**nexl scripting language**

**Introduction**

nexl expressions allows to perform a wide variety of data manipulations on a JavaScript primitives, arrays and objects hosted by nexl REST server.

For example :

1. resolves a hosts object and produces XML from that object

${hosts~X}

1. resolves a fruits array, sorts array, joins all elements with comma

${fruits#S&,}

nexl expression definition is

${[JS\_VAR\_NAME][action1][action2]...}

Actions perform data manipulation. nexl engine evaluates actions from the left to the right. Every action in chain gets a value calculated by previous action in chain. Some actions in chain can be ignored in some cases. When actions chain is finished the last calculated value is assigned to whole expression.

JS\_VAR\_NAME is a name of JavaScript variable in nexl source. If JS\_VAR\_NAME is not specified first action in chain will get an undefined value.

Action definition

ACTION\_ID[ACTION\_VALUE]

In our first expression example ${hosts~X} hosts is a name of JavaScript variable and ~X is an action, where ~ is ACTION\_ID ( object operation action ) and X is ACTION\_VALUE ( produce XML )

In the second expression example ${fruits#S&,} we have a fruits JavaScript variable and 2 actions in chain #S&, where #S action sorts fruits array and &, action joins array elements with comma.

Depending on ACTION\_ID, ACTION\_VALUE can be a string and/or another another nexl expression ( inner expression ). For example ${test@${value}} where @ is an ACTION\_ID and ${value} is an ACTION\_VALUE. ${value} is an expression itself and can contain it's own actions

Available actions :

|  |  |
| --- | --- |
| @ | Default value. Applies for undefined values. For all other values is ignored.  ACTION\_VALUE can be a string including internal nexl expressions |
| : | Typecast. Performs typecast for primitive values. Can convert any value to null or undefined  Available ACTOIN\_ID + ACTION\_VALUES  :num, :str, :bool, :null, :undefined |
| . | Object property resolution. Resolves property value for objects. Ignored for all other data types  ACTION\_VALUE can be a string including internal nexl expressions |
| ~ | Object operations. Performs different operations with JavaScript objects. Ignored for all other data types.  Available ACTION\_ID + ACTION\_VALUES  ~K ( resolves object keys ), ~V ( resolves object values ), ~O ( converts to object ), ~X ( produces XML ), ~Y ( produces YAML ), ~P ( produces key-value pairs ) |
| < | Object key reverse resolution. Allows to resolve an JavaScript's object key by value. Ignored for all non object types.  ACTION\_VALUE can be a string including internal nexl expressions |
| # | Array operations. Performs different operations with JavaScript arrays. All other data types are ignored.  Available ACTION\_ID + ACTION\_VALUES  #S ( sorts asc ), #s ( sorts desc ), #U ( unique value ), #D ( duplicate values ), #LEN ( elements count ), #A ( converts to array ), #F ( gets first array element or undefined ) |
| & | Joins JavaScript array elements with ACTION\_VALUE. Ignored for all other data types.  ACTION\_VALUE can be a string including internal nexl expressions |
| + | Add/Merge. Adds elements to existing JavaScript array or merges two JavaScript objects. Ignores for all other data types.  For JavaScript arrays ACTION\_VALUE can be a string including internal nexl expression  For JavaScript object ACTION\_VALUE must be internal nexl expression which is evaluated to JavaScript object |
| - | Eliminates items from JavaScript arrays or JavaScript objects. Ignored for all other data types  ACTION\_VALUE can be a string including internal nexl expressions |
| ^ | String operations  Available ACTION\_ID + ACTION\_VALUES  ^U ( upper case ), ^U1 ( capitalizes first letter ), ^L ( lower case ), ^T ( trims ), ^LEN ( length ) |
| ! | Undefined value operations  !E Converts empty strings, empty arrays and empty objects to undefined  !U Evaluates the whole expression to undefined if one of sub expressions are evaluated to undefined |
| \* | Mandatory value validator action.  Throws exception for undefined value. You can provide a custom error message as ACTION\_VALUE |
| | | Pushes current value to function parameters stack.  Doesn't have an ACTION\_ID |
| % | Reserved for future actions ( must be escaped ) |
| > | Reserved for future actions ( must be escaped ) |
| = | Reserved for future actions ( must be escaped ) |
| ? | Reserved for future actions ( must be escaped ) |

There are 3 additional actions :

* Accessing array elements [ … ]
* Substring [ … ]
* Function call (...)

Actions can be combined in any order without any limit. Actions count is also unlimited.

Please note ACTION\_IDS must be escaped if they are used in ACTION\_VALUES

**Resolving a JavaScript variable value**

The ${distanceToTheMoon} expression resolves a distanceToTheMoon variable value. This variable can be JavaScript primitive, array or object.

If distanceToTheMoon variable is not defined expression will be evaluated to undefined.

Empty expressions ${} are also evaluated to undefined

You can combine expressions with another expressions/strings/numbers etc.

Let's consider the following example

sites = [**'google.com'**, **'w3schools.com'**, **'facebook.com'**];

**urlTemplate** = **'http://www.${sites}'**;

The ${urlTemplate} expression is evaluated to following array

[**'http://www.google.com'**, **'http://www.w3schools.com'**, **'http://www.facebook.com'**]

**Default values action @**

ACTOIN\_ID is @

ACTION\_VALUE can be a string and/or nexl expression(s)

Applies for undefined values

This action replaces undefined value with value you provide in ACTION\_VALUE. Example :

${distanceToTheMoon@384400}

If distanceToTheMoon variable is not defined nexl engine will apply a '384400' default value. The 384400 value is treated as string, but you can cast it to number ( see more about typecast in next section ) :

${distanceToTheMoon@384400:num}

The ${@384400} expression is also evaluated to '384400' because a ${} expression is evaluated to undefined value.

The following expression is evaluated to empty string :

${@}

The default value can be nexl expression itself. For example :

${distanceToTheMoon@${calcDistance}}

If distanceToTheMoon is undefined nexl engine will apply a default value provided by ${calcDistance} expression

If a ${calcDistance} expression is also evaluated to undefined you can provide an additional default value(s) :

${distanceToTheMoon@${calcDistance}@{val1}@${val2}@384400}

Let’s consider the following nexl expression :

${@Hello ${world} 2017}

It has a 'Hello ${world} 2017' default value where ${world} points to world JavaScript variable.

**Typecast :**

ACTOIN\_ID is :

ACTION\_VALUE can be one of the following num, str, bool, null, undefined

Applies for anything ( num, str, bool are applied for primitives; null, undefined for anything )

The following type casts are available

|  |  |  |
| --- | --- | --- |
| ACTION\_ID + ACTION\_VALUE | Description | Example |
| :num | Casts string or boolean items to a number. All other data types are ignored.  Illegal numbers cast to undefined | ${stringItem:num} |
| :str | Casts boolean or numeric items to string. All other data types are ignored | ${boolItem:str} |
| :bool | Casts numeric or string items to boolean. All other data types are ignored.  Zero numbers are casted to false, non zero numbers casted to true.  'true' string items are casted to true, 'false' string items are casted to false. All other string items are casted to undefined | ${numericItem:bool} |
| :null | Any value of any type is converted to null | ${array1:null} |
| :undefined | Any value of any type is converted to undefined | ${obj1:undefined} |

Examples for casting

|  |  |
| --- | --- |
| nexl expression | Explanation |
| ${@} | Evaluates to empty string.  Empty expression ${} is evaluated to undefined value. If we add a default value action it will be applied. In this expression ${@} default value it empty string |
| ${@319:num} | Casts a '319' string value to numeric |
| ${@:num} | Casts an empty string to numeric ( result is undefined ) |
| ${@319:bool} | Casts a '319' string value to boolean ( result is undefined ) |
| ${@319:bool@hello} | Casts a '319' string value to boolean. The result is undefined. Therefore the second default value action is applied.  Final result is 'hello' string |
| ${@319:num:bool} | First casts a '319' string value to numeric, then casts a 319 numeric value to boolean ( result is true ) |
| ${@true:bool} | Casts a 'true' string value to boolean ( result is true ) |
| ${:null} | Converts an undefined value to null |
| ${obj1:null} | Converts obj1 to null |
| ${obj1:num} | If the obj1 is not a primitive value this casting action will be ignored.  :num, :str and :bool actions are applied only for primitives |

**Object property resolution .**

ACTOIN\_ID is .

ACTION\_VALUE can be a string and/or nexl expression(s)

Applies for objects

This action resolves objects properties. For example ${person.city} expression resolves a city property of person object where dot is an ACTOIN\_ID and 'city' is ACTION\_VALUE.

\_this\_ and \_parent\_ are reserved variable names. Do not declare variables with those names in your nexl sources.

\_this\_ points to a current object instance.

\_parent\_ points to a parent object instance.

Let’s consider examples for the following object

**person** = {

**name**: **'Alex'**,

**age**: 25,

**address**: {

**country**: **'Canada'**,

**city**: **'Toronto'**

}

};

|  |  |
| --- | --- |
| nexl expression | Explanation and result |
| ${person.name} | Resolves a name property of person object.  Result is  'Alex' |
| ${person.address.city} | First resolves a person object, then resolves an address property of that object and finally resolves a city property from address object.  Result is  'Toronto' |
| **prop1** = **'address'**;  …  ${person.${prop1}.country} | First resolves a person object. Then resolves his property provided by ${prop1} internal nexl expression which is 'address'. Finally resolves a country property of that object  Result is  'Canada' |
| ${person.${prop2}} | First resolves a person object. Then resolves his property which is provided by ${prop2} internal nexl expression. The prop2 variable is not declared therefore it’s evaluated to undefined. nexl engine skips undefined values when performing object property resolution. Therefore ${prop2} will be skipped and the whole expression will be equaled to a person object  Result is  {  name: **'Alex'**,  age: 25,  **address**: {  country: **'Canada'**,  city: **'Toronto'**  }  }; |
| ${person.${prop2@}} | First resolves a person object. Then resolves his property provided by ${prop2@:null} internal expression. The prop2 variable is not declared and evaluated to undefined. prop2 also has a default value action with empty string value. Therefore ${prop2@} expression will be evaluated to empty string. person object doesn’t have an empty string property. That’s why the whole expression will be evaluated to undefined  Result is  undefined |
| **props** = [**'name'**, **'age'**];  ..  ${person.${props}} | First resolves a person object. Then resolves his property provided by ${props} internal expression. This expression is evaluated to JavaScript array. Therefore nexl engine performs property resolution for each array element  Result is  [**'Alex'**, 25] |

Examples for \_this\_ and \_parent\_

Consider the following object

dirs = {

**ROOT\_DIR**: **'/home/project'**,

**HOME\_DIR**: **'${\_this\_.ROOT\_DIR}/nexl'**,

**SUBDIRS**: {

**LOGS\_DIR**: **'${\_parent\_.HOME\_DIR}/LOGS'**,

**BACKUP\_DIR**: **'${\_parent\_.HOME\_DIR}/BACKUP'**

},

**FILES**: {

**SERVER\_LOG**: **'${\_parent\_.SUBDIRS.LOGS\_DIR}/server.log'**,

**BACKUP\_LOG**: **'${\_parent\_.SUBDIRS.BACKUP\_DIR}/backup.log'**,

**START\_SCRIPT**: **'${\_parent\_.HOME\_DIR}/start.sh'**

}

};

|  |  |
| --- | --- |
| nexl expression | Explanation and result |
| ${dirs.ROOT\_DIR} | First resolves a dirs object and then resolves his ROOT\_DIR property  Result is  '/home/project' |
| ${dirs.HOME\_DIR} | First resolves a dirs object and then resolves his HOME\_DIR property. HOME\_DIR is a nexl expression itself.  Let’s consider a ${\_this\_.ROOT\_DIR} expression. In this expression \_this\_ points to a dirs object instance. Therefore ROOT\_DIR resolves a ROOT\_DIR property of dirs object  Result is  '/home/project/nexl' |
| ${dirs.SUBDIRS.BACKUP\_DIR} | This expression is evaluated to a string value which has a sub expression ${\_parent\_.HOME\_DIR}/BACKUP  \_parent\_ points to a dirs object, therefore HOME\_DIR resolves a HOME\_DIR property of dirs object  Result is  '/home/project/nexl/BACKUP' |
| ${dirs.FILES.BACKUP\_LOG} | Result is  '/home/project/nexl/BACKUP/backup.log' |
| ${dirs.FILES.START\_SCRIPT} | Result is  '/home/project/nexl/start.sh' |
| ${dirs.SUBDIRS.\_parent\_.FILES.\_this\_.START\_SCRIPT} | This examples shows a complex usage of \_this\_ and \_parent\_ pointers.  Result is  '/home/project/nexl/start.sh' |

**Object operations ~**

ACTOIN\_ID is ~

ACTION\_VALUE can be one of the following K, V, O, X, Y, P

Applies for objects ( except of O which is applies for anything )

Object operations action performs different kind of data manipulations with JavaScript objects in nexl expression

|  |  |  |
| --- | --- | --- |
| ACTION\_ID  + ACTNO\_VALUE | Description | Example |
| ~K | Resolves object key set as array.  If the current value for this action is not an object action will be ignored.  In this example first nexl engine resolved a obj1 JavaScript variable and if it's a JavaScript object, resolves his key set as array | ${obj1~K} |
| ~V | Resolves all object values as array ( including sub values ).  If the current value for this action is not an object action will be ignored  In this example first nexl engine resolved a obj1 JavaScript variable and if it's a JavaScript object, resolves all his values as array | ${obj1~V} |
| ~O | Converts item to object if it's not already an object.  If the current value for this action is object action will be ignored.  In this example first nexl engine resolved a item JavaScript variable and if it's not a JavaScript object, converts it to the object in the following way  {  "item": ITEM\_VALUE  }  where ITEM\_VALUE is a value which was resolved before  undefined values are converted to empty objects | ${item~O} |
| ~X | Produces an XML from object.  If current item ( obj1 in this example ) is not an object action will be ignored.  In this example first nexl engine resolves a obj1 JavaScript variable and if it's a JavaScript object, produces an XML from that object | ${obj1~X} |
| ~Y | Produces a YAML from object.  If current item ( obj1 in this example ) is not an object action will be ignored.  In this example first nexl engine resolves a obj1 JavaScript variable and if it's a JavaScript object, produces a YAML from that object | ${obj1~Y} |
| ~P | Produces a key=values pairs from object ( property file )  If current item ( obj1 in this example ) is not an object action will be ignored.  In this example first nexl engine resolves a obj1 JavaScript variable and if it's a JavaScript object, produces a key=value pairs from that object | ${obj1~P} |

Let’s consider examples for the following object

**person** = {

**name**: **'Alex'**,

**age**: 25,

**country**: **'Canada'**

};

|  |  |
| --- | --- |
| nexl expression | Explanation and result |
| ${person~K} | Resolves key set of person object as array  The result is  [**"name"**, **"age"**, **"country"**] |
| ${person~V} | Resolves values as of person object as array  The result is  [**"Alex"**, 25, **"Canada"**] |
| ${person~O} | This action is ignored because person is already object |
| ${person.name~O} | First nexl engine resolves a person object, then resolves a name property of that object and ~O action converts last result to object  The result is  {  **"person.name"**: **"Alex"**  } |

**Object key reverse resolution <**

ACTOIN\_ID is <

ACTION\_VALUE can be a string and/or nexl expression(s)

Applies for objects

Object key reverse resolution is a way to resolve an object keys by value(s). Example

**obj1** = {

**name**: **'Alex'**,

**age**: 25,

**country**: **'Canada'**

};

The ${obj1} expression resolves a obj1 object. If we add an object key reverse resolution action to this expression ${obj1<Alex} it will resolve all object keys which are contain a 'Alex' string value ( 'Alex' is an ACTION\_VALUE ). The ${obj1<Alex} expression is evaluated to [ 'name' ].

Object key reverse resolution is always producing an array. Array can be empty if there is no matching result.

The following expression will be evaluated to empty array

${obj1<25}

It happens because 25 is treated as string and not as numeric. To provide a numeric constant value use the following internal expression ${@25:num}

The whole expression is

${obj1<${@25:num}}

And it will be evaluated to [ 'age' ] array

Additionally it's possible to provide a multiple values for object key reverse resolution action by using of array. Let's declare an array

***arr1*** = [25, **'Canada'**, **true**];

And the following expression ${obj1<${arr1}} will be evaluated to

[**'age'**, **'country'**]

**Array operations #**

ACTOIN\_ID is #

ACTION\_VALUE can be one of the following S, s, U, D, LEN, A, F

Applies for arrays ( except of A which is applies for anything )

Array operations perform different kind of data manipulations with JavaScript arrays in nexl expression

|  |  |  |
| --- | --- | --- |
| Action | Description | Example |
| #S | Sorts array in ascending order.  If the current value for this action is not an array action will be ignored.  In this example first nexl engine resolves a arr1 JavaScript variable and if it's a JavaScript array, sorts it in ascending order | ${arr1#S} |
| #s | Same as previous action, but sorts in descending order | ${arr1#s} |
| #U | Eliminates all duplicate values from array making array values unique.  If the current value for this action is not an array action will be ignored.  In this example first nexl engine resolves a arr1 JavaScript variable and if it's a JavaScript array, removes all duplicate values from that array | ${arr1#U} |
| #D | Resolves all repeated (duplicate) items in array  If the current value for this action is not an array action will be ignored.  In this example first nexl engine resolves a arr1 JavaScript variable and if it's a JavaScript array, resolves duplicate values only in this array | ${arr1#D} |
| #LEN | Calculates array length.  If the current value for this action is not an array action will be ignored.  In this example first nexl engine resolves a arr1 JavaScript variable and if it's a JavaScript array, calculates his length | ${arr1#LEN} |
| #A | Converts any item to array if it's not already an array.  If the current value for this action is an array action will be ignored.  For example if the item = 'test', the result will be a [ 'test' ] | ${item#A} |
| #F | Resolves the first array element if array has only one element, or undefined value if array is empty or has more than one element.  If the current value for this action is not an array action will be ignored.  In this example first nexl engine resolves a arr1 JavaScript variable. If it's a JavaScript array and has only one element, resolves that element. But if that array is empty or has more than one element, makes the result undefined | ${arr1#F} |

Let's consider examples for the following array

fruits = [**'Mango'**, **'Banana'**, **'Apple'**, **'Banana'**];

|  |  |
| --- | --- |
| nexl expression | Explanation and result |
| ${fruits} | Just resolves a fruits array as is  Result is  [**'Mango'**, **'Banana'**, **'Apple'**, **'Banana'**] |
| ${fruits#S} | fruits array is sorted in ascending order  Result is  [**'Apple'**, **'Banana'**, **'Banana'**, **'Mango'**] |
| ${fruits#U} | Duplicate values are removed from fruits array, all items in fruits array become unique  Result is  [**'Mango'**, **'Banana'**, **'Apple'**] |
| ${fruits#D} | Not a duplicate values are removed from fruits array  Result is  [**'Banana'**] |
| ${fruits#A} | This action is ignored because fruits is already an array |
| ${fruits#F} | fruits array has 4 elements, therefore it is evaluated to undefined value  Result is  undefined |
| ${fruits#U#s} | First eliminates duplicate values and then sorts in descending order  Result is  [**'Mango'**, **'Banana'**, **'Apple'**] |
| primitiveVar = 'test';  …  ${primitiveVar#S} | This is evaluated to 'test' string because primitiveVar is not a JavaScript array but a string. Therefore array action is ignored |
| obj1 = {  name: 'Alex',  age: 25,  country: 'Canada'  };  …  ${obj1~V#LEN} | 1. nexl engine resolves a obj1 JavaScript variable which is object 2. Applies a ~V action for that object. This action resolves all object values as array 3. Applies a #LEN action for that array which calculates his length   Result is  3 |

**Join array elements &**

ACTOIN\_ID is &

ACTION\_VALUE can be a string and/or nexl expression(s)

Applies for arrays

This action joins all array elements with ACTION\_VALUE delimiter.

Let's consider examples for the following JavaScript array and string

**fruits** = [**'Mango'**, **'Banana'**, **'Apple'**, **'Banana'**];

**delimiter** = **'\*'**;

|  |  |
| --- | --- |
| nexl expression | Explanation and result |
| ${fruits&,} | Joins elements of fruits array with comma.  Result is :  Mango,Banana,Apple,Banana |
| ${fruits&${delimiter}} | Joins elements of fruits array with value provided by ${delimiter} expression, i.e. with asterisk character  Result is :  Mango\*Banana\*Apple\*Banana |
| ${fruits&\\*} | Joins elements of fruits array with asterisk character. The asterisk character is escaped not to be treated as a mandatory value action  Result is :  Mango\*Banana\*Apple\*Banana |
| ${fruits&\n} | Joins elements of fruits array with LF  Result is :  Mango  Banana  Apple  Banana |
| ${fruits&\t} | Joins elements of fruits array with TAB  Result is :  Mango Banana Apple Banana |
| ${fruits&\n${delimiter}} | Joins elements of fruits array with LF and asterisk characters  Result is :  Mango  \*Banana  \*Apple  \*Banana |

**Add/Merge +**

ACTOIN\_ID is +

ACTION\_VALUE can be a string and/or nexl expression(s)

Applies for JavaScript arrays and objects

By using of add/merge action you can add new elements to array or merge two objects. This action is ignored for all other data types.

Adding elements to array examples

|  |  |
| --- | --- |
| nexl expression | Explanation and result |
| **fruits** = [**'Mango'**, **'Banana'**, **'Apple'**];  …  ${fruits+Annona} | First nexl engine resolves a fruits JavaScript array and then adds a 'Annona' string element to the end of fruits array  The result is  [**'Mango'**, **'Banana'**, **'Apple'**, **'Annona'**] |
| **fruits** = [**'Mango'**, **'Banana'**, **'Apple'**];  …  ${fruits+Annona+Kiwi+Orange&,} | Annona, Kiwi and Orange are added to fruits array and then all array elements are joined with comma  The result is  'Mango,Banana,Apple,Annona,Kiwi,Orange' |
| **fruits** = [**'Mango'**, **'Banana'**, **'Apple'**];  **vegetables** = [**'Tomato'**, **'Cabbage'**];  …  ${fruits+${vegetables}} | The ${vegetables} expression is evaluated to array and it is added to the end of fruits array  The result is  [**'Mango'**, **'Banana'**, **'Apple'**, **'Tomato'**, **'Cabbage'**] |
| **fruits** = [**'Mango'**, **'Banana'**, **'Apple'**];  …  ${fruits+${@79:num}} | The 79 numeric value is added to fruits array because the ${@79:num} expression is evaluated to 79 numeric  The result is  [**'Mango'**, **'Banana'**, **'Apple'**, 79] |
| **fruits** = [**'Mango'**, **'Banana'**, **'Apple'**];  …  ${@Annona#A+${fruits}} | Step by step explanation   1. ${@Annona} is evaluated to a 'Annona' string value ( see default value section ) 2. ${@Annona#A} has a #A action which converts a 'Annona' string value to a [ 'Annona' ] array 3. Adding fruits array to the end of ['Annona'] array   The result is  [**'Annona'**, **'Mango'**, **'Banana'**, **'Apple'**] |
| **fruits** = [**'Mango'**, **'Banana'**, **'Apple'**];  **vegetablesPrices** = {  **Tomato**: 15,  **Cucumber**: 8  };  …  ${fruits+${vegetablesPrices~K}} | The ${vegetablesPrices~K} expression resolves a key set of vegetablesPrices object as array which are added to the end of fruits array  The result is  [**'Mango'**, **'Banana'**, **'Apple'**, **'Tomato'**, **'Cucumber'**] |
| **greeting** = **'Hello'**;  …  ${greeting+World} | greeting is not a JavaScript array or object therefore this action is ignored  The result is 'Hello' |

Merging objects examples

|  |  |
| --- | --- |
| nexl expression | Explanation and result |
| ${obj1+${obj2}} | If obj1 and obj2 are JavaScript objects the obj2 will be merged to a obj1. Otherwise this action will be ignored.  nexl engine performs a deep merge for all sub elements of each object |
| ${obj1+test} | If obj1 is a JavaScript object this action will be ignored because 'test' is a primitive string and cannot be merged to an object |

**Eliminate -**

ACTOIN\_ID is -

ACTION\_VALUE can a primitive or array

Applies for arrays and objects

This action eliminates item(s) from JavaScript array or object.

|  |  |
| --- | --- |
| nexl expression | Explanation and result |
| **fruits** = [**'Mango'**, **'Banana'**, **'Apple'**];  …  ${fruits-Banana-Apple} | The 'Banana' and 'Apple' string items are eliminated from fruits array  Result is  [**'Mango'**] |
| **mixedArray** = [25, **'hello'**, **true**, 79];  …  ${mixedArray-${@25:num}} | The 25 numeric element is eliminated from mixedArray.  ${mixedArray-25} will not eliminate a 25 element because 25 is treated as string. Therefore we need to use an external expression which supplies us with a numeric value  Result is  [25, **'hello'**, **true**, 79] |
| **fruits** = [**'Mango'**, **'Lemon'**, **'Banana'**, **'Apple'**];  **sourFruits** = [**'Lemon'**, **'Apple'**];  …  ${fruits-${sourFruits}} | All items in sourFruits array are eliminated from fruits array  Result is  [**'Mango'**, **'Banana'**] |
| **fruits** = [**'Mango'**, **undefined**, **'Apple'**]  …  ${fruits-${}} | The ${} expression is evaluated to undefined. All undefined items are eliminated from fruits array  Result is  [**'Mango'**, **'Apple'**] |
| **fruits** = [**'Mango'**, **undefined**, **null**, **'Apple'**]  …  ${arr-${:null}} | The ${:null} expression is evaluated to null. All null items are eliminated from fruits array  Result is  [**'Mango'**, **undefined**, **'Apple'**] |
| **car** = {  **name**: **'BMW'**,  **model**: **'501'**,  **price**: 999  };  …  ${car-price} | The price property is eliminated from car object  Result is  **car** = {  **name**: **'BMW'**,  **model**: **'501'**  }; |

**String operations ^**

ACTION\_ID is ^

ACTION\_VALUE can be one of the following U, U1, LEN, T

Applies for strings

String operations action allows to perform basic manipulations with JavaScript strings.

|  |  |  |  |
| --- | --- | --- | --- |
| Action | Purpose | Example | Explanation and result |
| ^U | Upper case a string | **greeting** = **'Hello, World !'**;  …  ${greeting^U} | The ^L action converts a greeting string to uppercase letters. For all other data types this action is ignored  Result is  'HELLO, WORLD !' |
| ^U1 | Capitalize first letter | *text* = **'i can do it'**;  …  ${text^U1} | The ^U1 action capitalizes a first letter in the text string. For all other data types this action is ignored  Result is  'I can do it' |
| ^L | Lower case a string | **greeting** = **'Hello, World !'**;  …  ${greeting^L} | The ^L action converts a greeting string to lowercase letters. For all other data types this action is ignored  Result is  'hello, world !' |
| ^T | Trims a string | *text* = **' I can do it '**;  …  ${text^T} | The ^T action trims a text string. For all other data types this action is ignored  Result is  'I can do it' |
| ^LEN | Calculates string length | **greeting** = **'Hello, World !'**;  …  ${greeting^LEN} | The ^LEN action calculates a length of greeting string. For all other data types this action is ignored  Result is  14 |

**Undefined value operations !**

ACTION\_ID is !

ACTION\_VALUE can be one of the following U, E

|  |  |  |  |
| --- | --- | --- | --- |
| Action | Purpose | Example | Explanation and result |
| !E | Converts an empty strings, empty arrays and empty objects to undefined | **emptyStr** = **''**;  **emptyArr** = [];  **emptyObj** = {};  …  ${emptyStr!E}  ${emptyArr!E}  ${emptyObj!E} | For all those 3 expressions result is  undefined |
| !U | Evaluates the expression to undefined if one of sub expressions is evaluated to undefined | *text* = **'My name is ${name}'**;  …  ${text!U} | The text JavaScript variable is referencing to a string value which contains a ${name} sub expression. If the ${name} expression is evaluated to undefined value, the ${text!U} expression will be also evaluated to undefined value. |

**Mandatory value validator \***

ACTION\_ID is \*

ACTION\_VALUE can be a string and/or nexl expression(s)

Applies for for undefined values

Mandatory value validator action throws error for undefined values. You can provide a custom error message in ACTION\_VALUE. See examples.

|  |  |
| --- | --- |
| nexl expression | Explanation and result |
| **item** = **'test'**;  …  ${item\*} | In this case mandatory value validator is ignored because item JavaScript variable is not equals to undefined |
| ${someUndeclaredVar\*} | If the someUndeclaredVar JavaScript variable is not declared or has an undefined value nexl engine will throw the following error :  The [${someUndeclaredVar\*}] expression cannot be evaluated to undefined ( it has a mandatory value validator ). Probably you have to provide it as external arg or check why it calculated to undefined |
| ${text\*Please provide a text variable} | If the text JavaScript variable is not declared or has an undefined value you will get a custom error message you provided :  Please provide a text variable |
| **fruits** = [];  …  ${fruits\*} | In this case mandatory value validator action is ignored because fruits is not equals to undefined ( it’s an array ) |
| **fruits** = [];  …  ${fruits!E\*Fruits array is empty} | Explanation :   1. Resolving a fruits array ${fruits} 2. Applying a !E action. This action converts an empty arrays to undefined value. fruits array is empty, so we get an undefined value 3. Mandatory value validator throw error because it got an undefined value   Result is error with custom message :  Fruits array is empty |
| **cars** = {  ...  };  ${cars<${model}!E\*The ${model@} car model is not declared in cars object} | Explanation :   1. Resolving a cars object ${cars} 2. Performing object key reverse resolution on cars object with ${model} value as ${cars<${model}} 3. If the model is not provided or cars object doesn't contain that model we will get an empty array as a result 4. !E action converts an empty array to undefined value. If that array is empty we will get an undefined value 5. Mandatory value validator will throw custom error message if it got an undefined value   Our custom error message contains internal nexl expression :  The ${model@} car model is not declared in cars object  In case of model is evaluated to undefined value the @ default value action will apply an empty string value |
| ${root.b\*^L~O} | 1. Resolves a root object 2. Resolves a b property from that object 3. Throws error if root object doesn’t have that property 4. If that property is string makes it lower case 5. Converts the result to object |

**Accessing array elements [...]**

ACTION\_ID is [ which ends with ]

ACTION\_VALUE is located between [...] square brackets

Applies for arrays ( and strings )

Array elements can be accessed by their indexes. Array indexes can be :

1. integer number where negative integers are counts from the end
2. ^ character which points to the first element
3. $ character which points to the last element
4. nexl expression which can be evaluated to a 1), 2), 3)

You can specify indexes range. You can specify multiple indexes/ranges in a single brackets pair

Let's consider examples for the following array

**fruits** = [**'Mango'**, **'Banana'**, **'Orange'**, **'Annona'**, **'Grape'**];

|  |  |
| --- | --- |
| nexl expression | Explanation and result |
| ${fruits[0]}  ${fruits[^]} | Gets first element  'Mango' |
| ${fruits[$]} | Gets last element  'Grape' |
| ${fruits[-1]} | Gets second element from the end  'Annona' |
| ${fruits[0..1]}  ${fruits[^..1]} | Gets elements range from 0 to 1  [ 'Mango', 'Banana' ] |
| ${fruits[0..-1]}  ${fruits[^..-2]} | Gets elements range from first to third from the end  ['Mango', 'Banana', 'Orange'] |
| ${fruits[1..-1]} | Gets elements range from the second to second element from the end  ['Banana', 'Orange', 'Annona'] |
| ${fruits[-2..$]} | Gest elements range from third from the end to the last element  ['Orange', 'Annona', 'Grape'] |
| ${fruits[0..$]}  ${fruits[^..$]} | Gets elements range from the first to the last  ['Mango', 'Banana', 'Orange', 'Annona', 'Grape'] |
| ${fruits[^, 2, $]} | Gets first element, third and last elements  ['Mango', 'Orange', 'Grape'] |
| ${fruits[^, 1..2, $]} | Gets first element, range from second to third and last element  ['Mango', 'Banana', 'Orange', 'Grape'] |
| item = 3;  …  ${fruits[ ${item} ]} | Gets fourth element  'Annona' |
| **item1** = -1;  **item2** = **'$'**;  …  ${fruits[ 0, ${item1}..${item2} ]} | Gets first element, range from second element from the end to last element |
| **item** = **'text'**;  …  ${fruits[ ${item} ]} | Error message. Array index must be an integer or one of the following strings '^', '$' |
| obj = {};  ${obj[3]} | Accessing array elements action will be ignored if obj is not an array |

**Substring [...]**

ACTION\_ID is [ which ends with ]

ACTION\_VALUE is located between [...] square brackets

Applies for strings

All rules are same to array elements access.

Let's consider examples for the following text

*text* = **'Hello, World !'**;

|  |  |
| --- | --- |
| nexl expression | Explanation and result |
| ${text[ 0 ]}  ${text[ ^ ]} | Gets first character  'H' |
| ${text[ 5, 13 ]} | Get character number 6 and 14 ( count starts from zero ). This action produces array of characters  [ ',', '!' ] |
| ${text[ 5, 13 ]&} | Same to a previous example but here we are joining all array element with empty string by using of & action  ',!' |
| ${text[ ^..4, 7..-2 ]} | Cuts the 'Hello' and 'World' words from the text  [ 'Hello', 'World' ] |
| ${text[ ^..4, 7..-2 ]& } | Cuts the 'Hello' and 'World' words from the text and joins them with space  'Hello World' |

**Function call ()**

ACTION\_ID is ( which ends with )

ACTION\_VALUE is located between (...) parentheses

Applies for functions

To perform additional data manipulations you can call for JavaScript functions directly from nexl expressions (for example if you need to perform array sort with special rules). For example

${arraySpecialSort( ARGUMENTS\_LIST... )}

Where arraySpecialSort() is a JavaScript function. ARGUMENTS\_LIST can be only nexl expressions comma delimited as following ( spaces are ignored )

${arraySpecialSort( ${item1}, ${item2}, ... )}

See examples

|  |  |
| --- | --- |
| nexl expression | Explanation and result |
| **function** *arraySpecialSort*(arr1, rules) {  …  **return** result;  }  …  ${arraySpecialSort( ${obj1~K}, ${x} )} | In this example we are calling arraySpecialSort() function and passing to a function two following arguments :  ${obj1~K}  ${x} |
| ${Math.round( ${value} )} | Calling for a standard Math.round() function and passing a ${value} argument |
| ${Math.round( ${Math.PI} )} | Calling for a standard Math.round() function and passing a ${Math.PI} argument.  The result is 3 |
| ${Math.round( ${@3\.14:num} )} | Calling for a standard Math.round() function and passing a 3.14 constant value. Let's take a closer look to a ${@3\.14:num} expression.  In ${@3\.14:num} we need to escape dot not to be treated as object property resolution action.  :num casts a '3.14' string value to numeric |

**nexl**.nexlize() global function gives you ability to use a nexl API from inside your own JavaScript function. Function definition :

**nexl**.nexlize(nexlExpression, **args**)

nexlExpression can be a string, array or object which are contain internal nexl expression. nexlize() function will expand those internal expressions

args is optional object to override data elements in nexl source

Example :

**function** *callNexlAPI*(...) {

...

**var *result*** = **nexl**.nexlize( **'${obj1~V}'**, { **ENV**: **'dev'** } );

...

**return** result;

}

**System and user functions**

Functions declared in nexl.system.functions and nexl.user.functions are global and accessible directly without prefix.

nexl provides set of useful functions in nexl.system.functions object. Do not override that object. Use nexl.user.functions to extend your code.

For example :

**nexl**.**user**.**functions**.increase = **function** (num) {

**return** num++;

}

You can call a increase() function in two ways :

1. ${increase( ${item} )}
2. ${nexl.user.functions.increase( ${item} )}

List of system functions

|  |
| --- |
| replaceAll(entity, searchItem, replaceItem)   * replaces all searchItem elements in array or all searchItem characters in string with replaceItem   not(param)   * inverts boolean   makeObj(key, value)   * makes new JavaScript object. If the key is not provided the object will be empty   isContains(entity, item)   * if entity array contains item or string entity contains item, returns true, otherwise false   isEquals(entity1, entity2)  isBool(item)  isStr(item)  isNum(item)  isNull(item)  isUndefined(item)  isPrimitive(item)  isArray(item)  isObject(item)  ifContains(entity, item, thenIf, elseIf)   * if entity array contains item or string entity contains item, returns thenIf, otherwise returns elseIf   ifEquals(entity1, entity2, thenIf, elseIf)  ifBool(item, thenIf, elseIf)  ifStr(item, thenIf, elseIf)  ifNum(item, thenIf, elseIf)  ifNull(item, thenIf, elseIf)  ifUndefined(item, thenIf, elseIf)  ifPrimitive(item, thenIf, elseIf)  ifArray(item, thenIf, elseIf)  ifObject(item, thenIf, elseIf) |

System functions usage example :

${isPrimitive( ${item} )}

**Pushing value to function parameters stack |**

ACTION\_ID is |

ACTION\_VALUE - doesn't have a value

Each nexl expression has a special data stack. This data stack can be used when you call a function in nexl expression. In this case all values in that data stack are passed as function’s arguments ( stack arguments are passed prior to arguments described in nexl expression ).

You can apply this action as much as you need. Every time you push a value to the data stack the current value in actions chain is reseted ( i.e. became undefined value ).

Stack lives within nexl expression. After expression calculation stack is deleted.

Literally it allows you to pass a value calculated by actions in nexl expression to the function for further calculations

Examples

|  |  |
| --- | --- |
| nexl expression | Explanation and result |
| ${obj|isObject()} | 1. Resolves a obj object 2. Pushes it to the stack. This action resets current value to undefined 3. Resolves a isObject function. It is a system function therefore it’s available globally 4. Calls a function. This function gets a obj as argument because it was pushed to the stack   In other words this expression does same to the following expression  ${isObject( ${obj} )} |
| ${item|isPrimitive()|not()} | Resolves an item, pushes his value to a stack and then calling a isPrimitive() function. Function evaluation result is also pushed to the stack and then not() function inverts the result |
| ${item1|item2|isEquals( ${a}, ${b} )} | Resolves item1 and pushes it to stack. Resolves item2 and pushes it to stack. Then calls a isEquals() function. This function accepts 4 arguments. First two arguments are taken from the stack and the followed arguments are taken from explicit function parameters ${a} and ${b} |