Dataweave Operators can be tested through postman with below URL links. The expected input is given on right hand side and for more information about the functions and expected output you can check the reference link given in each of the transform components.

|  |  |
| --- | --- |
| **Test Url** | **Test Input Data** |
| <http://localhost:8084/dwexamplesinpxml?examplexml=simplemap> | %dw 1.0  %output application/json  ---  {  address1: payload.order.buyer.address,  city: payload.order.buyer.city,  country: payload.order.buyer.nationality,  email: payload.order.buyer.email,  name: payload.order.buyer.name,  postalCode: payload.order.buyer.postCode,  stateOrProvince: payload.order.buyer.state  } |
| <http://localhost:8084/muledwclass2?exampleiojson=arraymap>  The as operator is also featured in this example. Take a list of books (JSON) and use DataWeave to output a JSON version of the book list. It goes through each <book> element from the input, simply using the map operator. The as operator ensures the transformation generates the correct type for each element (used for type coercion.  We can also name the index and value as given below  items: payload.books map (nextBook, nextIndex) -> {  type: "book",  price: nextBook.price as :number,  properties: {  title: nextBook.title,  author: $.author,  year: nextBook.year as :number  }  }  If the mapping expression was not using object constructor curly braces { }, you would want to use evaluation parentheses to define the scope of the mapping parameters. Even with the object constructor curly braces { }, it is fine to enclose the mapping parameters and the mapping expression in a pair of evaluation parentheses ( ), such as:  items: payload.books map ( (nextBook, nextIndex) -> {  type: "book",  price: nextBook.price as :number,  properties: {  title: nextBook.title,  author: $.author,  year: nextBook.year as :number  }  }  ) | {  "books": [  {  "-category": "cooking",  "title": {  "-lang": "en",  "#text": "Everyday Italian"  },  "author": "Giada De Laurentiis",  "year": "2005",  "price": "30.00"  },  {  "-category": "children",  "title": {  "-lang": "en",  "#text": "Harry Potter"  },  "author": "J K. Rowling",  "year": "2005",  "price": "29.99"  },  {  "-category": "web",  "title": {  "-lang": "en",  "#text": "XQuery Kick Start"  },  "author": [  "James McGovern",  "Per Bothner",  "Kurt Cagle",  "James Linn",  "Vaidyanathan Nagarajan"  ],  "year": "2003",  "price": "49.99"  },  {  "-category": "web",  "-cover": "paperback",  "title": {  "-lang": "en",  "#text": "Learning XML"  },  "author": "Erik T. Ray",  "year": "2003",  "price": "39.95"  }  ]  } |
| <http://localhost:8084/dwexamplesioxml?examplexml=removekey>  If the input contains sensitive information that should be removed. The transform replicates the inbound structure but uses a simple remove operator to take away specific key:value pairs.  The example goes through the whole set of elements in the input using the map operator | <users>  <user>  <personal\_information>  <first\_name>Emiliano</first\_name>  <middle\_name>Romoaldo</middle\_name>  <last\_name>Lesende</last\_name>  <ssn>001-08-84382</ssn>  </personal\_information>  <login\_information>  <username>3miliano</username>  <password>mypassword1234</password>  </login\_information>  </user>  <user>  <personal\_information>  <first\_name>Mariano</first\_name>  <middle\_name>Toribio</middle\_name>  <last\_name>de Achaval</last\_name>  <ssn>002-05-34738</ssn>  </personal\_information>  <login\_information>  <username>machaval</username>  <password>mypassword4321</password>  </login\_information>  </user>  </users> |
| <http://localhost:8084/dwexamplesioxml?examplexml=replace>  Similar to above example but instead of removing the key:value of the sensitive information this replaces it by asterisk (\*\*\*\*\*).  The example goes through the whole set of elements in the input using the map operator | <users>  <user>  <personal\_information>  <first\_name>Emiliano</first\_name>  <middle\_name>Romoaldo</middle\_name>  <last\_name>Lesende</last\_name>  <ssn>001-08-84382</ssn>  </personal\_information>  <login\_information>  <username>3miliano</username>  <password>mypassword1234</password>  </login\_information>  </user>  <user>  <personal\_information>  <first\_name>Mariano</first\_name>  <middle\_name>Toribio</middle\_name>  <last\_name>de Achaval</last\_name>  <ssn>002-05-34738</ssn>  </personal\_information>  <login\_information>  <username>machaval</username>  <password>mypassword4321</password>  </login\_information>  </user>  </users> |
| <http://localhost:8084/dwexamplesjsonxml?examplejson=addattr>  This example adds(injects) the attributes to input Json and creates the xml | [  {  "item": {  "type": "book",  "price": 30,  "properties": {  "title": "Everyday Italian",  "author": [  "Giada De Laurentiis"  ],  "year": 2005  }  }  },  {  "item": {  "type": "book",  "price": 29.99,  "properties": {  "title": "Harry Potter",  "author": [  "J K. Rowling"  ],  "year": 2005  }  }  },  {  "item": {  "type": "book",  "price": 49.99,  "properties": {  "title": "XQuery Kick Start",  "author": [  "James McGovern",  "Per Bothner",  "Kurt Cagle",  "James Linn",  "Vaidyanathan Nagarajan"  ],  "year": 2003  }  }  },  {  "item": {  "type": "book",  "price": 39.95,  "properties": {  "title": "Learning XML",  "author": [  "Erik T. Ray"  ],  "year": 2003  }  }  }  ] |
| <http://localhost:8084/dwexamplesjsonxml?examplejson=optfields>  This example adds the optional fields based on input fields by checking whether or not they exist in the input data. | [  {  "name" : "Julian",  "gender" : "Male",  "age" : 41,  "insurance": "Osde"  },  {  "name" : "Mariano",  "gender" : "Male",  "age" : 33  }  ] |
| <http://localhost:8084/dwexamplesiojson?examplejson=renamekey>  This is a good example of mapobject where we traverse through the key:value pair.  This takes in JSON input data and keeps most of the keys same but renames some of the keys to more meaningful names.  This example also shows the use of the as operator to coerce its type to string. | {  "flights":[  {  "availableSeats":45,  "airlineName":"Ryan Air",  "aircraftBrand":"Boeing",  "aircraftType":"737",  "departureDate":"12/14/2017",  "origin":"BCN",  "destination":"FCO"  },  {  "availableSeats":15,  "airlineName":"Ryan Air",  "aircraftBrand":"Boeing",  "aircraftType":"747",  "departureDate":"08/03/2017",  "origin":"FCO",  "destination":"DFW"  }]  } |
| <http://localhost:8084/dwexamplesxmljson?examplexml=constdirectives>  Converts an XML input to a JSON output that is structured differently and that contains URL links that are built from concatenating input content defining a few constant directives in The DataWeave Header. The transform also creates a few fields that are conditional and are only present in the output when they exist in the input | <ns0:getItemsResponse xmlns:ns0="http://www.alainn.com/SOA/message/1.0">  <ns0:PageInfo>  <pageIndex>0</pageIndex>  <pageSize>20</pageSize>  </ns0:PageInfo>  <ns1:Item xmlns:ns1="http://www.alainn.com/SOA/model/1.0">  <id>B0015BYNRO</id>  <type>Oils</type>  <name>Now Foods LANOLIN PURE</name>  <images>  <image type="SwatchImage">http://ecx.images-amazon.com/images/I/11Qoe774Q4L.\_SL30\_.jpg  </image>  </images>  </ns1:Item>  <ns1:Item xmlns:ns1="http://www.alainn.com/SOA/model/1.0">  <id>B002K8AD02</id>  <type>Bubble Bath</type>  <name>Deep Steep Honey Bubble Bath</name>  <summary>Disclaimer: This website is for informational purposes only.  Always check the actual product label in your possession for the most  accurate ingredient information due to product changes or upgrades  that may not yet be reflected on our web site. These statements made  in this website have not been evaluated by the Food and Drug  Administration. The products offered are not intended to diagnose,  treat  </summary>  <images>  <image type="SwatchImage">http://ecx.images-amazon.com/images/I/216ytnMOeXL.\_SL30\_.jpg  </image>  </images>  </ns1:Item>  <ns1:Item xmlns:ns1="http://www.alainn.com/SOA/model/1.0">  <id>B000I206JK</id>  <type>Oils</type>  <name>Now Foods Castor Oil</name>  <summary>One of the finest natural skin emollients available</summary>  <images>  <image type="SwatchImage">http://ecx.images-amazon.com/images/I/21Yz8q-yQoL.\_SL30\_.jpg  </image>  </images>  </ns1:Item>  <ns1:Item xmlns:ns1="http://www.alainn.com/SOA/model/1.0">  <id>B003Y5XF2S</id>  <type>Chemical Hair Dyes</type>  <name>Manic Panic Semi-Permanent Color Cream</name>  <summary>Ready to use, no mixing required</summary>  <images>  <image type="SwatchImage">http://ecx.images-amazon.com/images/I/51A2FuX27dL.\_SL30\_.jpg  </image>  </images>  </ns1:Item>  <ns1:Item xmlns:ns1="http://www.alainn.com/SOA/model/1.0">  <id>B0016BELU2</id>  <type>Chemical Hair Dyes</type>  <name>Herbatint Herbatint Permanent Chestnut (4n)</name>  <images>  <image type="SwatchImage">http://ecx.images-amazon.com/images/I/21woUiM0BdL.\_SL30\_.jpg  </image>  </images>  </ns1:Item>  </ns0:getItemsResponse> |
| <http://localhost:8084/dwexamplesioxml?examplexml=math>  This example takes an XML input and parses it into a different XML arrangement. After a single <header> element is copied, a map operation carries out the same steps for each 'item': several fields are passed on without any changes, then the discount and subtotal fields are calculated with references to constants defined in the header directives of the transform. A single set of subtotal, tax and total elements are created by performing a reduce operation over all of the items in the "items" array, performing calculations that sometimes involve constants defined in the header. The as operator is also used to coerce to a number and then performs basic math on these numbers. | <invoice>  <header>  <customer\_name>ACME, Inc.</customer\_name>  <customer\_state>CA</customer\_state>  </header>  <items>  <item>  <description>Product 1</description>  <quantity>2</quantity>  <unit\_price>10</unit\_price>  </item>  <item>  <description>Product 2</description>  <quantity>1</quantity>  <unit\_price>30</unit\_price>  </item>  </items>  </invoice> |
| <http://localhost:8084/dwexamplesxmljson?examplexml=groupby>  For now its not working **PENDING** | <school>  <teachers>  <teacher>  <name>Mariano</name>  <lastName>De Achaval</lastName>  <subject>DW</subject>  </teacher>  <teacher>  <name>Emiliano</name>  <lastName>Lesende</lastName>  <subject>DW</subject>  </teacher>  <teacher>  <name>Leandro</name>  <lastName>Shokida</lastName>  <subject>Scala</subject>  </teacher>  </teachers>  <students>  <student>  <name>Peter</name>  <lastName>Parker</lastName>  <hobby>DW</hobby>  <hobby>Scala</hobby>  </student>  <student>  <name>Homer</name>  <lastName>Simpson</lastName>  <hobby>Scala</hobby>  </student>  </students>  </school> |
| <http://localhost:8084/dwexamplesjsonxml?examplejson=mulinputs>  Here we have 3 different input JSON files, these three all arrive in one single Mule message, occupying  the payload and two flow variables. The payload contains an array of book objects, one flow variable has a set of currency exchange rates, and the other one a query.  The transform filters the first input using the conditions passed in the third input, then performs a map to deal with each remaining object separately. Within this map, it defines two variables: it  and props. Through the use of @, attributes are added into the XML tags. A second map operation inside the first one calculates  the price of each book for each of the currencies provided in the second input. Another map operation displays each element  in the author array as a separate <author></author> tag. | [  {  "item": {  "type": "book",  "price": 30,  "properties": {  "title": "Everyday Italian",  "author": [  "Giada De Laurentiis"  ],  "year": 2005  }  }  },  {  "item": {  "type": "book",  "price": 29.99,  "properties": {  "title": "Harry Potter",  "author": [  "J K. Rowling"  ],  "year": 2005  }  }  },  {  "item": {  "type": "book",  "price": 49.99,  "properties": {  "title": "XQuery Kick Start",  "author": [  "James McGovern",  "Per Bothner",  "Kurt Cagle",  "James Linn",  "Kurt Cagle",  "Vaidyanathan Nagarajan"  ],  "year": 2003  }  }  },  {  "item": {  "type": "book",  "price": 39.95,  "properties": {  "title": "Learning XML",  "author": [  "Erik T. Ray"  ],  "year": 2003  }  }  }  ]  {  "USD": [  {"currency": "EUR", "ratio":0.92},  {"currency": "ARS", "ratio":8.76},  {"currency": "GBP", "ratio":0.66}  ]  }  {  "publishedAfter": 2004  } |
| <http://localhost:8084/dwexamplesiojson?examplejson=mergeinputs>  This example uses multiple payloads and merges them into the output. This is also a very good example of scatter-gather. Here we have 2 transform components in the scatter-gather(sg) and send the 2 different payloads to the transform component outside this sg component. | **Input 1:**  [  { "bookId":"101",  "title":"world history",  "year":"2017",  "isbn":"11111",  "price":"19.99"  },  {  "bookId":"202",  "title":"the great outdoors",  "year":"2016",  "isbn":"22222",  "price":"15.99"  }  ]  **Input 2:**  [  {  "bookId":"101",  "author":"john doe"  },  {  "bookId":"202",  "author":"jane doe"  }  ] |
| This example shows removal of an attribute as well as recursive function call to check all the attributes.  The example defines a function in the dataweave header and then calls it on the body. It goes through the whole set of elements in the input using a mapObject operator. Note that it differs from the map operation in that map object processes both key and value of its input, rather than just the keys. It also uses the when and otherwise operators to pick out only instances where an XML attribute exists, and to do nothing otherwise. It also uses the match operator.  See Attribute selector expressions for how to access XML attributes in DataWeave | <users>  <user username="Julian" password="1234"/>  <user username="Mariano" password="4321"/>  </users> |