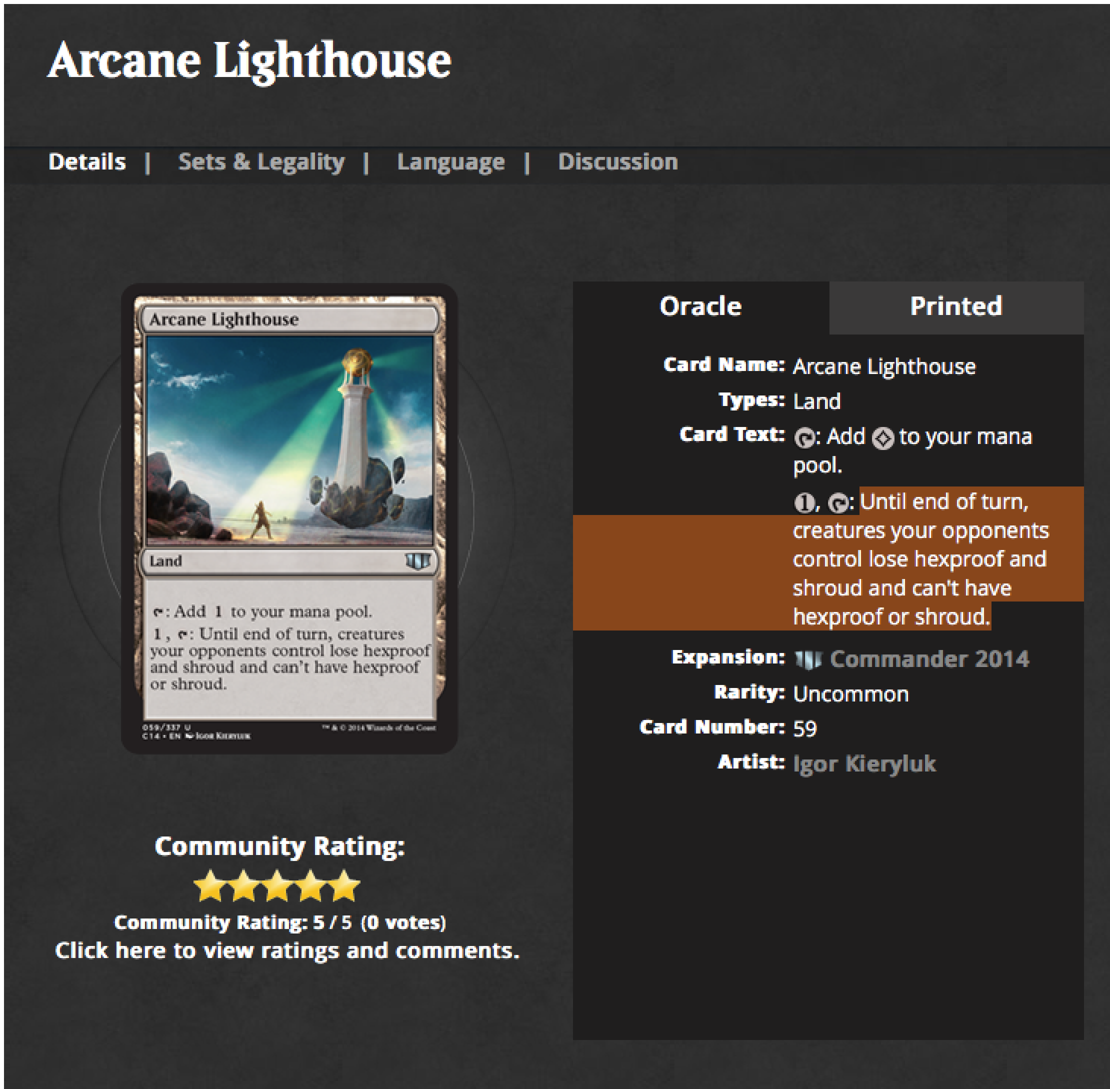


Figure 1-1 http://gatherer.wizards.com/Pages/Card/Details.aspx?multiverseid=389430.

1.3 Accessing a tree object

In Pharo it is always powerful to get an object and interact with it. So let us do that now using the `XMLDOMParser`. Note that the escaped the `'` with an extra quote as in `can''t`.

```
| tree |
tree := (XMLDOMParser on:
'<?xml version="1.0" encoding="UTF-8"?>

<cardset>
  <card>
    <cardname lang="en">Arcane Lighthouse</cardname>
    <types>Land</types>
    <year>2014</year>
    <rarity>Uncommon</rarity>
    <expansion>Commander 2014</expansion>
    <cardtext>Tap: Add 1 uncolor to you mana pool.
  1 uncolor + Tap: Until end of turn, creatures your opponents
   control lose hexproof and shroud and can''t have
```

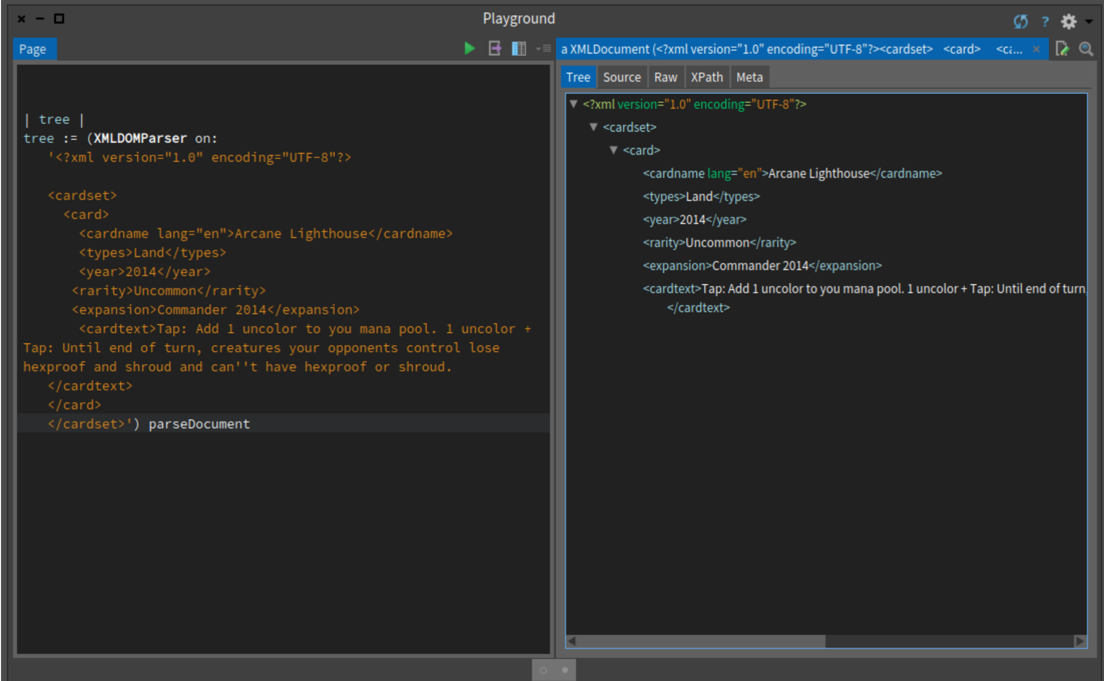


Figure 1-2 Grabbing and playing with a tree.

```
    hexproof or shroud.</cardtext>
  </card>
</cardset>') parseDocument
```

1.4 Nodes and atomic values

The following elements are nodes:

```
<cardset> (root element node)

<cardname lang="en">Arcane Lighthouse</cardname> (element node)

lang="en" (attribute node)
```

Atomic values are nodes with no children or parent. Here are some examples of atomic values:

```
Arcane Lighthouse

"en"
```

1.5 Basic tree relationships

Since we are talking about trees, nodes can have multiple relationships with each other: parent, child and siblings. Let us set some simple vocabulary.

- **Parent.** Each element and attribute has one parent. In the Arcane Lighthouse example, the card element is the parent of the cardname, types, year, rarity, expansion and cardtext.
- **Children.** Element nodes may have zero, one or more children. cardname, types, year, rarity, expansion and cardtext nodes are all children of the card element

- **Siblings.** Siblings are nodes that have the same parent. cardname, types, year, rarity, expansion and cardtext nodes are all siblings.
- **Ancestors.** A node's parent, parent's parent, etc. Ancestors of the cardname element are the card element and the cardset nodes.
- **Descendants** A node's children, children's children, etc. Descendants of the cardset element are the card,cardname, types, year, rarity, expansion and cardtext elements.

1.6 A large example

Let us expand our example to have cover more cases.

```
| tree |
tree := (XMLDOMParser on:
'<?xml version="1.0" encoding="UTF-8"?>

<cardset>
  <card>
    <cardname lang="en">Arcane Lighthouse</cardname>
    <types>Land</types>
    <year>2014</year>
    <rarity>Uncommon</rarity>
    <expansion>Commander 2014</expansion>
    <cardtext>Tap: Add 1 uncolor to you mana pool.
  1 uncolor + Tap: Until end of turn, creatures your opponents
   control lose hexproof and shroud and can''t have
   hexproof or shroud.</cardtext>
  </card>
  <card>
    <cardname lang="en">Desolate Lighthouse</cardname>
    <types>Land</types>
    <year>2013</year>
    <rarity>Rare</rarity>
    <expansion>Avacyn Restored</expansion>
    <cardtext>Tap: Add Colorless to your mana pool.
1BlueRed, Tap: Draw a card, then discard a card.</cardtext>
  </card>
</cardset>') parseDocument
```

Select the raw tab and click on self in the inspector (as shown in Figure 1-3). Now we are ready to learn XPath.

1.7 Node selection

The following table shows the XPath expressions.

Expression	Description
nodename	Selects all nodes with the name ”nodename”
/	Selects from the root node
//	Selects any node from the current node that match the selection
.	Selects the current node
..	Selects the parent of the current node
@	Selects attributes

In the following we expect that the variable `tree` is bound the full document tree we previously created parsing the xml string. In Pharo expressions selecting nodes returns set of nodes. Now let us play with the system to really see how it works.

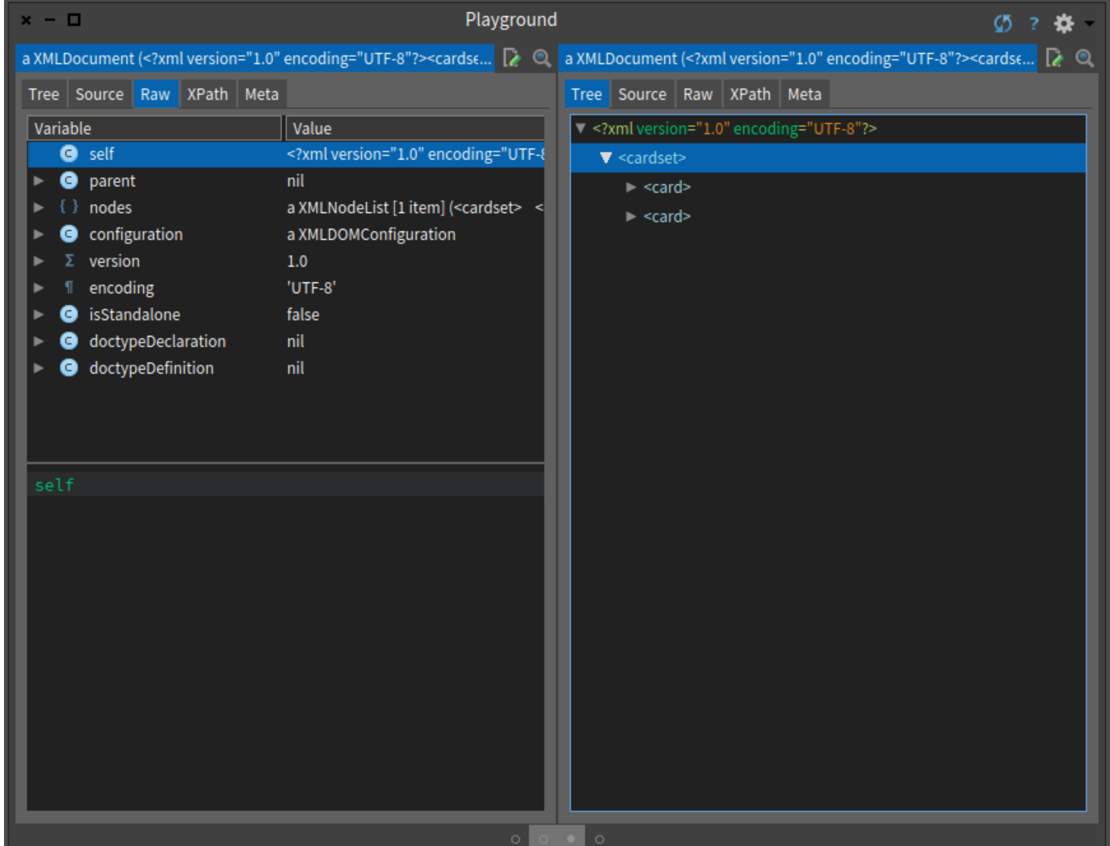


Figure 1-3 Select the raw tab and click on self in the inspector.

Node tag name

nodename	**Selects all nodes with the name ”nodename”**
card	Selects all nodes with the name ”card”

Current and parent

.	Selects the current node
..	Selects the parent of the current node

The following expression shows that `.` (period) selects the current node.

```
(tree xpath: '.') first == tree
>>> true
```

Matching path based children nodes

The operator / selects from the root node.

/	**Selects from the root node**
/cardset	Selects the root element cardset
cardset/card	Selects all the card nodes that are children of cardset

The following expression selects all the card nodes under cardset node.

```
path := XPath for: '/cardset/card'.
path in: tree.
```

It is equivalent to the following expression using the `xpath:` message

```
tree xpath: '/cardset/card'
```

Matching deep nodes

The // operation selects all the nodes matching the selection.

//	Selects any node from the current node
//year	Selects all year nodes in all the children of the current node
cardset//year	Selects all year nodes that are descendant of cardset

Let us try with another element such as the expansion of a card.

```
tree xpath: '//expansion'
>>>
a XPathNodeSet(<expansion>Commander 2014</expansion> <expansion>Avacyn
    Restored</expansion>)
```

In Pharo you can also send message to the node. So the previous expression can be expressed as follows using the message //:

```
tree // 'expansion'
>>>
a XPathNodeSet(<expansion>Commander 2014</expansion> <expansion>Avacyn
    Restored</expansion>)
```

Identifying attributes

`@` matches attributes.

Expression	Description
@	Selects attributes
//@lang	Selects all attributes that are named lang

The following expression returns all the attributes whose name is `lang`.

```
(tree xpath: '//@lang')
>>>  a XPathNodeSet(lang=""en"" lang=""en"")
```

1.8 Predicates

Predicates are used to find a specific node or a node that contains a specific value. Predicates are always embedded in square brackets.

Let us study some examples:

First element

The following expression selects the first card child of the cardset element.

```
tree xpath: '/cardset/card[1]'
>>>
a XPathNodeSet(<card>
    <cardname lang=""en"">Arcane Lighthouse</cardname>
    <types>Land</types>
    <year>2014</year>
    <rarity>Uncommon</rarity>
```

```
    <expansion>Commander 2014</expansion>
    <cardtext>Tap: Add 1 uncolor to you mana pool.
  1 uncolor + Tap: Until end of turn, creatures your opponents
   control lose hexproof and shroud and can't have
   hexproof or shroud.</cardtext>
  </card>)
```

In the XPath Pharo implementation the message `??` can be used for position or block predicates.

the previous expression is equivalent to the following one

```
tree / 'cardset' /'card' ?? 1 .
```

Block or position predicates can be applied with `??` to axis node test arguments or to result node sets.

The following expression returns the first element of each 'card' descendant:

```
tree // 'card' / ('*' ?? 1)
>>> "a XPathNodeSet(<cardname lang=""en"">Arcane Lighthouse</cardname> <cardname
    lang=""en"">Desolate Lighthouse</cardname>)"
```

Other position functions

The following expression selects the last card node that is the child of the cardset node.

```
tree xpath: '/cardset/card[last()]'.
```

The following selects the last but one node, in our case since we only have two elements we get the first.

```
tree xpath: '/cardset/card[last()-1]'.
>>>
a XPathNodeSet(<card>
      <cardname lang=""en"">Arcane Lighthouse</cardname>
      <types>Land</types>
      <year>2014</year>
      <rarity>Uncommon</rarity>
      <expansion>Commander 2014</expansion>
      <cardtext>Tap: Add 1 uncolor to you mana pool.
    1 uncolor + Tap: Until end of turn, creatures your opponents
     control lose hexproof and shroud and can't have
     hexproof or shroud.</cardtext>
    </card>)
```

We can also use the position function and use it to identify nodes. The following selects the first two card nodes that are children of the cardset node.

```
(tree xpath: '/cardset/card[position()<3]') size = 2
>>> true
```

Selecting based on node value

In addition we can select nodes based on a value of a node. The following query selects all the card nodes (of the cardset) that have a year greater than 2014.

```
tree xpath: '/cardset/card[year>2013]'.
```

The following query selects all the cardname nodes of the card children of cardset that have a year greater than 2014.

```
/cardset/card[year>2013]/cardname
>>> a XPathNodeSet(<cardname lang=""en"">Arcane Lighthouse</cardname>)
```

Selecting nodes based on attribute value

We can also select nodes based on the existence or value of an attribute. The following expression returns the cardname that have the lang attribute and whose value is 'en'.

```
tree xpath: '//cardname[@lang]
>>> a XPathNodeSet(<cardname lang=""en"">Arcane Lighthouse</cardname> <cardname
    lang=""en"">Desolate Lighthouse</cardname>)
tree xpath: '//cardname[@lang='en']
```

Note that we can simply get the card from the name using '..'.

```
tree xpath: '//cardname[@lang='en']/..
>>>
```

1.9 Selecting Unknown Nodes

In addition we can use wildcard to select any node.

Wildcard	Description
	Matches any element node
@*	Matches any attribute node
node()	Matches any node of any kind

For example //* selects all elements in a document.

```
(tree xpath: '//*') size
>>> 15
```

While //@* selects all the attributes of any node.

```
tree xpath: '//@*'
>>> a XPathNodeSet(lang=""en"" lang=""en"")
```

For example //cardname[@*] selects all cardname elements which have at least one attribute of any kind.

```
tree xpath: '//cardname[@*]'
>>> a XPathNodeSet(<cardname lang=""en"">Arcane Lighthouse</cardname> <cardname
    lang=""en"">Desolate Lighthouse</cardname>)
```

The following expression selects all child nodes of cardset.

```
tree xpath: '/cardset/*'.
```

The following expression selects all the cardname of all the child nodes of cardset.

```
tree xpath: '/cardset/*/cardname'.
```

1.10 Handling multiple queries

By using the | operator in an XPath expression you can select several paths. The following expression selects both the cardname and year of card nodes located anywhere in the document.

```
tree xpath: '//card/cardname | //card//year'
>>> a XPathNodeSet(<cardname lang=""en"">Arcane Lighthouse</cardname> <year>2014</year>
<cardname lang=""en"">Desolate Lighthouse</cardname> <year>2013</year>)"
```

1.11 XPath axes

Xpath introduces another way to select nodes using *location step* following the syntax: `axisname::node-test[predicate]`. Such expressions can be used in the steps of location paths (see below).

An axis defines a node-set relative to the current node. Here is a table of the available axes.

AxisName	Result
ancestor	Selects all current node ancestors
ancestor-or-self	... and the current node itself
attribute	Selects all current node attributes
child	Selects all current node children
descendant	Selects all current node descendants
descendant-or-self	... and the current node itself
following	Selects everything after the current node closing tag
following-sibling	Selects all siblings after the current node
namespace	Selects all current node namespace nodes
parent	Selects current node parent
preceding	Selects all nodes that appear before the current node except ancestors, attribute nodes and namespace nodes
preceding-sibling	Selects all siblings before the current node
self	Selects the current node

Paths

A location path can be absolute or relative. An absolute location path starts with a slash (/) (/step/step/...) and a relative location path does not (step/step/...). In both cases the location path consists of one or more location steps, each separated by a slash.

Each step is evaluated against the nodes in the current node-set. A location step, `axisname::node-test[predicate]`, consists of:

- an axis (defines the tree-relationship between the selected nodes and the current node)
- a node-test (identifies a node within an axis)
- zero or more predicates (to further refine the selected node-set)

The following example access the year node of all the children of the cardset.

```
tree xpath: '/cardset/child::node()/year').
>>>a XPathNodeSet(<year>2014</year> <year>2013</year>)
```

The following expression gets the ancestor of the year node and selects the cardname.

```
(tree xpath: '/cardset/card/year') first  xpath: 'ancestor::card/cardname'
>>> "a XPathNodeSet(<cardname lang=""en"">Arcane Lighthouse</cardname>)"
```

1.12 Conclusion

XPath is a powerful language and the Pharo XPath library developed and maintained by Monty Kamath is implementing the full standard 1.0. In addition coupled with life programming capabilities of Pharo it gives a really powerful to explore structured data.

CHAPTER 2

Scraping HTML

Internet pages provide a lot of information and often you would like to be able to access and manipulate it in another form than HTML: HTML is just plain verbose. What you would like is to get access to only the information you are interested in and get the results in a form that you can easily build more software. This is the objective of HTML scraping. In Pharo you can scrape web pages using different libraries such as XMLParser and SOUP. In this chapter we will show you how we can do that using XMLParser to locate and collect the data we need and JSON to format and output the information.

This chapter has been originally written by Peter Kenny and we thank him for sharing with the community this little tutorial.

2.1 Getting started

You can use the Catalog browser to load XMLParserHTML and NeoJSON just execute the following expressions:

```
Gofer it
   smalltalkhubUser: 'PharoExtras' project: 'XMLParserHTML';
   configurationOf: 'XMLParserHTML';
   loadStable.
```

```
Gofer it
   smalltalkhubUser: 'PharoExtras' project: 'XPath';
   configurationOf: 'XPath';
   loadStable.
```

```
Gofer it
   smalltalkhubUser: 'SvenVanCaekenberghe' project: 'Neo';
   configurationOf: 'NeoJSON';
   loadStable.
```

2.2 Define the Problem

This tutorial is based on a real life problem. We need to consult a database published by the US Department of Agriculture, extract data for over 8000 food ingredients and their nutrient contents and output the results as a JSON file. The main list of ingredients can be found at the following url: https://ndb.nal.usda.gov/ndb/search/list?sort=ndb&ds=Standard+Reference (as shown in Figure 2-1). You can also find the HTML version of the file in the github repository of this book https://github.com/SquareBracketAssociates/Booklet-Scraping/resources.

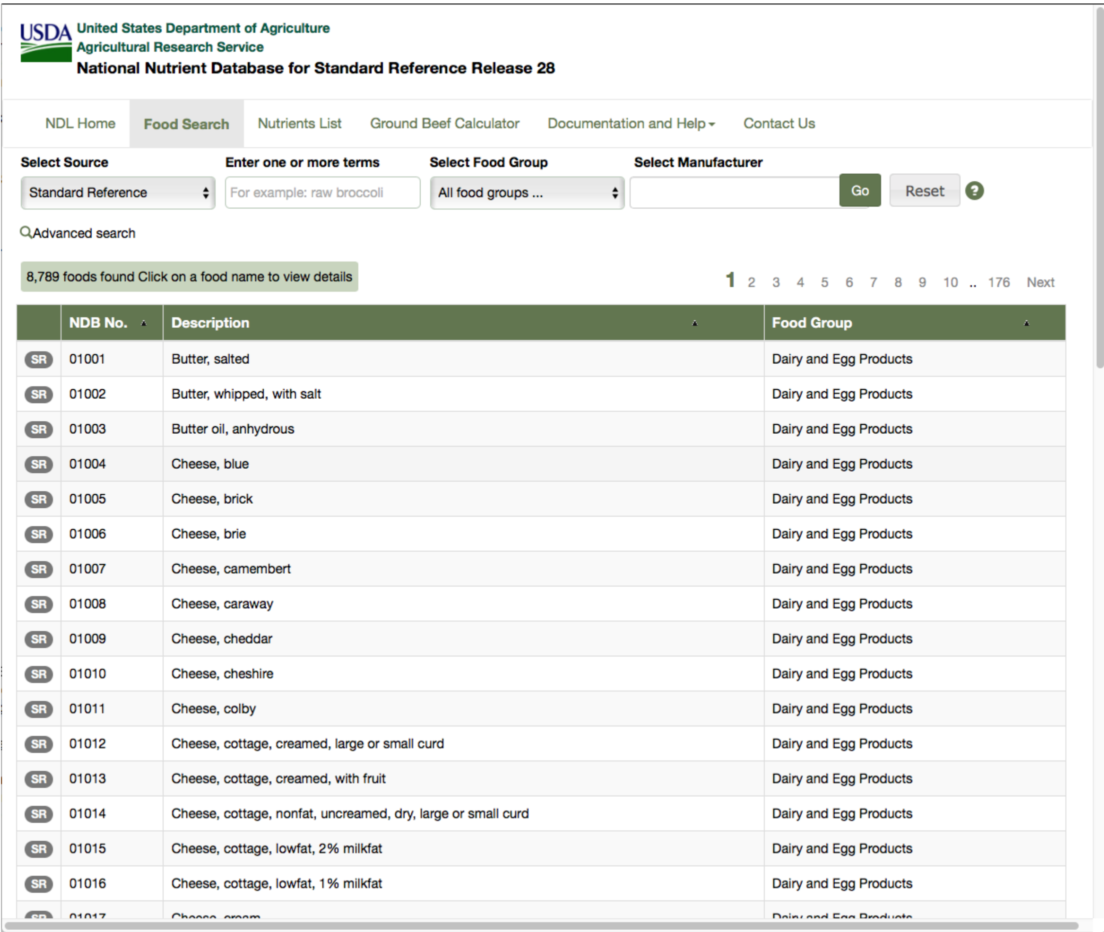


Figure 2-1 Food list.

This table shows the first 50 rows, each corresponding to an ingredient. The table shows the NDB number, description and food group for each ingredient. Clicking on the number or description leads to a detailed table for the ingredient. This table comes in two forms, basic details and full details, and the information we want is in the full details. The full detailed table for the first ingredient can be found at the url: https://ndb.nal.usda.gov/ndb/foods/show/1?format=Full (as shown in Figure 2-2).

There are two areas of information that need to be extracted from this detailed table:

- There is a row of special factors, in this case beginning with 'Carbohydrate Factor: 3.87'. This is to be extracted as a set of (name, value) pairs. The number of factors can vary; some ingredients do not have any.
- There is a table of data for various nutrients, which are arranged in groups - proximates, vitamins, lipids etc. The number of columns in the table varies from one ingredient to another, but in every case the first three columns are nutrient name, unit of measurement and quantity; we have to extract these columns for every listed nutrient.

The requirement is to extract all this information for each ingredient, and then output it as a JSON file:

- NBD number, description and food group from the main list;
- Factor names and values from the detailed table;
- Nutrient details from the detailed table.

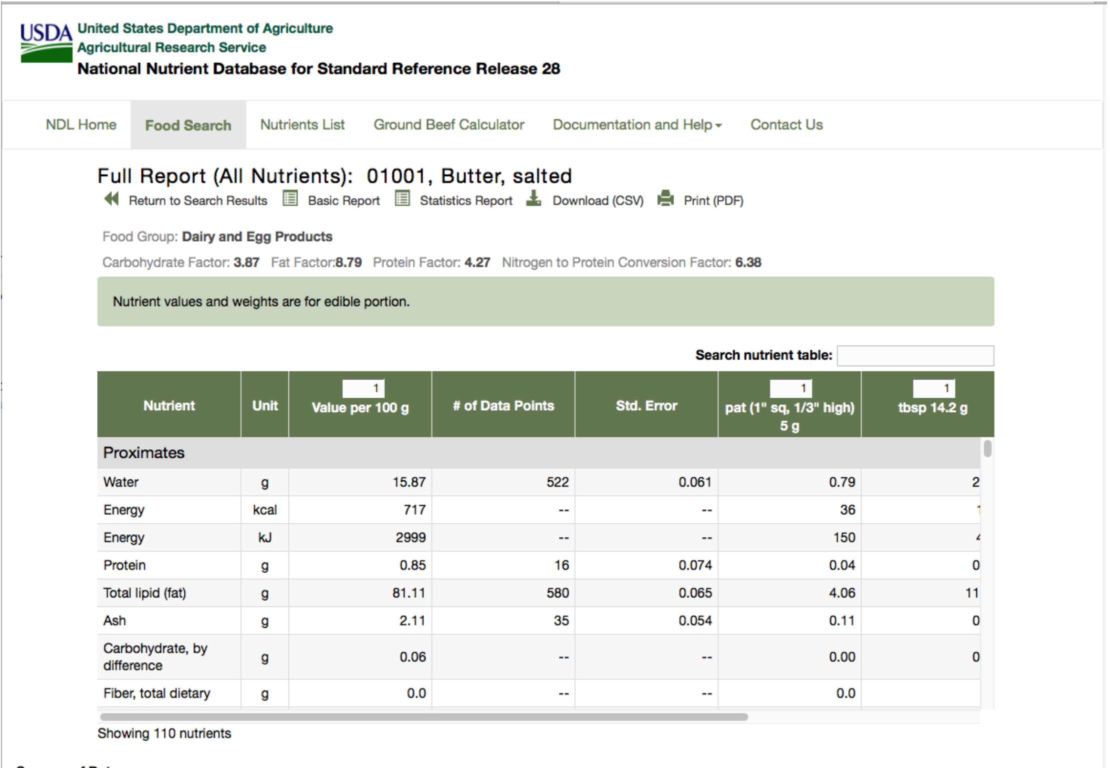


Figure 2-2 Food details - Salted Butter.

2.3 First find the required data

To start, we have to find where the required data are to be found in the HTML file. The general rule about this is that there are no rules. Web site designers are concerned only with the visual effect of the page, and they can use any of the many tricks of HTML to produce the desired effects. We use the XML HTML parser to convert text into an XML tree (a tree whose nodes are XML objects). We then explore this tree to find the elements we want, and for each one we have to find signposts showing a route through the tree to uniquely locate the element, using a combination of XPath and Smalltalk programming as required. We may use the names or attributes of the HTML tags, each of which becomes an instance of XMLElement in the tree, or we may match against the text content of a node.

First read in the table of ingredients (first 50 rows only) as in the url.

```
| ingredientsXML |
ingredientsXML := XMLHTMLParser parseURL:
    'https://ndb.nal.usda.gov/ndb/search/list?sort=ndb&ds=Standard+Reference'.
ingredientsXML inspect
```

You can execute the expression and inspect its result. You will obtain an inspector on the tree and you can navigate this tree as shown in Figure 2-3.

Since you may want to work on files that you saved on your disc you can also parse a file and get an XML tree as follows:

```
| ingredientsXML |
ingredientsXML := (XMLHTMLParser onFileNamed: 'FoodsList.html') parseDocument.
```

The simplest way to explore the tree is starting from the top, i.e. by opening up the `<body>` node, but this can be tedious and confusing; we often find that there are many levels of nested `<div>` nodes before finding what we want. Alternatively, we can use XPath speculatively to look for in-

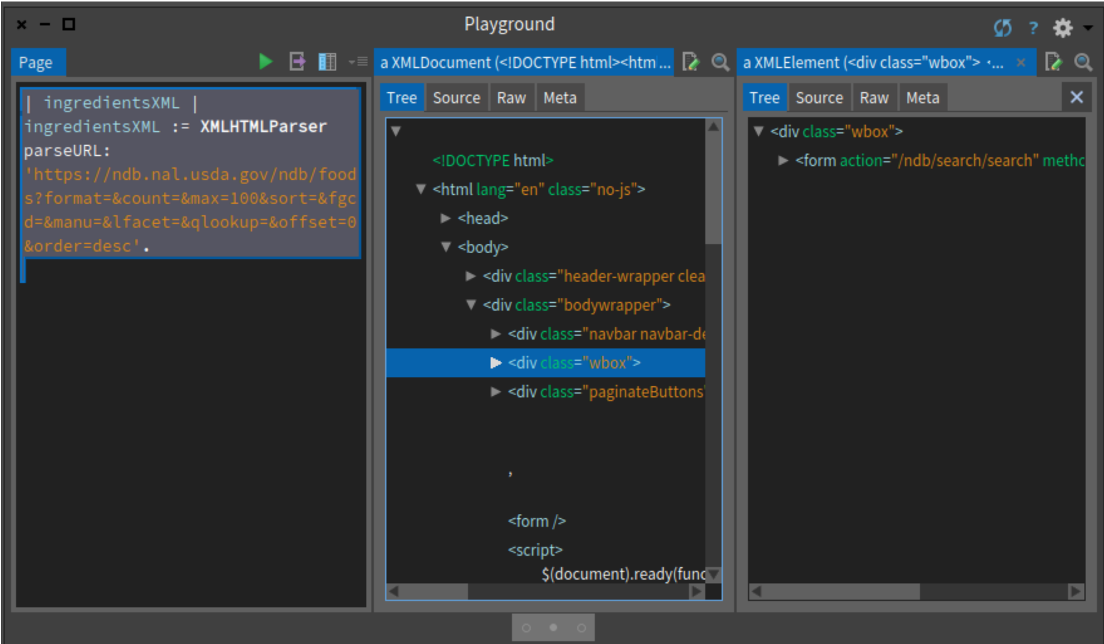


Figure 2-3 Navigating the XML document inside the inspector.

teresting nodes. In the case of the foods list, we might guess that the list of ingredients will be stored in a `<table>` node. Having parsed the web page as shown above in a playground, we can then enter:

```
ingredientsXML xPath: '//table'
```

and select 'do it and go', which shows an `XMLNodeList` of all the table nodes - only one in this case. If there were several, we could use the attributes of the node or any of its ancestors to locate the one we want. We find by searching up several generations a node `<div class="wbox">` which is unique, so we could use this as a signpost. The table body contains a row for each ingredient; the first cell in the row is a "badge" which is of no interest, but the remaining three cells in the row are the number, description and group name that we want. The second and third cells both contain an emebedded node `<a href="....">` showing the relative url of the associated detail table.

The exploration of the detail table proceeds in a similar way; we search for text nodes which contain the word "Factor", and then for a table containing the nutrient details. More of this below.

2.4 Going back to our problem

Here we present the essential points of the data scraping and JSON output for one item, in a logical order. The code is presented as it could be entered in a playground. There are brief comments on the format of the data and the signposts used to locate it. First read in the table of ingredients (first 50 rows only) as before.

```
ingredientsXML := XMLHTMLParser parseURL:
    'https://ndb.nal.usda.gov/ndb/search/list?sort=ndb&ds=Standard+Reference'.
```

The detail rows are in the body of the table in the div node whose class is 'wbox'.

```
ingredientRows := (ingredientsXML xPath: '//div[@class=''wbox'']//tbody/tr').
```

Note that the signposts do not need to show every step of the way, provided the route is unique; we do not need to mention the `<table>` node, because there is only one `<tbody>`. Now extract the

text content of the four cells in each row; 'strings first' is a convenient way of finding the text in a node while ignoring any descendent nodes, and we routinely trim redundant spaces.

```
ingredientCells := ingredientRows collect:
              [:row| (row xPath: 'td') collect:
                              [ :cell| cell strings first trim]].
```

To prepare for export to JSON, it is handy to put the three required fields (ignoring the first) in a Dictionary indexed by their field names. Using an OrderedDictionary is not strictly necessary, but it does mean that the JSON output is easier for a human to understand.

```
ingredientsJSON := ingredientCells collect:
              [ :row| { 'nbd_no' -> (row at: 2).
                              'full-name' -> (row at: 3).
                              'food-group' -> (row at: 4)}
asOrderedDictionary ].
```

If we 'do it and go' the next line, we can see the JSON layout. For this demo, we do not need to export to a JSON file; it is easier to look at it as text in the playground.

```
NeoJSONWriter toStringPretty: ingredientsJSON first.
```

We can find the relative url address of the ingredient details from the href in the second cell. Because this is the address of the basic details table, we edit it to discard all the parameters, so that we can edit in the parameters for the full table.

```
ingredientAddress := ingredientRows collect:
              [ :row| (row xPath:'td[2]/a/@href') first value copyUpTo: $?].
```

Up to this point, we have been constructing lists with data for all 50 ingredients in the table. To show how to process the ingredient details, we just process the first ingredient in the file. The production version would have to run through all the rows in the ingredientAddress collection. We read and parse the detail file, after editing the url.

```
ingredientDetailsXML := XMLHTMLParser parseURL: 'https://ndb.nal.usda.gov',
    ingredientAddress first, '?format=Full'.
```

The data for the factors are contained in `<span>` nodes within `<div class="row">` nodes. This does not identify them uniquely, so we extract all such nodes with XPath and then use ordinary Smalltalk to find the ones mentioning the word 'Factor'.

```
factorCells := (ingredientDetailsXML xPath: '//div[@class=''row'']//span')
      collect: [:each| each strings first trim].

factors := OrderedCollection new.
1 to: factorCells size by: 2 do: [ :index|
  ((factorCells at: index) matches: 'Factor') ifTrue: [factors addLast:
    {'factor' -> (factorCells at: index).
    'amt' -> ((factorCells at: index + 1) trimRight:[:c|c asInteger = 160])}
    asOrderedDictionary]].
```

Note: it appears that the web designers have used no-break space characters to control the formatting, and these are not removed by 'trim', so we use the 'trimRight:' clause above to remove them.

The layout of the nutrients table is messy, presumably to achieve the effect of the inserted row with the nutrient group name. This means that we cannot locate the nutrient rows using `<tr>` nodes, as we did for the main list. Instead we have to get at all the individual table cells in `<td>` nodes, and then count them in groups equal to the row length. Since the row length is not a constant, we have to determine it by examining the data for one row that in a `<tr>` node.

```
nutrientCells := (ingredientDetailsXML xPath: '//table//td') collect: [:each|each
    strings first trim].

nutRowLength := (ingredientDetailsXML xPath: '//table/tbody/tr') first elements size.
```

```
{       "nbd_no" : "01001",
        "full-name" : "Butter, salted",
        "food-group" : "Dairy and Egg Products",
        "factors" : [
                {       "factor" : "Carbohydrate Factor:",
                        "amt" : "3.87"  },
                {       "factor" : "Fat Factor:",
                        "amt" : "8.79"  },
                {       "factor" : "Protein Factor:",
                        "amt" : "4.27"  },
                {       "factor" : "Nitrogen to Protein Conversion Factor:",
                        "amt" : "6.38"  }
        ],
        "nutrients" : [
                {       "group" : "Proximates",
                        "nutrient" : "Water",
                        "unit" : "g",
                        "per100g" : "15.87"     },
                {       "group" : "Proximates",
                        "nutrient" : "Energy",
                        "unit" : "kcal",
                        "per100g" : "717"       },
        <106 nutrients omitted>
                {       "group" : "Other",
                        "nutrient" : "Caffeine",
                        "unit" : "mg",
                        "per100g" : "0" },
                {       "group" : "Other",
                        "nutrient" : "Theobromine",
                        "unit" : "mg",
                        "per100g" : "0" }
        ]
}
```


Figure 2-4 Sample of JSON output.

```
nutrients := OrderedCollection new.
1 to: nutrientCells size by: nutRowLength do:
[:index|nutrients addLast:
  { 'group' -> (nutrientCells at: index).
  'nutrient' -> (nutrientCells at: index + 1).
  'unit' -> (nutrientCells at: index + 2).
  'per100g' -> (nutrientCells at: index + 3) }
  asOrderedDictionary ].
```

Finally assemble all the information for the first ingredient as a JSON file. NeoJSON automatically takes care of embedding dictionaries within a collection within a dictionary. (See specimen in Figure 2-4)

```
NeoJSONWriter toStringPretty:
  ((ingredientsJSON first)
    at: 'factors' put: factors asArray;
    at: 'nutrients' put: nutrients asArray;
    yourself).
```

2.5 Turning the pages

The code above will extract the data for one ingredient, and could obviously be repeated for all the 50 items in one page of data. However, the entire database contains 8789 ingredients at the time of

writing, which amounts to 176 pages. The database seems to impose a limit of 50 ingredients per page, so to process the entire database we need to read the pages in succession. Each page contains a link which, if clicked, will load the next page. We can do this programmatically, by finding the link after processing the page. The link is contained in node `<div class="paginateButtons">`, so we can use the code:

```
nextButtons := (ingredientsXML xPath: '//div[@class=''paginateButtons'']//a')
       select:[:node| node strings first = 'Next'].

nextURL := (nextButtons size > 0)
      ifTrue:['https://ndb.nal.usda.gov', (nextButtons first attributeAt: 'href')]
      ifFalse: [nil].
```

This is a common requirement in processing large databases on the web, and so we can use a standard pattern:

```
<code to initialise results>
nextURL := <url for first page of database>
[nextURL isNil] whileFalse:
[pageXML := XMLHTMLParser parseURL: nextURL.
<code to extract data from pageXML to results>
<code to determine nextURL from pageXML; should yield 'nil' for last page>
]
```

2.6 Conclusion

We have presented a way to extract information from a structured document. The methods used are of course particular to the layout of the USDA database, but the general principles should be clear. A mixture of XPath and Smalltalk can be used in order to locate the required data.

One problem which can arise, if we need to repeat the extraction with updated data, is that the web designers can change the layout of the pages; this did in fact happen with the USDA table in the 15 months between originally tackling the problem and writing this article. The usual result is that the signposts no longer work, and the XPath results are empty. If the update is being run automatically, say on a daily basis, it may be worth while inserting defensive code in the processing, which will raise an exception if the results are not as expected. How to do this will depend on the particular application.

CHAPTER 3

Scraping Magic

In this chapter we will scrap the web site of Magic the gathering and in particular the card database. (Yes I play Magic not super good but well I have fun). Here is one example http://gatherer.wizards.com/Pages/Card/Details.aspx?multiverseid=389430 as shown in Figure 3-1. Now we will try to show you how we explore the HTML page using the excellent Pharo inspector: diving in the tree nodes and checking live their attributes or children is simply super cool.

3.1 Getting a tree

The first thing was to make sure that we can get a tree from the web page. For this task we used the `XMLHTMLParser` class and sends it the message `parseURL:`. How did we find this message... Simply looking on the class side methods of the class. How did we find the class, well looking at the subclass of `XMLDOMParser` because HTML is close to XML or the inverse :).

```
| tree |
tree := (XMLHTMLParser parseURL:
    'http://gatherer.wizards.com/Pages/Card/Details.aspx?multiverseid=389430')
```

3.2 First the card visual

First we would like to grab the card visual because this is fun and cool. When we open the card visual in a separate window we see that the url is http://gatherer.wizards.com/Handlers/Image.ashx?multiverseid=389430&type=card. Therefore we started to look for Handlers in the nodes as shown in Figure 3-2.

```
| tree |
tree := (XMLHTMLParser parseURL:
    'http://gatherer.wizards.com/Pages/Card/Details.aspx?multiverseid=389430').
tree xpath: '//img'
```

No so cool but working...

Toying with the inspector, we come up with the following ugly expression to get the name of the JPEG

```
| tree |
tree := (XMLHTMLParser parseURL:
    'http://gatherer.wizards.com/Pages/Card/Details.aspx?multiverseid=389430').
((tree xpath: '//img') third @ 'src') first value allButFirst: 5
>>> 'Handlers/Image.ashx?multiverseid=389430&type=card'
```

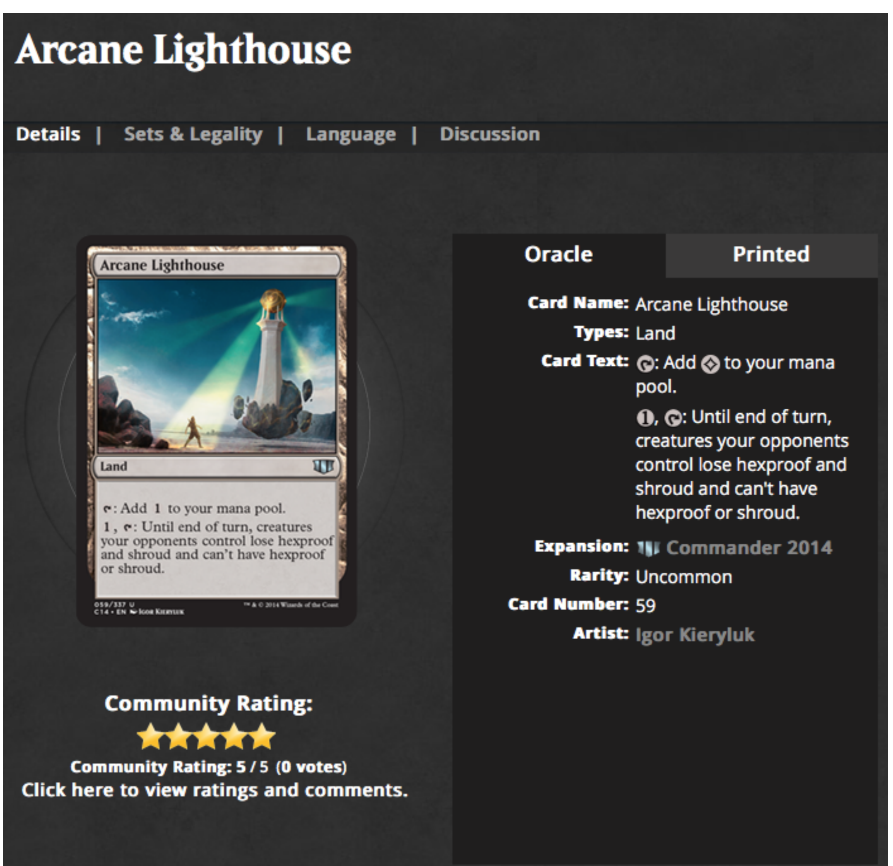


Figure 3-1 http://gatherer.wizards.com/Pages/Card/Details.aspx?multiverseid=389430.

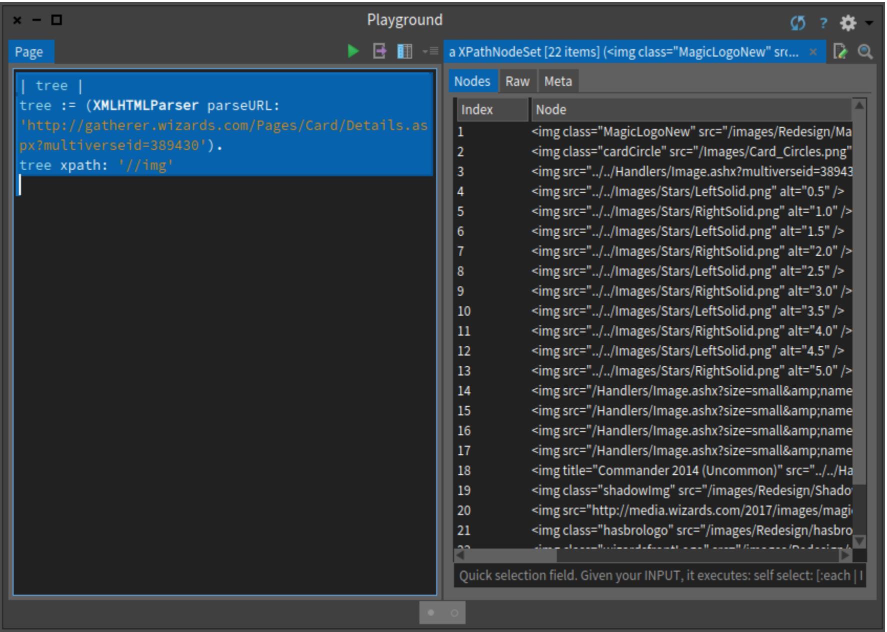


Figure 3-2 Exploring images.

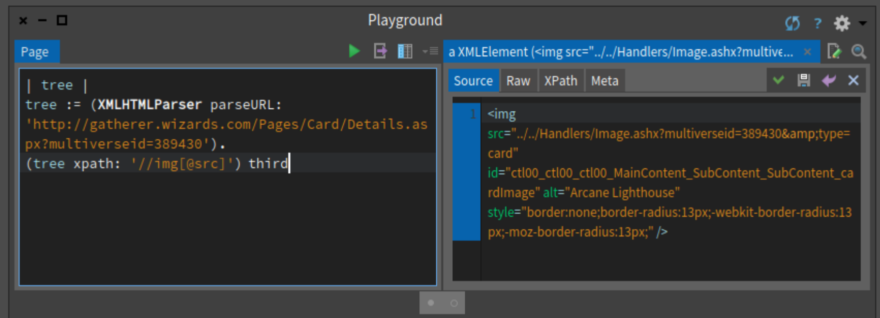


Figure 3-3 Exploring images.

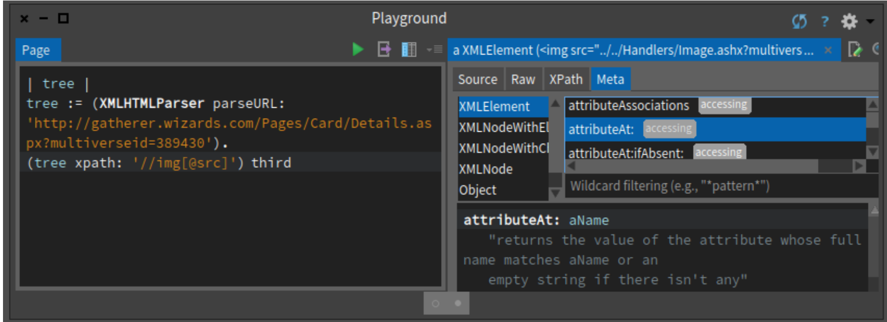


Figure 3-4 Narrowing the node.

Ugly isn't it? This happens often when scraping HTML, but we can do better. By the way note also that we start to enter directly XPath command using the XPath pane and using the doit and go facilities of the inspector. This way we do not have to get the page from internet all the time.

3.3 Revisiting it

We could not really show you such ugly expressions so we had to find a better one.

So first we look at the img that has src as atttribute as shown below and in Figure 3-3.

```
| tree |
tree := (XMLHTMLParser parseURL:
    'http://gatherer.wizards.com/Pages/Card/Details.aspx?multiverseid=389430').
(tree xpath: '//img[@src]')
```

Then as shown in Figure 3-4 we inspected the right node.

Finally since we were on this exact node, we looked in its class to see if we could get an API to get the attribute in a nice way as shown in Figure 3-5.

```
| tree |
tree := (XMLHTMLParser parseURL:
    'http://gatherer.wizards.com/Pages/Card/Details.aspx?multiverseid=389430').
(tree xpath: '//img[@src]') third attributeAt: 'src'
```

Now that we have the visual path, we can use the HTTP client of Pharo to get the image as shown in Figure 3-6.

```
| tree path |
tree := (XMLHTMLParser parseURL:
    'http://gatherer.wizards.com/Pages/Card/Details.aspx?multiverseid=389430').
```

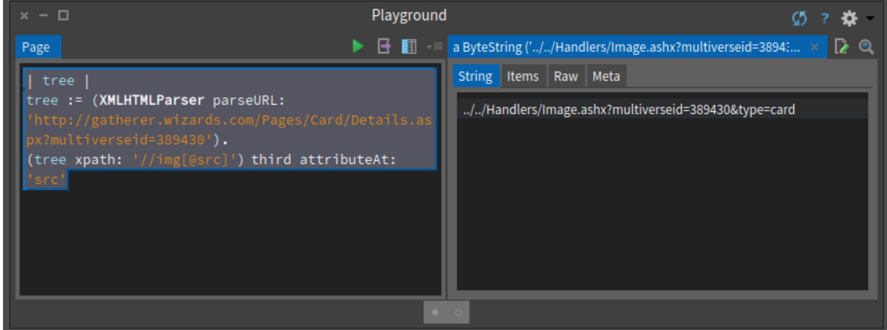


Figure 3-5 Exploring the class API on the spot: looking to see if there is a attribute something method.

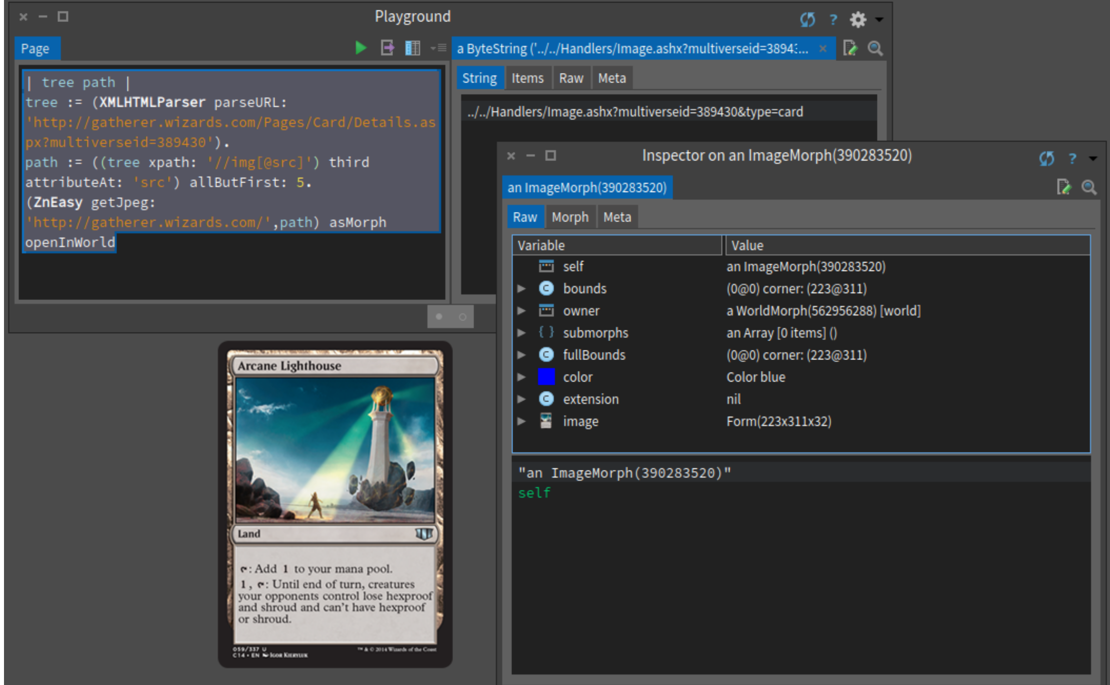


Figure 3-6 Getting the card visual inside Pharo.

```
path := ((tree xpath: '//img[@src]')  third attributeAt: 'src') allButFirst: 5.
(ZnEasy getJpeg: 'http://gatherer.wizards.com/',path) asMorph openInWorld
```

3.4 Getting data

Since this web page is probably generated, we look for example for the artist string in the source and we found the following matches:

```
ClientIDs.artistRow = 'ctl00_ctl00_ctl00_MainContent_SubContent_SubContent_artistRow';
```

This one is more interesting:

```
<div id="ctl00_ctl00_ctl00_MainContent_SubContent_SubContent_artistRow" class="row">
   <div class="label">
       Artist:</div>
   <div id="ctl00_ctl00_ctl00_MainContent_SubContent_SubContent_ArtistCredit"
     class="value">
```

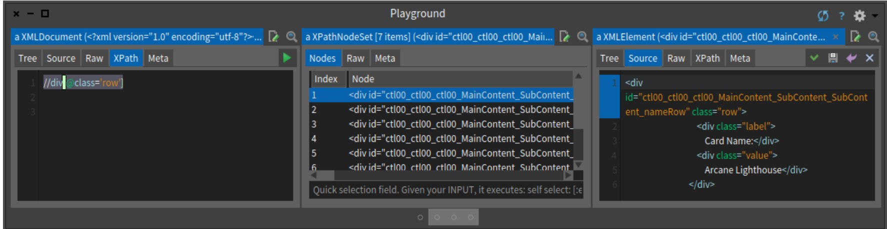


Figure 3-7 Getting the card information.

```
    <a href="/Pages/Search/Default.aspx?action=advanced&amp;artist=[%22Igor
    Kieryluk%22]">Igor Kieryluk</a></div>
```

We can build queries to identify node elements having this id. To avoid to perform an internet request each time, we typed directly XPath path in the XPath pane of the inspector as shown in Figure 3-7. Now trying to get faster we looked at all the class="row" as shown in Figure 3-7

```
//div[@class='row']
```

The following expression returns the pair label and value for example for the card name label and its value.

```
//div[@class='row']/div[@class='label']| //div[@class='row']/div[@class='value']
```

So we can now query all the fields

```
| tree |
tree := (XMLHTMLParser parseURL:
    'http://gatherer.wizards.com/Pages/Card/Details.aspx?multiverseid=389430').
container := tree xpath: '//div[@class=''row'']/div[@class=''label'']|
    //div[@class=''row'']/div[@class=''value'']'.
container collect: [ :each | each contentString trimBoth  ].
>>> a XMLOrderedList('Card Name:' 'Arcane Lighthouse' 'Types:' 'Land' 'Card Text:'
': Add  to your mana pool. , : Until end of turn, creatures your opponents control
lose hexproof and shroud and can''t have hexproof or shroud.'
'Expansion:' 'Commander 2014' 'Rarity:' 'Uncommon' 'Card Number:' '59' 'Artist:' 'Igor
    Kieryluk')
```

Now we can convert this into a dictionary

```
| tree  |
tree := (XMLHTMLParser parseURL:
    'http://gatherer.wizards.com/Pages/Card/Details.aspx?multiverseid=389430').
container := tree xpath: '//div[@class=''row'']/div[@class=''label'']|
    //div[@class=''row'']/div[@class=''value'']'.
((container collect: [ :each | each contentString trimBoth  ])
  asOrderedCollection groupsOf: 2 atATimeCollect: [ :x :y | x -> y]) asDictionary
```

And convert it into JSON for fun

```
| tree dict |
tree := (XMLHTMLParser parseURL:
    'http://gatherer.wizards.com/Pages/Card/Details.aspx?multiverseid=389430').
container := tree xpath: '//div[@class=''row'']/div[@class=''label'']|
    //div[@class=''row'']/div[@class=''value'']'.
dict := ((container collect: [ :each | each contentString trimBoth  ])
  asOrderedCollection groupsOf: 2 atATimeCollect: [ :x :y | x -> y]) asDictionary.

NeoJSONWriter toStringPretty:dict
>>>

'{
```

```
  "Card Number:" : "59",
  "Card Name:" : "Arcane Lighthouse",
  "Artist:" : "Igor Kieryluk",
  "Types:" : "Land",
  "Card Text:" : ": Add  to your mana pool. , : Until end of turn, creatures your
    opponents control lose
   hexproof and shroud and can''t have hexproof or shroud.",
  "Expansion:" : "Commander 2014",
  "Rarity:" : "Uncommon"
}'
```

Now we can apply the same technique to access all the cards and also different pages to extract all the card unique id and query the database. But this is left as an exercise.

3.5 Conclusion

We show you how we could access the page and navigate interactively through it using XPath and live programming feature of Pharo. This chapter should show the great value to be able to tweak you live a document and navigate to find the information you really want.