**Dataweave Language Tutorial**

1. **To generate an json/java array of integers from 1 to 500**

%dw 2.0

output application/json

---

1 to 500

Output:

[

1,

2,

3,

.

.

.,

500

]

%dw 2.0

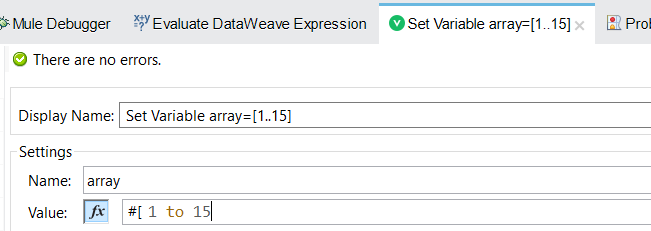
output application/java

---

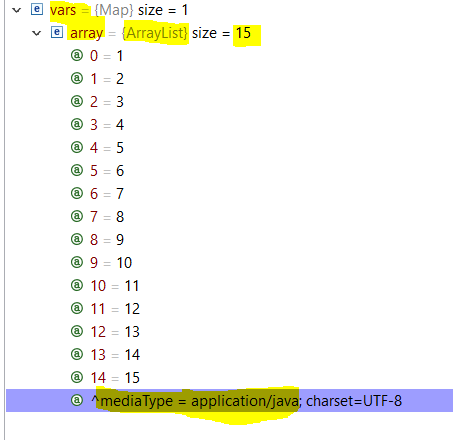
1 to 500

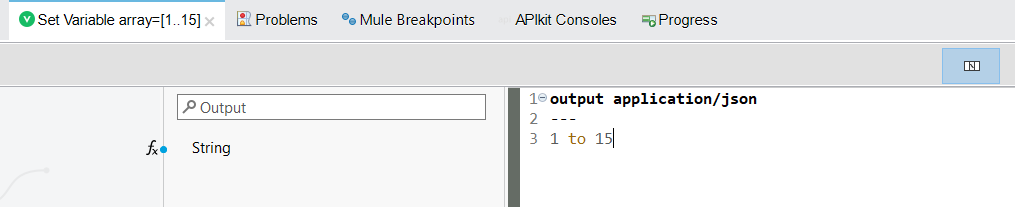
Output:

[1, 2, 3, 4, 5, 6,...,500]

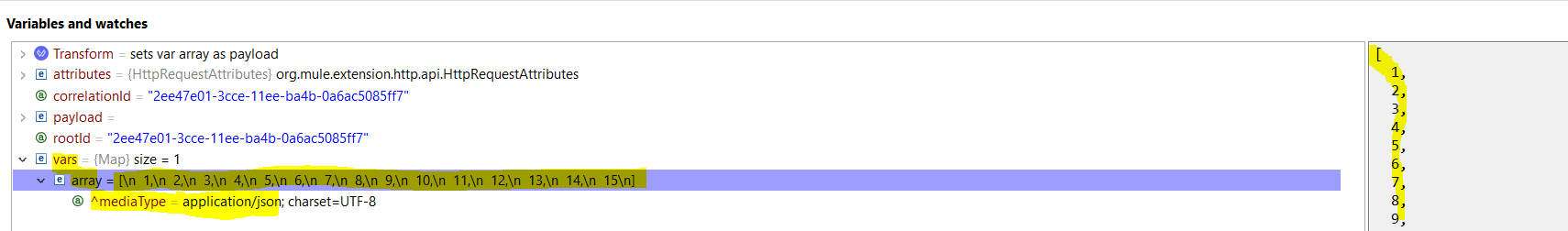


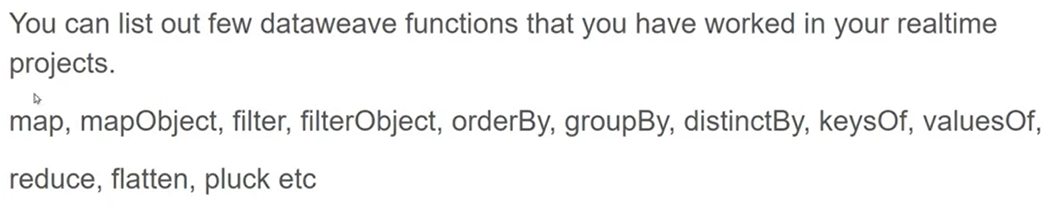
Output:





**Output:**





**2023-Accenture Interview Input:**

[

{

"Email": "rose@edge.com",

"FirstName": "Rose",

"Phone": "(512) 757-6000",

"Id": "0035i00000Am90vAAB",

"LastName": "Gonzalez",

"dept": "HR",

"Salary": 1000

},

{

"Email": "sean@edge.com",

"FirstName": "Sean",

"Phone": "(512) 757-6000",

"Id": "0035i00000Am90wAAB",

"LastName": "Forbes",

"dept": "Sales",

"Salary": 5000

},{

"Email": "surya@edge.com",

"FirstName": "Surya",

"Phone": "(877) 757-6000",

"Id": "0012300000Am90wAAB",

"LastName": "Shaw",

"dept": "IT",

"Salary": 10000

}

]

Script:

//DWL Program to skip the employee object if his salary is less than 1000 n give empsrno starting from 1 instead of 0 n concatenate firstname+ lastname as Name n order by empsal in desc order

%dw 2.0

output application/json

---

payload filter ((item, index) -> item.Salary>1000 ) map ((item, index) ->

{

Empsrno: (index+1),

Empid: item.Id,

Empname: item.FirstName ++" " ++ item.LastName,

Empdept: item.dept,

EmpSal: item.Salary,

Empemail: item.Email,

Empphone: item.Phone

}

) orderBy ((item, index) -> item.EmpSal[-1 to 0] )

Output:

[

{

"Empsrno": 2,

"Empid": "0012300000Am90wAAB",

"Empname": "Surya Shaw",

"Empdept": "IT",

"EmpSal": 10000,

"Empemail": "surya@edge.com",

"Empphone": "(877) 757-6000"

},

{

"Empsrno": 1,

"Empid": "0035i00000Am90wAAB",

"Empname": "Sean Forbes",

"Empdept": "Sales",

"EmpSal": 5000,

"Empemail": "sean@edge.com",

"Empphone": "(512) 757-6000"

}

]

**2023-Cognizant Interview Input1:**

[

{

"id": 4429,

"name": "Wendy Crafts",

"type": "business",

"address": "982 Cherry Ave,Chennai,TN,25362,india",

"air\_miles": 90000,

"created\_at": "2010-01-06T00:42:34"

},

{

"id": 4430,

"name": "Robert Barath",

"type": "business",

"address": "12354 Foo Street,Boulder,Colorado,15361,Murrica",

"air\_miles": 55240,

"created\_at": "2006-07-20T21:39:45"

},

{

"id": 4444,

"name": "Wendy Crafts",

"type": "personal",

"address": "982 Cherry Ave,Chennai,TN,25362,india",

"air\_miles": 10000,

"created\_at": "2010-01-06T00:42:34"

}

]

Script:

//DWL Program to convert json to xml

%dw 2.0

output application/xml

---

accounts: {

(payload map ( item , index ) -> {

account @(id: item.id , air\_miles: item.air\_miles): {

id: item.id,

name: item.name,

'type': item.'type',

address: item.address,

air\_miles: item.air\_miles,

createdAt: item.created\_at

}

})

}

output:

<?xml version='1.0' encoding='UTF-8'?>

<accounts>

<account id="4429" air\_miles="90000">

<id>4429</id>

<name>Wendy Crafts</name>

<type>business</type>

<address>982 Cherry Ave,Chennai,TN,25362,india</address>

<air\_miles>90000</air\_miles>

<createdAt>2010-01-06T00:42:34</createdAt>

</account>

<account id="4430" air\_miles="55240">

<id>4430</id>

<name>Robert Barath</name>

<type>business</type>

<address>12354 Foo Street,Boulder,Colorado,15361,Murrica</address>

<air\_miles>55240</air\_miles>

<createdAt>2006-07-20T21:39:45</createdAt>

</account>

<account id="4444" air\_miles="10000">

<id>4444</id>

<name>Wendy Crafts</name>

<type>personal</type>

<address>982 Cherry Ave,Chennai,TN,25362,india</address>

<air\_miles>10000</air\_miles>

<createdAt>2010-01-06T00:42:34</createdAt>

</account>

</accounts>

**Input2:**

[

{

"id": 4429,

"name": "Wendy Crafts",

"type": "business",

"address": "982 Cherry Ave,Chennai,TN,25362,india",

"air\_miles": 90000,

"created\_at": "2010-01-06T00:42:34",

"sury": "kant"

},

{

"id": 4430,

"name": "Robert Barath",

"type": "business",

"address": "12354 Foo Street,Boulder,Colorado,15361,Murrica",

"air\_miles": 55240,

"created\_at": "2006-07-20T21:39:45"

},

{

"id": 4444,

"name": "Wendy Crafts",

"type": "personal",

"address": "982 Cherry Ave,Chennai,TN,25362,india",

"air\_miles": 10000,

"created\_at": "2010-01-06T00:42:34"

}

]

Script:

//prgram to print the input array of object without id field i.e 0th index field dynamically means not writing like name: item.name,'type': item.type .......... in map operator

%dw 2.0

output application/json

---

payload map ((item, index) ->

item filterObject ((value, key, index) -> index >0 )

)

Output:

[

{

"name": "Wendy Crafts",

"type": "business",

"address": "982 Cherry Ave,Chennai,TN,25362,india",

"air\_miles": 90000,

"created\_at": "2010-01-06T00:42:34",

"sury": "kant"

},

{

"name": "Robert Barath",

"type": "business",

"address": "12354 Foo Street,Boulder,Colorado,15361,Murrica",

"air\_miles": 55240,

"created\_at": "2006-07-20T21:39:45"

},

{

"name": "Wendy Crafts",

"type": "personal",

"address": "982 Cherry Ave,Chennai,TN,25362,india",

"air\_miles": 10000,

"created\_at": "2010-01-06T00:42:34"

}

]

**2023-Capgemini Interview (L1 Round)**

**Input:**

[

{

"name": "Roger"

},

{

"name": "Michael"

},

{

"name": "Harris"

}

]

Script:

//prgram to print key as "user 1" for every name oject starting with index 1,2,3,

%dw 2.0

output application/json

---

payload map ((item, index) ->

{

("user " ++ (index +1) ): item.name

} )

Or

//prgram to print key as "user 1" for every name oject starting with index 1,2,3,

%dw 2.0

output application/json

---

payload map{

("user " ++ ($$ +1)) : $.name

}

Output:

[

{

"user 1": "Roger"

},

{

"user 2": "Michael"

},

{

"user 3": "Harris"

}

]

**Round 2:**

**Input1**

%dw 2.0

output application/json

var array={"1": "abc", "2": "xyz"}

---

Script:

//Program to print key of array object i.e 1 n 2 in an array like this

[

"1",

"2"

]

%dw 2.0

output application/json

var array={"1": "abc", "2": "xyz"}

---

array pluck ((value, key, index) -> key )

Output:

[

"1",

"2"

]

**Input2**

%dw 2.0

output application/json

var array1=["1", "2"]

var obj1={"1": "abc", "2": "xyz", "3": "pqr"}

---

Script:

//Program to print matching values of object obj1 i.e abc n xyz in an array like this

[

"abc",

"xyz"

]

Where the key of object obj1 is equal to elements of an array array1

Script:

%dw 2.0

output application/json

var array1=["1", "2"]

var obj1={"1": "abc", "2": "xyz", "3": "pqr"}

---

//obj1 mapObject( (($$): $) if array1 contains $$ as String) pluck $

// both will be working as both are same

obj1 mapObject ((value, key, index) -> ((key) : value) if array1 contains key as String ) pluck ((value, key, index) -> value )

// o pluck ( (value, key, index) -> if (a contains key as String) value else null)

Output:

[

"abc",

"xyz"

]

**2023-NeuraFlash Interview**

**Input:**

["A", "C", "T"]

["A", "B","C"]

Script:

//program to print common-elements,uncommon-elements from an array

%dw 2.0

output application/json

var a1 = ["A", "C", "T"]

var a2 = ["A", "B","C"]

---

{

"Duplicate\_Elements": a1 filter ((item, index) -> a2 contains item ) ,

"Not\_Duplicate\_Elements": ((a1 filter ((item, index) -> !(a2 contains item) )) ++ (a2 filter ((item, index) -> !(a1 contains item) )))

}

Output:

{

"Duplicate\_Elements": [

"A",

"C"

],

"Not\_Duplicate\_Elements": [

"T",

"B"

]

}

%dw 2.0

output application/json

var a1 = ["A", "C", "T"]

var a2 = ["A", "B","C"]

var a3= [1,5,7,6,8]

var a4= [7,9,1,3]

var a5=[1,3,2,1,4,5,2]

var a6= a1 ++ a2

---

{

"common-elemnts":

a1 filter ((item, index) -> a2 contains item ),

"commmon-elementsd": a1 ++ a2 groupBy ((item, index) -> item ) pluck ((value, key, index) -> value) map ((item, index) -> if(sizeOf(item)>1) item[0] else "" ) filter ((item, index) -> !isEmpty(item) ),

"non-common-elements":

a1 filter ((item, index) -> !(a2 contains item) ),

"common-elemnts":

a3 filter ((item, index) -> a4 contains item ),

"non-common-elements":

a3 filter ((item, index) -> !(a4 contains item) ),

"non-common-elements":

a4 filter ((item, index) -> !(a3 contains item) ),

"non-common": a4 -- a3,

"non-common": a3 -- a4,

"counting\_of\_duplicate\_Numbers\_in\_an\_array" : (a5 groupBy ((item, index) -> item ) pluck $ map ((item, index) ->

{ ("Duplicate\_number": item[0]) if(sizeOf(item)>1),

( "count\_Of\_Duplicate\_No":sizeOf(item)) if(sizeOf(item)>1)

} )) filter ((item, index) -> !(isEmpty(item)) ),

"counting\_of\_duplicate\_strings\_in\_an\_array": (a1++ a2 groupBy ((item, index) -> item ) pluck ((value, key, index) -> value ) map ((item, index) ->{

("Duplicate\_string": item[0]) if(sizeOf(item)>1),

( "count\_Of\_Duplicate\_string":sizeOf(item)) if(sizeOf(item)>1)

} )) filter ((item, index) -> !(isEmpty(item)) ),

"counting\_of\_notduplicate\_strings\_in\_an\_array": (a1++ a2 groupBy ((item, index) -> item ) pluck ((value, key, index) -> value ) map ((item, index) ->{

("Duplicate\_string": item[0]) if(sizeOf(item)>1),

( "count\_Of\_Duplicate\_string":sizeOf(item)) if(sizeOf(item)>1)

} )) filter ((item, index) -> !(isEmpty(item)) )

}

**2023-Infosys Interview**  
**Input:**

<order>

<product-lineitems>

<product-lineitem>

<net-price>100.0</net-price>

</product-lineitem>

<product-lineitem>

<net-price>498.00</net-price>

</product-lineitem>

</product-lineitems>

</order>

**Expected Output:**

{

"order": {

"product-lineitems": {

"product-lineitem": [

{

"net-price": "100.0"

},

{

"net-price": "498.00"

}

]

}

}

}

Script:

%dw 2.0

output application/json

---

{

"order":

{

"product-lineitems":

{

"product-lineitem": payload.order."product-lineitems" pluck ((value, key, index) -> value)

}

}

}

Output:

{

"order": {

"product-lineitems": {

"product-lineitem": [

{

"net-price": "100.0"

},

{

"net-price": "498.00"

}

]

}

}

}

Dataweave Tutorial:

Datatypes Supported in Dataweave:

* String Strings are defined between quotes.  eg: “surya”
* Number Number type, covering both integer and floating-point numbers. eg: 1 ,1.234
* Boolean Boolean type only has two values: true and false eg: true
* Array Arrays are an ordered series of values where the values can be of any type: eg: [1,”surya”]
* Object Objects are a series of key-value mappings, where the value can be of any type: eg: {“name” : “surya”}

**valuesOf**

**map**

**flatten**

**reduce**

**mapObject**

**filter**

**filterObject**

**isBlank**

**sizeOf**

**isEmpty**

**typeOf():** You can check the type of a value by using typeOf

Script:

%dw 2.0

output application/json

var a= "surya"

var b= 1234

var c= true

var d= ["surya"]

var e= {"id": 1234}

---

{

"typeofa": typeOf(a),

"typeofb": typeOf(b),

"typeofc": typeOf(c),

"typeofd": typeOf(d),

"typeofe": typeOf(e)

}

Output:

{

"typeofa": "String",

"typeofb": "Number",

"typeofc": "Boolean",

"typeofd": "Array",

"typeofe": "Object"

}

selectors are as a way to query your data.

* Single-value selector: . The single-value selector (.) allows you to lookup Object values by their key.

Eg:

Input:

{

"environment": {

"stage": "prod"

},

"host": "linux"

}

Script:

%dw 2.0

output application/json

---

payload.environment.stage

output:

"prod"

* Index selector: [n] to traverse Arrays with the index selector ([n]). Use the index selector to get to a value in an Array based on its position from the beginning of the Array:

Eg:

Input:

["prod", ["qa", "dev"]]

Script:

%dw 2.0

output json

---

payload[1][0]

output:

"qa"

* Range selector: [n to m] If you need multiple sequential values from an Array, DataWeave allows you to select a range of values with the range selector ([n to m]). Instead of returning a single value like the index selector does, it will return an Array of values:

Eg:

Input:

["prod", "qa", "dev"]

Script:

%dw 2.0

output application/json

---

payload[ -1 to 0]

Output:

[

"dev",

"qa",

"prod"

]

* Multi-value selector: .\* The multi-value selector (.\*) returns an Array containing any value that matches the key.

Eg:

Input:

<movies>

<title>The Terminator</title>

<title>Titanic</title>

<title>Avatar</title>

<director>James Cameron</director>

</movies>

Script:

%dw 2.0

output application/json

---

payload.movies.\*title

Output:

[

"The Terminator",

"Titanic",

"Avatar"

]

* Descendants selector: .. The descendants selector (..) is the perfect tool to use when you need the values for a certain key no matter where they appear in a piece of data.

Input:

<?xml version='1.0' encoding='UTF-8'?>

<customer>

<name>Max the Mule</name>

<locations>

<name>Max’s house</name>

<address>1234 Mule Street</address>

</locations>

<locations>

<name>Max's cabin</name>

<address>345 Lakeside Road</address>

</locations>

</customer>

Script:

%dw 2.0

output json

---

payload..name

output:

[

"Max the Mule",

"Max’s house",

"Max's cabin"

]

Logical operators allow you to write true/false kind of expression.

Operators

| **Expression** | **Label** |
| --- | --- |
| A > B | Greater than |
| A < B | Less than |
| A >= B | Greater than or equal to |
| A <= B | Less than or equal to |
| A == B | Equal to |
| A != B | Not equal to |
| A ~= B | Similar to |
| not A | Logical negation |
| !A | Logical negation |
| A and B | Logical and |
| A or B | Logical or |

Flow control is used when you want to execute certain parts of your code in some situations, while not executing others. In other words, it’s a way to add logic to your scripts.

# **If Else**

If/else expressions allow you to make decisions using logical operators and branch as a result.

Eg:

Input:

{

"price": 150.00

}

Script:

%dw 2.0

output json

var action = if (payload.price < 100) "buy" else if(payload.price > 140) "sell" else "hold"

---

{

price : payload.price,

action : action

}

Output:

{

"price": 150,

"action": "sell"

}

Pattern matching is another method of flow control,

Syntax:

<input\_expression> match {

case <condition> -> <execute\_if\_condition\_pass>

case <condition> -> <execute\_if\_condition\_pass>

else -> <execute\_if\_no\_condition\_pass>

}

Use pattern matching to return "Light off" when the status for the input below is 0 or "Light on" when the status is 1.

Input:

{

"id": 290505,

"status": 0,

"name": "Cabin outdoor light"

}

Script:

%dw 2.0

output application/json

---

payload.status match{

case 0-> "Light off"

case 1-> "Light on"

else -> "No Action"

}

Output:

"Light off"

Functions:

# **Named Functions**

We create functions in the declarations section of the script using the fun keyword.

**best practices**

**1)Use of remove variable component to remove variables**

**2)standardized way of naming convention for project like wm-sys-invoices-eadm,wm-prc-cust-invoices,wm-prc-customers,wm-exp-invoices**

**3) use of various types of cases like snake\_case,camelCase,kebab-case,PascalCase for defining**

**4) no hard-coding of config values.Externalize the dw,sql script**

**5)Removal of undeclared,unused components,attributes,variables**

**6)use of subflows,flow-references wherever needed.**

**Use Of Various Cases(snake\_case,kebab-case,camelCase):**

1)API RAML Project Name or Anypoint Studio API Project Name should be in lower kebab-case

Eg: wm-sys-invoices-eadm,wm-prc-cust-invoices,wm-exp-cust-invoices

2)Resource name should be either in lower case( if only one word like /customers or /invoices) or it should be in lower-kebab-case( if joining two words like /service-status or /payments-inquiry

3) Uriparam should be declared always in lower\_snake\_case like /{customer\_id} or /{invoice\_id}

4)queryParams should be either in lowercase(if one word like id) or in lower\_snake\_case(if joining two words from\_date or to\_date or ticket\_number)

5) headers should be either in lowercase(if one word like limit) or in lower\_snake-case(if joining two words source\_app or target\_app or request\_tracking\_id)

6)body should be either in lowercase(if one word like id) or in lower\_snake\_case(if joining two words from\_date or to\_date or ticket\_number)

7)response should be either in lowercase(if one word like id,status) or in lower\_snake\_case(if joining two words dispatch\_date or service\_status or ticket\_number)