

Chapter 7

Water Transform: Transforming XML Objects – XSLT

IN THIS CHAPTER

- ◆ Understanding Simple Data Transforms
- ◆ Sorting
- ◆ Combining and Splitting Fields
- ◆ Flattening and Adding Structure
- ◆ Modifying Original

ALMOST EVERY BUSINESS APPLICATION gets some amount of data from external systems. Water Import (Chapter 8) can be used for converting structured non-XML data to XML Water objects, but the data will likely require some additional amount of processing to make it compatible with the object definitions required for internal applications.

XSLT is a special-purpose document transformation language, while Water Transform is designed for general-purpose data transformation. Water Transform can use any Water method for helping with the transformation.

Simple Transformations

To demonstrate transformations, you first need a sample data object to transform. The first example shows an instance of `thing` with two fields named `full_name` and `title`. The instance is assigned to the variable `x`.

```
<set a_customer=  
  <thing  
    full_name="Mike Plusch"  
    title=" CTO "  
  />  
</set>
```

Trimming Whitespace

The following example shows how to remove any whitespace from the beginning or end of a string using the `trim` method:

```
a_customer.<set title=a_customer.title.<trim/> />
```

Output in ConciseXML:

```
<thing
  full_name="Mike Plusch"
  title="CTO"
/>
```

The first line is a call to `set`. The `set` call takes arguments of the form `key=value`. The `title` field of `a_customer` will be set to the original `title` that is trimmed.

Transforming Multiple Fields

The next example shows two conversions in one call to `set`. First a new instance is created and assigned to `X`.

```
<set X=
  <thing
    full_name="Mike Plusch"
    title="CTO - Chief Technology Officer"
  />
/>
```

In the following example, `to_uppercase` converts the first name to uppercase, and `<subvector end=10/>` takes only the first ten characters of the `title`:

```
X.<set
  first_name=X.first_name.<to_uppercase/>
  title=X.title.<subvector end=10/>
/>
```

Output in ConciseXML:

```
<thing
  full_name="MIKE PLUSCH"
  title="CTO - Chie"
/>
```

Replacing Values

Another standard transformation is removing or replacing a substring from a larger string.

```
***data_value***.<replace "***" ""/><replace "_" " " />
```

Output in ConciseXML:

```
"data value"
```

The preceding example is a Water Path with three path parts. The first path part is the string `***data_value***`. The second path part is a call to `replace`. This call replaces all the occurrences of `**` with the empty string `"`, effectively removing all occurrences of `**`. `replace` returns the new string, and the third path part `<replace "_" " " />` takes the new string `"data_value"` and replaces all occurrences of an underscore (`_`) with a space that returns the new string `"data value"`.

`replace` is useful for converting or removing individual characters from a string or for converting entire words. All the occurrences of `"Water"`, for example, could be replaced by `"Steam"`.

Notice how the methods can be called in a sequence. In Unix, the sequence of method calls would be called a process pipeline. `Water` refers to this sequence as a path. The output of one method is fed into the input of the next method.

```
a_string.<replace "Water" "Steam"/>
```

Renaming Fields

Transforming one object into another object often involves the changing of field names. The first expression in the example below shows the original instance. The second expression is the desired result. Notice how all the fields' values remain untouched, but underscores (`_`) are removed from every field name.

```
<thing
  first_name="Mike"
  middle_name="D"
  last_name="Plusch"
  title="CTO"
/>
```

Desired Output:

```
<thing
  firstname="Mike"
  middlename="D"
  lastname="Plusch"
  title="CTO"
/>
```

The next group of examples shows several ways to rename fields. The first expression sets a new instance to `x`. The second expression calls the `rename` method, which takes pairs of arguments of the form `new_key="orig_key"`. `rename` returns the object whose fields were renamed.

```

<set x=
  <thing
    first_name="Mike"
    middle_name="D"
    last_name="Plusch"
    title="CTO"
  />
/>
x.<rename fname="first_name"/>

```

Output in ConciseXML:

```

<thing
  fname="Mike"
  middle_name="D"
  last_name="Plusch"
  title="CTO"
/>

```

`rename` is similar to `set` in that it takes multiple pairs of arguments in the form of `new_key="original_key"` as shown below.

```

x.<rename middlename="middle_name"
  last_name="lastname"
/>

```

`rename_field` is similar to `rename`, but it allows you to specify the new key and old key with strings or any Water expression. The following two lines, for example, have the same effect:

```

x.<rename f_name="first_name"/>
x.<rename_field new_key="f_name" old_key="first_name"/>

```

`rename` uses a single argument where the argument key is the new key and the argument value is the old key. `rename_field`, by contrast, takes two arguments, `new_key` and `old_key`. `rename_field` has some flexibility over `rename` that is shown below.

Although `rename_field` can only change one field at a time, multiple calls to `rename_field` can be made within a single Water Path.

```

x.<rename_field new_key="middlename" old_key="middle_name"/>.
  <rename_field new_key="lastname" old_key="last_name"/>

```

Removing Fields

Fields can be removed from an object with a call to `remove`. Here's an example of removing one field:

```

X.<remove "first_name"/>

```

Looping for Multiple Transformations

This particular transform follows a specific pattern – remove every underscore from the names of fields. If the same transformation applies to every key, then automate the process by using `for_each` loop. The following is the original object:

```
<set x=
  <thing
    first_name="Mike"
    middle_name="D"
    last_name="Plusch"
    title="CTO"
  />
/>
```

The next example removes the underscore from all field keys using a call to `replace`.

```
x.<for_each include=user_field>
  x.<rename_field old_key=key new_key=key.<replace "_" "-"/> />
</for_each>
x
```

Output in ConciseXML:

```
<thing
  first-name="Mike"
  middle-name="D"
  last-name="Plusch"
  title="CTO"
/>
```

The first line says that some code should be executed for every user-defined field in `x`. The second line calls `rename_field` on `x`. `key` is a local variable that is always available in the content of `for_each`. It contains the key for the current field. The `new_key` is the value of the original key without the underscore. Note that the `title` field had no underscores, and therefore remained the same.

`remove` returns the value that was removed. This is different than either `set` or `rename`, which returns the modified object.

`remove` can remove any number of fields in the same call. For example:

```
X.<remove "fname" "mname" "lname"/>
```

If a call to `remove` lists two or more keys, `remove` returns a vector of all the values from the fields that were removed.

Advanced Transforms

Advanced transforms make changes across multiple fields, such as splitting one field into two fields and combining multiple fields into a single field. The sorting and reversing of fields in an object are also covered.

Combining Multiple Fields into One Field

A common transformation is combining data from multiple fields into one field. The following example shows the original object and the desired output:

```
<thing
  fname="Mike"
  mname="D"
  lname="Plusch"
/>
```

Desired Output:

```
<thing
  full_name="Mike D Plusch"
/>
```

First, set the starting object to `x`.

```
<set x=
  <thing
    fname="Mike"
    mname="D"
    lname="Plusch"
  />
/>
```

`full_name` is calculated by concatenating three fields with a space between each field.

```
x.<set full_name=X.fname.<concat " " X.mname " " X.lname/> />
```

Output in ConciseXML:

```
<thing
  fname="Mike"
  mname="D"
  lname="Plusch"
```

```

    full_name="Mike D Plusch"
  />

```

The second step is to remove the fields that are no longer needed.

```

x.<remove "fname" "mname" "lname"/>
x

```

Output in ConciseXML:

```

<thing
  full_name="Mike D Plusch"
/>

```

Another option for joining the fields is calling `join`. `join` takes a vector of values and concatenates the values together with a separator.

```

X.<set full_name=<vector X.fname X.mname X.lname/>.<join " "/> />

```

Using 'if' in a Transform

If the middle name is the empty string "", then two spaces would exist between the first and last names. The following example uses `if` to add a space only if there is a middle name:

```

X.<set full_name=
  "<concat X.fname " "
    <if> X.mname.<is_not ""/>
      X.mname.<concat " "/>
    </if>
  X.lname
  />
/>

```

Another option would be to use `for_each` to return only non-empty values and then `join` those values with a space.

```

X.<set full_name=
  X.<for_each include=<vector "fname" "mname" "lname"/>
    returns='all_except_null'
  >
    <if> value.<is_not ""/> value </if>
  </for_each>.<join " "/>
/>

```

Splitting a Field

The opposite of combining multiple fields into a single field is splitting a single field into multiple fields. The following example shows the original object and the desired object after the transform:

```
<thing
  full_name="Mike D Plusch"
/>
```

Desired Output:

```
<thing
  fname="Mike"
  mname="D"
  lname="Plusch"
/>
```

To accomplish this transform, I created a method named `split_field`. `split_field` takes the original key as the first argument and a vector of strings in the `into_keys` argument that represent the new fields to be created.

```
<defmethod split_field orig_key into_keys separator=" ">
  <set values=this.<get orig_key/>.<split separator/> />
  into_keys.<for_each>
    this.<set_value value values.<get key/>/>
  </for_each>
  this.<remove orig_key/>
  this
</defmethod>
```

The first expression in the method sets `values` to a vector of strings created by splitting the value of the field. The second expression is a `for_each`, which sets a new field to the value of the corresponding value in `values`. The third expression removes the original field, and the last expression returns the new object.

```
<thing
  full_name="Mike D Plusch"
/>.
<split_field "full_name" into_keys=<vector "fname" "mname" "lname"/> />
```

Output in ConciseXML:

```
<thing fname="Mike" mname="D" lname="Plusch"/>
```

Sorting and Reversing

If a list or vector of values is not in the correct order, then the list must be reordered. Water Transform uses `reverse` and `sort` for these tasks.

First create a new instance with a `priorities` field that contains a vector of three string objects. Set the instance to the variable `X`.

```
<set X=  
<thing priorities=<vector "high" "med" "low"/> /> />  
</>
```

The following example calls `reverse` on the `priorities` field and replaces the original value:

```
X.<set priorities=X.priorities.<reverse/> />
```

Output in ConciseXML:

```
<thing priorities=<vector "low" "med" "high"/> />
```

Call `sort` to order the values by name.

```
X.<set priorities=X.priorities.<sort/> />
```

Output in ConciseXML:

```
<thing priorities=<vector "high" "low" "med"/> />
```

Conversions

When converting one object to another object, the way that fields are grouped together will likely be different. It is very common to convert an object with fields in nested objects into an object that has all the fields at one level, and vice-versa.

Creating a Hierarchical Object from a Flat Object

Incoming data is more often “flat” than “nested.” Flat data is data that has no nested object structure. The data can be transferred by using a single Comma Separated Value (CSV). This makes flat data very convenient to transfer. Nested data has some explicit multilevel structure. If the object was a hotel reservation, a flat object might have fields `guest_1_name`, `guest_2_name`, and `guest_3_name`, and a nested object might have a `guests` field that held a vector of three names. Before an application can manipulate an object, a flat object needs to be transformed into the appropriate nested structured.

The first example shows how to take an object with flattened address fields and transform it into a nested address object in a single field. The address in the original object is represented using three fields: `street`, `city`, and `state`. The desired structure requires that those three fields appear in a single object that is the value of the `address` field. The `fullname` field remains at the top level. The following example shows the original object and the desired output:

```
<thing fullname="Mike Plusch"
  street="18 Park"
  city="Boston"
  state="MA"
/>
```

Desired Output:

```
<thing fullname="Mike Plusch"
  address=<thing street="18 Park"
    city="Boston"
    state="MA"
  />
/>
```

First, put the object to be transformed in a local variable named *X*.

```
<set X=
  <thing fullname="MP" street="18 Park" city="Boston" state="MA" />
/>
```

The following example shows the transformation process to go from a flat to a nested object:

```
X.<set address=<thing street=X.street city=X.city state=X.state/> />
X.<remove "street" "city" "state"/>
X
```

Output in ConciseXML:

```
<thing fullname="Mike Plusch"
  address=<thing street="18 Park"
    city="Boston"
    state="MA"
  />
/>
```

Flattening a Nested Object

The opposite of the preceding section is how to transform a nested object into a flat object. The following example shows the original nested object and the desired output that is flat:

```
<thing fullname="Mike Plusch"
  address=<thing street="18 Park"
    city="Boston"
    state="MA"
  />
/>
```

Desired Output:

```
<thing fullname="Mike Plusch"
  street="18 Park"
  city="Boston"
  state="MA"
/>
```

The following example is the transform:

```
X.<set street=X.address.street
  city=X.address.city
  state=X.address.state
/>
X.<remove "address"/>
X
```

Output in ConciseXML:

```
<thing fullname="Mike Plusch"
  street="18 Park"
  city="Boston"
  state="MA"
/>
```

Transforming the Class or Type

The transformations in this chapter have used generic instances of `thing`. In most cases, an object is of a particular class or type. The following example shows the original object as a `customer` object, and the desired object as a `client` object:

```
<customer name="MP"/> → <client name="MP"/>
```

Executing the preceding example returns an error because `customer` and `client` have not been defined. The following code defines simple classes for `customer` and `client` and assigns a new instance of `customer` into the local variable `x`:

```
<defclass customer name/>
<defclass client name/>
<set x=<customer name="MP"/> />
```

The following example is the transformation from `customer` to `client`:

```
X.<set _parent=client/> → <client name="MP"/>
```

The transformation was simply the setting of the `_parent` field to the class object `client`. Now the object will inherit any methods or data from the `client` class, not the `customer` class.



See Chapter 22 for details on inheritance.

Creating New Objects During Transformation

The transformations described up to this point have all transformed the original object, rather than creating a new object. This is likely the common case for most transformations. Creating a new object, rather than modifying the original object, makes sense in some cases. First, if only a small subset of the fields from the original object are used, then it makes more sense to copy over the specific fields from the original object into a new object. A second case for copying is if the original object needs to be saved for comparison or auditing. A third case is if the original object was received from a partner and the same object will be sent back to the partner with only a minimal number of changes.

The original object and the desired new object are shown below.

```
<purchase_order
  po_number="1001"
  po_date="2000-10-20"
  status="requires approval"
  line_items=<vector
    <thing part="engine" quan=1/>
    <thing part="wheels" quan=4/>
  />
  total=200
/>
```

Output in ConciseXML:

```
<purchase_order
  po_number="1001"
  status="requires approval"
/>
```

If the `original` variable holds the original object, then the following code creates a new object and copies over specific fields from the original object:

```
<purchase_order
  po_number=original.po_number
  status=original.status
/>
```

If only a small subset of the fields from the original object are required in the new object, then creating a new object with the specific fields is the most efficient technique.

The following example produces the same result as above. Instead of copying specific fields from the original, it copies all the fields from the `original` object and removes the ones that are not required. This makes more sense if most of the original fields are required in the new object.

```
<set a_new=original.<copy/> />
a_new.<remove "po_date" "line_items" "total"/>
a_new
```

Output in ConciseXML:

```
<purchase_order
  po_number="1001"
  status="requires approval"
/>
```

The call to `copy` returns a copy of the object.

Sending Back a Modified Original

A partner might send you a purchase order, but you can only make changes to specific fields of the purchase order before sending it back. Only the `status` field, for example, may be changeable.

If your application needs to process the purchase order, the transformation of the partner's purchase order into your internal structure must not lose any information. This makes it possible to reconstruct the original purchase order after processing to send back to the partner.

A useful technique is to keep the original object, process only a subset of the original object, and copy back specific fields into the original object.

The original object is shown below and will be put into the `original` local variable.

```
<set original=
  <purchase_order
    po_number="1001"
    po_date="2000-10-20" <!-- field not used -->
    status="requires approval" <!-- field to be changed -->
    line_items=<vector <!-- field not used -->
      <thing part="engine" quan=1/>
      <thing part="wheels" quan=4/>
    />
    total=200
  />
/>
```

For internal processing, a new purchase order is created that copies a few fields from the original purchase order.

```

<set internal_po=
  <purchase_order
    orig_po_number=original.po_number
    invoice_number="2002" <!-- new field -->
    status=original.status
    total=original.total
  />
/>

```

The internal processing will likely change some fields. The `status` field, for example, may change.

```
internal_po.<set status="APPROVED"/>
```

After internal processing, only specific fields from the `internal_po` are copied back to the original purchase order. In this particular case, only the `status` field is changed in the original object. This makes it much easier to guarantee that no other changes were made to the original.

```
original.<set status=internal_po.status/>
```

Output in ConciseXML:

```

<purchase_order
  po_number="1001"
  po_date="2000-10-20" <!-- field not used -->
  status="APPROVED" <!-- field to be changed -->
  line_items=<vector <!-- field not used -->
    <thing part="engine" quan=1/>
    <thing part="wheels" quan=4/>
  />
  total=200
/>

```

The partner gets back their original object with only changes to specific fields. Any fields that are not understood by the recipient are untouched. This process can easily handle changes to the incoming data format without requiring any changes to the recipient's code.

Summary

This chapter covered many common cases for transforming data. Water Transform is a general-purpose data transformation language that can use any standard or custom Water method to create special purpose transformations. Chapter 8 shows how Water Import can convert structured non-XML data to XML Water objects.