

Expression language methods

You can process values extracted from user utterances that you want to reference in a context variable, condition, or elsewhere in the response.

Where to use the expression syntax

To expand variable values inside other variables, or apply methods to output text or context variables, use the `<? expression ?>` expression syntax. For example:

- Referencing a user's input from a dialog node text response

```
You said <? input.text ?>.
```

- Incrementing a numeric property from the JSON editor

```
"output":{"number":"<? output.number + 1 ?>"}
```

- Checking for a specific entity value from a dialog node condition

```
@city.toLowerCase() == 'paris'
```

- Checking for a specific date range from a dialog node response condition

```
@sys-date.after(today())
```

- Adding an element to a context variable array from the context editor

| Context variable name | Context variable value |
|-----------------------|---|
| toppings | <? context.toppings.append('onions') ?> |

You can use SpEL expressions in dialog node conditions and dialog node response conditions also.



Important: When a SpEL expression is used in a node condition, the surrounding `<? ?>` syntax is not required.

The following sections describe methods you can use to process values. They are organized by data type:

- [Arrays](#)
- [Date and Time](#)
- [Numbers](#)
- [Objects](#)
- [Strings](#)

Arrays

You cannot use these methods to check for a value in an array in a node condition or response condition within the same node in which you set the array values.

JSONArray.addAll(JSONArray)

This method appends one array to another.

For this dialog runtime context:

```
{
  "context": {
    "toppings_array": ["onion", "olives"],
    "more_toppings": ["mushroom","pepperoni"]
  }
}
```

Make this update:

```
{
  "context": {
```

```
    "toppings_array": "<? $toppings_array.addAll($more_toppings) ?>"
  }
}
```

Result: The method itself returns `null`. However, the first array is updated to include the values from the second array.

```
{
  "context": {
    "toppings_array": ["onion", "olives", "mushroom", "pepperoni"]
  }
}
```

JSONArray.append(object)

This method appends a new value to the JSONArray and returns the modified JSONArray.

For this Dialog runtime context:

```
{
  "context": {
    "toppings_array": ["onion", "olives"]
  }
}
```

Make this update:

```
{
  "context": {
    "toppings_array": "<? $toppings_array.append('ketchup', 'tomatoes') ?>"
  }
}
```

Result:

```
{
  "context": {
    "toppings_array": ["onion", "olives", "ketchup", "tomatoes"]
  }
}
```

JSONArray.clear()

This method clears all values from the array and returns null.

Use the following expression in the output to define a field that clears an array that you saved to a context variable (\$toppings_array) of its values.

```
{
  "output": {
    "array_eraser": "<? $toppings_array.clear() ?>"
  }
}
```

If you subsequently reference the \$toppings_array context variable, it returns '[]' only.

JSONArray.contains(Object value)

This method returns true if the input JSONArray contains the input value.

For this Dialog runtime context which is set by a previous node or external application:

```
{
  "context": {
    "toppings_array": ["onion", "olives", "ham"]
  }
}
```

Dialog node or response condition:

```
$toppings_array.contains('ham')
```

Result: `true` because the array contains the element `ham`.

JSONArray.containsIgnoreCase(Object value)

This method returns `true` if the input JSONArray contains the input value, regardless of whether the value is specified in uppercase or lowercase letters.

For this Dialog runtime context which is set by a previous node or external application:

```
{
  "context": {
    "toppings_array": ["onion", "olives", "ham"]
  }
}
```

Dialog node or response condition:

```
$toppings_array.containsIgnoreCase( 'HAM' )
```

Result: `true` because the array contains the element `ham` and the case is ignored.

JSONArray.containsIntent(String intent_name, Double min_score, [Integer top_n])

This method returns `true` if the `intents` JSONArray specifically contains the specified intent, and that intent has a confidence score that is equal to or higher than the specified minimum score. Optionally, you can specify a number to indicate that the intent must be included within that number of top elements in the array. The `top_n` parameter is ignored if you specify a negative number.

Returns `false` if the specified intent is not in the array, does not have a confidence score that is equal to or greater than the minimum confidence score, or the array index of the intent is lower than the specified index location.

The service automatically generates an `intents` array that lists the intents that the service detects in the input whenever user input is submitted. The array lists all intents that are detected by the service in order of highest confidence first.

You can use this method in a node condition to not only check for the presence of an intent, but to set a confidence score threshold that must be met before the node can be processed and its response returned.

For example, use the following expression in a node condition when you want to trigger the dialog node only when the following conditions are met:

- The `#General_Ending` intent is present.
- The confidence score of the `#General_Ending` intent is over 80%.
- The `#General_Ending` intent is one of the top 2 intents in the intents array.

```
intents.containsIntent("General_Ending", 0.8, 2)
```

JSONArray.filter(temp, "temp.property operator comparison_value")

Filters an array by comparing each array element value to a value you specify. This method is similar to a [collection projection](#). A collection projection returns a filtered array based on a name in an array element name-value pair. The filter method returns a filtered array based on a value in an array element name-value pair.

The filter expression consists of the following values:

- `temp`: Name of a variable that is used temporarily as each array element is evaluated. For example, `city`.
- `property`: Element property that you want to compare to the `comparison_value`. Specify the property as a property of the temporary variable that you name in the first parameter. Use the syntax: `temp.property`. For example, if `latitude` is a valid element name for a name-value pair in the array, specify the property as `city.latitude`.
- `operator`: Operator to use to compare the property value to the `comparison_value`.

Supported operators are:

| Operator | Description |
|----------|-----------------|
| == | Is equal to |
| > | Is greater than |
| < | Is less than |

| | |
|--------------------|-----------------------------|
| <code>>=</code> | Is greater than or equal to |
| <code><=</code> | Is less than or equal to |
| <code>!=</code> | Is not equal to |

Supported filter operators

- `comparison_value`: Value that you want to compare each array element property value against. To specify a value that can change depending on the user input, use a context variable or entity as the value. If you specify a value that can vary, add logic to guarantee that the `comparison_value` value is valid at evaluation time or an error will occur.

Filter example 1

For example, you can use the filter method to evaluate an array that contains a set of city names and their population numbers to return a smaller array that contains only cities with a population over 5 million.

The following `$cities` context variable contains an array of objects. Each object contains a `name` and `population` property.

```
[
  {
    "name": "Tokyo",
    "population": 9273000
  },
  {
    "name": "Rome",
    "population": 2868104
  },
  {
    "name": "Beijing",
    "population": 20693000
  },
  {
    "name": "Paris",
    "population": 2241346
  }
]
```

In the following example, the arbitrary temporary variable name is `city`. The SpEL expression filters the `$cities` array to include only cities with a population of over 5 million:

```
$cities.filter("city", "city.population > 5000000")
```

The expression returns the following filtered array:

```
[
  {
    "name": "Tokyo",
    "population": 9273000
  },
  {
    "name": "Beijing",
    "population": 20693000
  }
]
```

You can use a collection projection to create a new array that includes only the city names from the array returned by the filter method. You can then use the `join` method to display the two name element values from the array as a String, and separate the values with a comma and a space.

```
The cities with more than 5 million people include <? T(String).join(", ", ($cities.filter("city", "city.population > 5000000")).![name]) ?>.
```

The resulting response is: `The cities with more than 5 million people include Tokyo, Beijing.`

Filter example 2

The power of the filter method is that you do not need to hard code the `comparison_value` value. In this example, the hard coded value of 5000000 is replaced with a context variable instead.


In this example, the `$population_min` context variable contains the number `5000000`. The arbitrary temporary variable name is `city`. The SpEL

expression filters the `$cities` array to include only cities with a population of over 5 million:

```
$cities.filter("city", "city.population > $population_min")
```

The expression returns the following filtered array:

```
[
  {
    "name": "Tokyo",
    "population": 9273000
  },
  {
    "name": "Beijing",
    "population": 20693000
  }
]
```

 **Tip:** When comparing number values, be sure to set the context variable involved in the comparison to a valid value before the filter method is triggered. Note that `null` can be a valid value if the array element you are comparing it against might contain it. For example, if the population name and value pair for Tokyo is `"population": null`, and the comparison expression is `"city.population == $population_min"`, then `null` would be a valid value for the `$population_min` context variable.

You can use a dialog node response expression like this:

```
The cities with more than $population_min people include <? T(String).join(", ", ($cities.filter("city", "city.population > $population_min")).![name]) ?>.
```

The resulting response is: `The cities with more than 5000000 people include Tokyo, Beijing.`

Filter example 3

In this example, an entity name is used as the `comparison_value`. The user input is, `What is the population of Tokyo?` The arbitrary temporary variable name is `y`. You created an entity named `@city` that recognizes city names, including `Tokyo`.

```
$ $cities.filter("y", "y.name == @city")
```

The expression returns the following array:

```
[
  {
    "name": "Tokyo",
    "population": 9273000
  }
]
```

You can use a collection project to get an array with only the population element from the original array, and then use the `get` method to return the value of the population element.

```
The population of @city is: <? ($cities.filter("y", "y.name == @city")).![population]).get(0) ?>.
```

The expression returns: `The population of Tokyo is 9273000.`

JSONArray.get(Integer)

This method returns the input index from the JSONArray.

For this Dialog runtime context which is set by a previous node or external application:

```
{
  "context": {
    "name": "John",
    "nested": {
      "array": [ "one", "two" ]
    }
  }
}
```

Dialog node or response condition:

```
$nested.array.get(0).getAsString().contains('one')
```

Result: **True** because the nested array contains **one** as a value.

Response:

```
"output": {
  "generic" : [
    {
      "values": [
        {
          "text" : "The first item in the array is <?$nested.array.get(0)?>"
        }
      ],
      "response_type": "text",
      "selection_policy": "sequential"
    }
  ]
}
```

JSONArray.getRandomItem()

This method returns a random item from the input JSONArray.

For this Dialog runtime context:

```
{
  "context": {
    "toppings_array": ["onion", "olives", "ham"]
  }
}
```

Dialog node output:

```
{
  "output": {
    "generic" : [
      {
        "values": [
          {
            "text": "<?$toppings_array.getRandomItem() ?> is a great choice!"
          }
        ],
        "response_type": "text",
        "selection_policy": "sequential"
      }
    ]
  }
}
```

Result: **"ham is a great choice!"** or **"onion is a great choice!"** or **"olives is a great choice!"**

Note: The resulting output text is randomly chosen.

JSONArray.indexOf(value)

This method returns the index number of the element in the array that matches the value you specify as a parameter or **-1** if the value is not found in the array. The value can be a String (**"School"**), Integer(**8**), or Double (**9.1**). The value must be an exact match and is case sensitive.

For example, the following context variables contain arrays:

```
$ {
  "context": {
    "array1": ["Mary","Lamb","School"],
    "array2": [8,9,10],
    "array3": [8.1,9.1,10.1]
  }
}
```

The following expressions can be used to determine the array index at which the value is specified:

```
$ <? $array1.indexOf("Mary") ?> returns `0`
```

```
<? $array2.index0f(9) ?> returns `1`  
<? $array3.index0f(10.1) ?> returns `2`
```

This method can be useful for getting the index of an element in an intents array, for example. You can apply the `index0f` method to the array of intents generated each time user input is evaluated to determine the array index number of a specific intent.

```
intents.index0f("General_Greetings")
```

If you want to know the confidence score for a specific intent, you can pass the earlier expression in as the `index` value to an expression with the syntax `intents[index].confidence`. For example:

```
intents[intents.index0f("General_Greetings")].confidence
```

JSONArray.join(String delimiter)

This method joins all values in this array to a string. Values are converted to string and delimited by the input delimiter.

For this Dialog runtime context:

```
{  
  "context": {  
    "toppings_array": ["onion", "olives", "ham"]  
  }  
}
```

Dialog node output:

```
{  
  "output": {  
    "generic" : [  
      {  
        "values": [  
          {  
            "text": "This is the array: <? $toppings_array.join(';') ?>"  
          }  
        ],  
        "response_type": "text",  
        "selection_policy": "sequential"  
      }  
    ]  
  }  
}
```

Result:

```
This is the array: onion;olives;ham;
```

If a user input mentions multiple toppings, and you defined an entity named `@toppings` that can recognize topping mentions, you could use the following expression in the response to list the toppings that were mentioned:

```
So, you'd like <? @toppings.values.join(',') ?>.
```

If you define a variable that stores multiple values in a JSON array, you can return a subset of values from the array, and then use the `join()` method to format them properly.

Collection projection

A `collection projection` SpEL expression extracts a subcollection from an array that contains objects. The syntax for a collection projection is `array_that_contains_value_sets.[value_of_interest]`.

For example, the following context variable defines a JSON array that stores flight information. There are two data points per flight, the time and flight code.

```
"flights_found": [  
  {  
    "time": "10:00",  
    "flight_code": "OK123"  
  },  
  {  
    "time": "12:30",  
    "flight_code": "LH421"  
  }  
]
```

```
},
{
  "time": "16:15",
  "flight_code": "TS4156"
}
]
```

To return the flight codes only, you can create a collection projection expression by using the following syntax:

```
$ <? $flights_found.![flight_code] ?>
```

This expression returns an array of the `flight_code` values as `["OK123","LH421","TS4156"]`. See the [SpEL Collection projection documentation](#) for more details.

If you apply the `join()` method to the values in the returned array, the flight codes are displayed in a comma-separated list. For example, you can use the following syntax in a response:

```
The flights that fit your criteria are:
<? T(String).join(",", $flights_found.![flight_code]) ?>.
```

Result: `The flights that match your criteria are: OK123,LH421,TS4156.`

JSONArray.joinToArray(template, retainDataType)

This method extracts information from each item in the array and builds a new array that is formatted according to the template you specify. The template can be a string, a JSON object, or an array. The method returns an array of strings, an array of objects, or an array of arrays, depending on the type of template.

This method is useful for formatting information as a string you can return as part of the output of a dialog node, or for transforming data into a different structure so you can use it with an external API.

In the template, you can reference values from the source array using the following syntax:

```
$ %e.{property}%
```

where `{property}` represents the name of the property in the source array.

For example, suppose your assistant has stored an array containing flight details in a context variable. The stored data might look like this:

```
"flights": [
  {
    "flight": "AZ1040",
    "origin": "JFK",
    "carrier": "Alitalia",
    "duration": 485,
    "destination": "FCO",
    "arrival_date": "2019-02-03",
    "arrival_time": "07:00",
    "departure_date": "2019-02-02",
    "departure_time": "16:45"
  },
  {
    "flight": "DL1710",
    "origin": "JFK",
    "carrier": "Delta",
    "duration": 379,
    "destination": "LAX",
    "arrival_date": "2019-02-02",
    "arrival_time": "10:19",
    "departure_date": "2019-02-02",
    "departure_time": "07:00"
  },
  {
    "flight": "VS4379",
    "origin": "BOS",
    "carrier": "Virgin Atlantic",
    "duration": 385,
    "destination": "LHR",
    "arrival_date": "2019-02-03",
    "arrival_time": "09:05",
    "departure_date": "2019-02-02",
    "departure_time": "21:40"
  }
]
```



```
]
```

To build an array of strings that describe these flights in a user-readable form, you might use the following expression:

```
$ ${Flight_data}.joinToArray("Flight %e.flight% to %e.destination%", true)
```

This expression would return the following array of strings: `["Flight AZ1040 to FCO","Flight DL1710 to LAX","Flight VS4379 to LHR"]`.

The optional `retainDataType` parameter specifies whether the method should preserve the data type of all input values in the returned array. If `retainDataType` is set to `false` or omitted, in some situations, strings in the input array might be converted to numbers in the returned array. For example, if the selected values from the input array are `"1"`, `"2"`, and `"3"`, the returned array might be `[1, 2, 3]`. To avoid unexpected type conversions, specify `true` for this parameter.

Complex templates

A more complex template might contain formatting that displays the information in a legible layout. For a complex template, you might want to store the template in a context variable, which you can then pass to the `joinToArray` method instead of a string.

For example, this complex template contains a subset of the array elements, adding labels and formatting:

```
<br/>Flight number: %e.flight% <br/> Airline: %e.carrier% <br/> Departure date: %e.departure_date% <br/>
Departure time: %e.departure_time% <br/> Arrival time: %e.arrival_time% <br/>
```



Note: Make sure any formatting you use in your template is supported by the channel integration that will be displaying the assistant output.

If you create a context variable called `Template`, and assign this template as its value, you can then use that variable in your expressions:

```
$ ${Flight_data}.joinToArray(${Template})
```

At run time, the response would look like this:

```
Flight number: AZ1040
Airline: Alitalia
Departure date: 2019-02-02
Departure time: 16:45
Arrival time: 07:00

Flight number: DL1710
Airline: Delta
Departure date: 2019-02-02
Departure time: 07:00
Arrival time: 10:19

Flight number: VS4379
Airline: Virgin Atlantic
Departure date: 2019-02-02
Departure time: 21:40
Arrival time: 09:05
```

JSON Object templates

Instead of a string, you can define a template as a JSON object. This provides a way to standardize the formatting of information from different systems, or to transform data into the format required for an external service.

In this example, a template is defined as a JSON object that extracts flight details from the elements specified in the array stored in the `Flight data` context variable:

```
{
  "departure": "Flight %e.flight% departs on %e.departure_date% at %e.departure_time%. ",
  "arrival": "Flight %e.flight% arrives on %e.arrival_date% at %e.arrival_time%."
}
```

Using this template, the `joinToArray()` method returns a new array of objects with the specified structure.

JSONArray.remove(Integer)

This method removes the element in the index position from the JSONArray and returns the updated JSONArray.

For this Dialog runtime context:

```
{
  "context": {
    "toppings_array": ["onion", "olives"]
  }
}
```

Make this update:

```
{
  "context": {
    "toppings_array": "<? $toppings_array.remove(0) ?>"
  }
}
```

Result:

```
{
  "context": {
    "toppings_array": ["olives"]
  }
}
```

JSONArray.removeValue(object)

This method removes the first occurrence of the value from the JSONArray and returns the updated JSONArray.

For this Dialog runtime context:

```
{
  "context": {
    "toppings_array": ["onion", "olives"]
  }
}
```

Make this update:

```
{
  "context": {
    "toppings_array": "<? $toppings_array.removeValue('onion') ?>"
  }
}
```

Result:

```
{
  "context": {
    "toppings_array": ["olives"]
  }
}
```

JSONArray.set(Integer index, Object value)

This method sets the input index of the JSONArray to the input value and returns the modified JSONArray.

For this Dialog runtime context:

```
{
  "context": {
    "toppings_array": ["onion", "olives", "ham"]
  }
}
```

Dialog node output:

```
{
  "context": {
    "toppings_array": "<? $toppings_array.set(1,'ketchup')?>"
  }
}
```

Result:

```
{
  "context": {
    "toppings_array": ["onion", "ketchup", "ham"]
  }
}
```

JSONArray.size()

This method returns the size of the JSONArray as an integer.

For this Dialog runtime context:

```
{
  "context": {
    "toppings_array": ["onion", "olives"]
  }
}
```

Make this update:

```
{
  "context": {
    "toppings_array_size": "<? $toppings_array.size() ?>"
  }
}
```

Result:

```
{
  "context": {
    "toppings_array_size": 2
  }
}
```

JSONArray.split(String regexp)

This method splits the input string by using the input regular expression. The result is a JSONArray of strings.

For this input:

```
"bananas;apples;pears"
```

This syntax:

```
{
  "context": {
    "array": "<?input.text.split(";")?>"
  }
}
```

Results in this output:

```
{
  "context": {
    "array": [ "bananas", "apples", "pears" ]
  }
}
```

com.google.gson.JsonArray support

In addition to the built-in methods, you can use standard methods of the `com.google.gson.JsonArray` class.

New array

```
new JsonArray().append('value')
```

To define a new array that will be filled in with values that are provided by users, you can instantiate an array. You must also add a placeholder value to the

array when you instantiate it. You can use the following syntax to do so:

```
{
  "context":{
    "answer": "<? output.answer?:new JSONArray().append('temp_value') ?>"
  }
}
```

Date and Time

Several methods are available to work with date and time.

For information about how to recognize and extract date and time information from user input, see [@sys-date and @sys-time entities](#).

The following string formats are supported for date-time literals on which the methods below may be invoked.

- For time only: `HH:mm:ss` or `HH:mm`
- For date only: `yyyy-MM-dd`
- For date and time: `yyyy-MM-dd HH:mm:ss`
- For date and time with time zone: `yyyy-MM-dd HH:mm:ss VV`. The V symbol is from the [DateTimeFormatter](#) and represents time zone in IANA Time Zone Database (TZDB) format, for example, Europe/London.

.after(String date or time)

Determines whether the date/time value is after the date/time argument.

.before(String date or time)

Determines whether the date/time value is before the date/time argument.

For example:

- `@sys-time.before('12:00:00')`
- `@sys-date.before('2016-11-21')`
- If comparing different items, such as `time vs. date`, `date vs. time`, and `time vs. date and time`, the method returns false and an exception is printed in the response JSON log `output.log_messages`.

For example, `@sys-date.before(@sys-time)`.

- If comparing `date and time vs. time` the method ignores the date and only compares times.

now(String time zone)

Returns a string with the current date and time in the format `yyyy-MM-dd HH:mm:ss`. Optionally specify a `timezone` value to get the current date and time for a specific time zone, with a returned string in the format `yyyy-MM-dd HH:mm:ss 'GMT'XXX`.

- Static function.
- The other date/time methods can be invoked on date-time values that are returned by this function and it can be passed in as their argument.
- The user interface creates a `$timezone` context variable for you automatically so the correct time is returned when you test from the "Try it out" pane. If you don't pass a time zone, the time zone that is set automatically by the UI is used. Outside of the UI, `GMT` is used as the time zone. To learn about the syntax to use to specify the time zone, see [Time zones supported by system entities](#).

Example of `now()` being used to first check whether it's morning before responding with a morning-specific greeting.

```
{
  "conditions": "now().before('12:00:00')",
  "output": {
    "generic": [
      {
        "values": [
          {
            "text": "Good morning!"
          }
        ],
        "response_type": "text",
        "selection_policy": "sequential"
      }
    ]
  }
}
```

Example of using now() with a timezone to return the current time (in England):

```
{
  "output": {
    "generic": [
      {
        "values": [
          {
            "text": "The current date and time is: <? now('Europe/London') ?>"
          }
        ],
        "response_type": "text",
        "selection_policy": "sequential"
      }
    ]
  }
}
```

You can substitute the hard-coded time zone value with a context variable to dynamically change the time based on a time zone that is passed to the expression. For example: `<? now('$myzone') ?>`. The `$myzone` context variable might be set to `'Australia/Sydney'` in one conversation and to `'Mexico/BajaNorte'` in another.

.reformatDateTime(String format)

Formats date and time strings to the format desired for user output.

Returns a formatted string according to the specified format:

- `MM/dd/yyyy` for 12/31/2016
- `h a` for 10pm

To return the day of the week:

- `EEEE` for Tuesday
- `E` for Tue
- `u` for day index (1 = Monday, ..., 7 = Sunday)

For example, this context variable definition creates a \$time variable that saves the value 17:30:00 as `5:30 PM`.

```
{
  "context": {
    "time": "<? @sys-time.reformatDateTime('h:mm a') ?>"
  }
}
```

Format follows the Java [SimpleDateFormat](#) rules.

Note: When trying to format time only, the date is treated as `1970-01-01`.

.sameMoment(String date/time)

- Determines whether the date/time value is the same as the date/time argument.

.sameOrAfter(String date/time)

- Determines whether the date/time value is after or the same as the date/time argument.
- Analogous to `.after()`.

.sameOrBefore(String date/time)

- Determines whether the date/time value is before or the same as the date/time argument.

today()

Returns a string with the current date in the format `yyyy-MM-dd`.

- Static function.
- The other date methods can be invoked on date values that are returned by this function and it can be passed in as their argument.
- If the context variable `$timezone` is set, this function returns dates in the client's time zone. Otherwise, the `GMT` time zone is used.

Example of a dialog node with `today()` used in the output field:

```
{
  "conditions": "#what_day_is_it",
  "output": {
    "generic": [
      {
        "values": [
          {
            "text": "Today's date is <? today() ?>."
          }
        ],
        "response_type": "text",
        "selection_policy": "sequential"
      }
    ]
  }
}
```

Result: `Today's date is 2018-03-09.`

Date and time calculations

Use the following methods to calculate a date.

| Method | Description |
|--|---|
| <code><date>.minusDays(n)</code> | Returns the date of the day n number of days before the specified date. |
| <code><date>.minusMonths(n)</code> | Returns the date of the day n number of months before the specified date. |
| <code><date>.minusYears(n)</code> | Returns the date of the day n number of years before the specified date. |
| <code><date>.plusDays(n)</code> | Returns the date of the day n number of days after the specified date. |
| <code><date>.plusMonths(n)</code> | Returns the date of the day n number of months after the specified date. |
| <code><date>.plusYears(n)</code> | Returns the date of the day n number of years after the specified date. |

where `<date>` is specified in the format `yyyy-MM-dd` or `yyyy-MM-dd HH:mm:ss`.

To get tomorrow's date, specify the following expression:

```
{
  "output": {
    "generic": [
      {
        "values": [
          {
            "text": "Tomorrow's date is <? today().plusDays(1) ?>."
          }
        ],
        "response_type": "text",
        "selection_policy": "sequential"
      }
    ]
  }
}
```

Result if today is March 9, 2018: `Tomorrow's date is 2018-03-10.`

To get the date for the day a week from today, specify the following expression:

```
{
  "output": {
    "generic": [
      {
        "values": [
          {
            "text": "Next week's date is <? @sys-date.plusDays(7) ?>."
          }
        ]
      }
    ]
  }
}
```

```
    ],
    "response_type": "text",
    "selection_policy": "sequential"
  }
]
}
```

Result if the date captured by the @sys-date entity is today's date, March 9, 2018: **Next week's date is 2018-03-16.**

To get last month's date, specify the following expression:

```
{
  "output": {
    "generic": [
      {
        "values": [
          {
            "text": "Last month the date was <? today().minusMonths(1) ?>."
          }
        ],
        "response_type": "text",
        "selection_policy": "sequential"
      }
    ]
  }
}
```

Result if today is March 9, 2018: **Last month the date was 2018-02-9.**

Use the following methods to calculate time.

| Method | Description |
|---|---|
| <code><time>.minusHours(n)</code> | Returns the time n hours before the specified time. |
| <code><time>.minusMinutes(n)</code> | Returns the time n minutes before the specified time. |
| <code><time>.minusSeconds(n)</code> | Returns the time n seconds before the specified time. |
| <code><time>.plusHours(n)</code> | Returns the time n hours after the specified time. |
| <code><time>.plusMinutes(n)</code> | Returns the time n minutes after the specified time. |
| <code><time>.plusSeconds(n)</code> | Returns the time n secons after the specified time. |

where `<time>` is specified in the format `HH:mm:ss`.

To get the time an hour from now, specify the following expression:

```
{
  "output": {
    "generic": [
      {
        "values": [
          {
            "text": "One hour from now is <? now().plusHours(1) ?>."
          }
        ],
        "response_type": "text",
        "selection_policy": "sequential"
      }
    ]
  }
}
```

Result if it is 8 AM: **One hour from now is 09:00:00.**

To get the time 30 minutes ago, specify the following expression:

```
{
  "output": {
    "generic": [
      {
        "values": [
          {
            "text": "A half hour before @sys-time is <? @sys-time.minusMinutes(30) ?>."
          }
        ],
        "response_type": "text",
        "selection_policy": "sequential"
      }
    ]
  }
}
```

Result if the time captured by the @sys-time entity is 8 AM: **A half hour before 08:00:00 is 07:30:00.**

To reformat the time that is returned, you can use the following expression:

```
{
  "output": {
    "generic": [
      {
        "values": [
          {
            "text": "6 hours ago was <? now().minusHours(6).reformatDateTime('h:mm a') ?>."
          }
        ],
        "response_type": "text",
        "selection_policy": "sequential"
      }
    ]
  }
}
```

Result if it is 2:19 PM: **6 hours ago was 8:19 AM.**

Working with time spans

To show a response based on whether today's date falls within a certain time frame, you can use a combination of time-related methods. For example, if you run a special offer during the holiday season every year, you can check whether today's date falls between November 25 and December 24 of this year. First, define the dates of interest as context variables.

In the following start and end date context variable expressions, the date is being constructed by concatenating the dynamically-derived current year value with hard-coded month and day values.

```
$ "context": {
  "end_date": "<? now().reformatDateTime('Y') + '-12-24' ?>",
  "start_date": "<? now().reformatDateTime('Y') + '-11-25' ?>"
}
```

In the response condition, you can indicate that you want to show the response only if the current date falls between the start and end dates that you defined as context variables.

```
now().after($start_date) && now().before($end_date)
```

java.util.Date support

In addition to the built-in methods, you can use standard methods of the `java.util.Date` class.

To get the date of the day that falls a week from today, you can use the following syntax.

```
{
  "context": {
    "week_from_today": "<? new Date(new Date().getTime() +
      (7 * (24*60*60*1000L))) ?>"
  }
}
```

This expression first gets the current date in milliseconds (since January 1, 1970, 00:00:00 GMT). It also calculates the number of milliseconds in 7 days. (The `(24*60*60*1000L)` represents one day in milliseconds.) It then adds 7 days to the current date. The result is the full date of the day that falls a

week from today. For example, `Fri Jan 26 16:30:37 UTC 2018`. Note that the time is in the UTC time zone. You can always change the 7 to a variable (`$number_of_days` , for example) that you can pass in. Just be sure that its value gets set before this expression is evaluated.

If you want to be able to subsequently compare the date with another date that is generated by the service, then you must reformat the date. System entities (`@sys-date`) and other built-in methods (`now()`) convert dates to the `yyyy-MM-dd` format.

```
{
  "context": {
    "week_from_today": "<? new Date(new Date().getTime() +
      (7 * (24*60*60*1000L))).format('yyyy-MM-dd') ?>"
  }
}
```

After reformatting the date, the result is `2018-01-26`. Now, you can use an expression like `@sys-date.after($week_from_today)` in a response condition to compare a date specified in user input to the date saved in the context variable.

The following expression calculates the time 3 hours from now.

```
{
  "context": {
    "future_time": "<? new Date(new Date().getTime() + (3 * (60*60*1000L)) -
      (5 * (60*60*1000L))).format('h:mm a') ?>"
  }
}
```

The `(60*60*1000L)` value represents an hour in milliseconds. This expression adds 3 hours to the current time. It then recalculates the time from a UTC time zone to EST time zone by subtracting 5 hours from it. It also reformats the date values to include hours and minutes AM or PM.

Numbers

These methods help you get and reformat number values.

For information about system entities that can recognize and extract numbers from user input, see [@sys-number entity](#).

If you want the service to recognize specific number formats in user input, such as order number references, consider creating a pattern entity to capture it. See [Creating entities](#) for more details.

If you want to change the decimal placement for a number, to reformat a number as a currency value, for example, see the [String format\(\) method](#).

toDouble()

```
$ Converts the object or field to the Double number type. You can call this method on any object or field. If the conversion fails, *null* is returned.
```

toInt()

```
$ Converts the object or field to the Integer number type. You can call this method on any object or field. If the conversion fails, *null* is returned.
```

toLong()

```
$ Converts the object or field to the Long number type. You can call this method on any object or field. If the conversion fails, *null* is returned.

If you specify a Long number type in a SpEL expression, you must append an `L` to the number to identify it as such. For example, `5000000000L`. This syntax is required for any numbers that do not fit into the 32-bit Integer type. For example, numbers that are greater than 2^31 (2,147,483,648) or lower than -2^31 (-2,147,483,648) are considered Long number types. Long number types have a minimum value of -2^63 and a maximum value of 2^63-1 (or 9,223,372,036,854,775,807).

If you need to find out if a number is too long to be recognized properly in the dialog, you can check whether there are more than 18 integers in the number by using an expression like this:

``` {<codeblock>
```

18 ?>

```
$ If you need to work with numbers that are longer than 18 integers, consider using a pattern entity (with a regular expression such as `\\d{20}`) to work with them instead of using `@sys-number`.
```

### Standard math {: #dialog-methods-numbers-standard-math}

Use SpEL expressions to define standard math equations, where the operators are represented by using these symbols:

| Arithmetic operation | Symbol |
|----------------------|--------|
| addition             | +      |
| division             | /      |
| multiplication       | *      |
| subtraction          | -      |

For example, in a dialog node response, you might add a context variable that captures a number specified in the user input (`@sys-number`), and saves it as `$your_number`. You can then add the following text as a text response:

```
``` {: codeblock}
I'm doing math. Given the value you specified ($your_number), when I add 5, I get: <? $your_number + 5 ?>.
When I subtract 5, I get: <? $your_number - 5 ?>.
When I multiply it by 5, I get: <? $your_number * 5 ?>.
When I divide it by 5, I get: <? $your_number/5 ?>.
```

If the user specifies `10`, then the resulting text response looks like this:

```
I'm doing math. Given the value you specified (10), when I add 5, I get: 15.
When I subtract 5, I get: 5.
When I multiply it by 5, I get: 50.
When I divide it by 5, I get: 2.
```

Java number support

java.lang.Math()

Performs basic numeric operations.

You can use the the Class methods, including these:

- max()

```
{
  "context": {
    "bigger_number": "<? T(Math).max($number1,$number2) ?>"
  },
  "output": {
    "generic": [
      {
        "values": [
          {
            "text": "The bigger number is $bigger_number."
          }
        ],
        "response_type": "text",
        "selection_policy": "sequential"
      }
    ]
  }
}
```

- min()

```
{
  "context": {
    "smaller_number": "<? T(Math).min($number1,$number2) ?>"
  },
  "output": {
    "generic": [
      {
        "values": [
          {
            "text": "The smaller number is $smaller_number."
          }
        ],
        "response_type": "text",
        "selection_policy": "sequential"
      }
    ]
  }
}
```

```
}  
}
```

- `pow()`

```
{  
  "context": {  
    "power_of_two": "<? T(Math).pow($base.toDouble(),2.toDouble()) ?>"  
  },  
  "output": {  
    "generic": [  
      {  
        "values": [  
          {  
            "text": "Your number $base to the second power is $power_of_two."  
          }  
        ],  
        "response_type": "text",  
        "selection_policy": "sequential"  
      }  
    ]  
  }  
}
```

See the [java.lang.Math reference documentation](#) for information about other methods.

java.util.Random()

Returns a random number. You can use one of the following syntax options:

- To return a random boolean value (true or false), use `<?new Random().nextBoolean()?>`.
- To return a random double number between 0 (included) and 1 (excluded), use `<?new Random().nextDouble()?>`
- To return a random integer between 0 (included) and a number you specify, use `<?new Random().nextInt(n)?>` where n is the top of the number range you want + 1. For example, if you want to return a random number between 0 and 10, specify `<?new Random().nextInt(11)?>`.
- To return a random integer from the full Integer value range (-2147483648 to 2147483648), use `<?new Random().nextInt()?>`.

For example, you might create a dialog node that is triggered by the #random_number intent. The first response condition might look like this:

```
Condition = @sys-number  
{  
  "context": {  
    "answer": "<? new Random().nextInt(@sys-number.numeric_value + 1) ?>"  
  },  
  "output": {  
    "generic": [  
      {  
        "values": [  
          {  
            "text": "Here's a random number between 0 and @sys-number.literal: $answer."  
          }  
        ],  
        "response_type": "text",  
        "selection_policy": "sequential"  
      }  
    ]  
  }  
}
```

See the [java.util.Random reference documentation](#) for information about other methods.

You can use standard methods of the following classes also:

- `java.lang.Byte`
- `java.lang.Integer`
- `java.lang.Long`
- `java.lang.Double`
- `java.lang.Short`
- `java.lang.Float`

Objects

JSONObject.clear()

This method clears all values from the JSON object and returns null.

For example, you want to clear the current values from the \$user context variable.

```
{
  "context": {
    "user": {
      "first_name": "John",
      "last_name": "Snow"
    }
  }
}
```

Use the following expression in the output to define a field that clears the object of its values.

```
{
  "output": {
    "object_eraser": "<? $user.clear() ?>"
  }
}
```

If you subsequently reference the \$user context variable, it returns `{}` only.

You can use the `clear()` method on the `context` or `output` JSON objects in the body of the API `/message` call.

Clearing context

When you use the `clear()` method to clear the `context` object, it clears **all** variables except these ones:

- `context.conversation_id`
- `context.timezone`
- `context.system`

Warning: All context variable values means:

```
$ - All default values that were set for variables in nodes that have been triggered during the current session.
- Any updates made to the default values with information provided by the user or external services during the current session.
```

To use the method, you can specify it in an expression in a variable that you define in the output object. For example:

```
$ {
  "output": {
    "generic": [
      {
        "values": [
          {
            "text": "Response for this node."
          }
        ],
        "response_type": "text",
        "selection_policy": "sequential"
      }
    ],
    "context_eraser": "<? context.clear() ?>"
  }
}
```

Clearing output

When you use the `clear()` method to clear the `output` object, it clears all variables except the one you use to clear the output object and any text responses that you define in the current node. It also does not clear these variables:

- `output.nodes_visited`
- `output.nodes_visited_details`

To use the method, you can specify it in an expression in a variable that you define in the output object. For example:

```
$ {
  "output": {
    "generic": [
      {
```

```

        "values": [
            {
                "text": "Have a great day!"
            }
        ],
        "response_type": "text",
        "selection_policy": "sequential"
    }
],
"output_eraser": "<? output.clear() ?>"
}
}

```

If a node earlier in the tree defines a text response of `I'm happy to help.` and then jumps to a node with the JSON output object defined earlier, then only `Have a great day.` is displayed as the response. The `I'm happy to help.` output is not displayed, because it is cleared and replaced with the text response from the node that is calling the `clear()` method.

JSONObject.has(String)

This method returns true if the complex JSONObject has a property of the input name.

For this Dialog runtime context:

```

{
  "context": {
    "user": {
      "first_name": "John",
      "last_name": "Snow"
    }
  }
}

```

Dialog node output:

```

{
  "conditions": "$user.has('first_name')"
}

```

Result: The condition is true because the user object contains the property `first_name`.

JSONObject.remove(String)

This method removes a property of the name from the input `JSONObject`. The `JSONElement` that is returned by this method is the `JSONElement` that is being removed.

For this Dialog runtime context:

```

{
  "context": {
    "user": {
      "first_name": "John",
      "last_name": "Snow"
    }
  }
}

```

Dialog node output:

```

{
  "context": {
    "attribute_removed": "<? $user.remove('first_name') ?>"
  }
}

```

Result:

```

{
  "context": {
    "user": {
      "last_name": "Snow"
    },

```

```
    "attribute_removed": {  
      "first_name": "John"  
    }  
  }  
}
```

com.google.gson.JsonObject support

In addition to the built-in methods, some of the standard methods of the `com.google.gson.JsonObject` class are also supported.

Strings

There methods help you work with text.

For information about how to recognize and extract certain types of Strings, such as people names and locations, from user input, see [System entities](#).

Note: For methods that involve regular expressions, see [RE2 Syntax reference](#) for details about the syntax to use when you specify the regular expression.

String.append(Object)

This method appends an input object to the string as a string and returns a modified string.

For this Dialog runtime context:

```
{  
  "context": {  
    "my_text": "This is a text."  
  }  
}
```

This syntax:

```
{  
  "context": {  
    "my_text": "<? $my_text.append(' More text.') ?>"  
  }  
}
```

Results in this output:

```
{  
  "context": {  
    "my_text": "This is a text. More text."  
  }  
}
```

String.contains(String)

This method returns true if the string contains the input substring.

Input: "Yes, I'd like to go."

This syntax:

```
{  
  "conditions": "input.text.contains('Yes')"  
}
```

Results: The condition is `true`.

String.endsWith(String)

This method returns true if the string ends with the input substring.

For this input:

```
"What is your name?".
```

This syntax:

```
{
  "conditions": "input.text.endsWith('?)"
}
```

Results: The condition is **true**.

String.equals(String)

This method returns **true** if the specified string equals the input string exactly.

Input: "Yes"

This syntax:

```
{
  "conditions": "input.text.equals('Yes')"
}
```

Results: The condition is **true**.

If the input is **Yes.**, then the result is **false** because the user included a period and the expression expects only the exact text, **Yes** without any punctuation.

String.equalsIgnoreCase(String)

This method returns **true** if the specified string equals the input string, regardless of whether the case of the letters match.

Input: "yes"

This syntax:

```
{
  "conditions": "input.text.equalsIgnoreCase('Yes')"
}
```

Results: The condition is **true**.

If the input is **Yes.**, then the result is **false** because the user included a period and the expression expects only the text, **Yes**, in uppercase or lowercase letters without any punctuation.

String.extract(String regexp, Integer groupIndex)

This method returns a string from the input that matches the regular expression group pattern that you specify. It returns an empty string if no match is found.



Note: This method is designed to extract matches for different regex pattern groups, not different matches for a single regex pattern. To find different matches, see the [getMatch](#) method.

In this example, the context variable is saving a string that matches the regex pattern group that you specify. In the expression, two regex patterns groups are defined, each one enclosed in parentheses. There is an inherent third group that is comprised of the two groups together. This is the first (groupIndex 0) regex group; it matches with a string that contains the full number group and text group together. The second regex group (groupIndex 1) matches with the first occurrence of a number group. The third group (groupIndex 2) matches with the first occurrence of a text group after a number group.

```
{
  "context": {
    "number_extract": "<? input.text.extract('([\\d]+)(\\b [A-Za-z]+)',n) ?>"
  }
}
```

When you specify the regex in JSON, you must provide two backslashes (\\). If you specify this expression in a node response, you need one backslash only. For example:

```
<? input.text.extract('([\\d]+)(\\b [A-Za-z]+)',n) ?>
```

Input:

```
"Hello 123 this is 456".
```

Result:

- When n=**0**, the value is **123 this**.
- When n=**1**, the value is **123**.
- When n=**2**, the value is **this**.

String.find(String regexp)

This method returns true if any segment of the string matches the input regular expression. You can call this method against a JSONArray or JSONObject element, and it will convert the array or object to a string before making the comparison.

For this input:

```
"Hello 123456".
```

This syntax:

```
{
  "conditions": "input.text.find('^[^\\d]*[\\d]{6}[^\\d]*$')"
}
```

Result: The condition is true because the numeric portion of the input text matches the regular expression `^[^\\d]*[\\d]{6}[^\\d]*$`.

String.getMatch(String regexp, Integer matchIndex)

This method returns a string from the input that matches the occurrence of the regular expression pattern that you specify. This method returns an empty string if no match is found.

As matches are found, they are added to what you can think of as a *matches array*. If you want to return the third match, because the array element count starts at 0, specify 2 as the `matchIndex` value. For example, if you enter a text string with three words that match the specified pattern, you can return the first, second, or third match only by specifying its index value.

In the following expression, you are looking for a group of numbers in the input. This expression saves the second pattern-matching string into the `$second_number` context variable because the index value 1 is specified.

```
{
  "context": {
    "second_number": "<? input.text.getMatch('([\\d]+)',1) ?>"
  }
}
```

If you specify the expression in JSON syntax, you must provide two backslashes (\\). If you specify the expression in a node response, you need one backslash only.

For example:

```
<? input.text.getMatch('([\\d]+)',1) ?>
```

- User input:

```
"hello 123 i said 456 and 8910".
```

- Result: **456**

In this example the expression looks for the third block of text in the input.

```
<? input.text.getMatch('(\\b [A-Za-z]+)',2) ?>
```

For the same user input, this expression returns **and**.

String.isEmpty()

This method returns true if the string is an empty string, but not null.

For this Dialog runtime context:

```
{
  "context": {
    "my_text_variable": ""
  }
}
```

This syntax:


```
{
  "conditions": "$my_text_variable.isEmpty()"
}
```

Results: The condition is `true` .

String.length()

This method returns the character length of the string.

For this input:

```
"Hello"
```

This syntax:

```
{
  "context": {
    "input_length": "<? input.text.length() ?>"
  }
}
```

Results in this output:

```
{
  "context": {
    "input_length": 5
  }
}
```

String.matches(String regexp)

This method returns true if the string matches the input regular expression.

For this input:

```
"Hello".
```

This syntax:

```
{
  "conditions": "input.text.matches('^Hello$')"
}
```

Result: The condition is true because the input text matches the regular expression `^Hello$` .

String.startsWith(String)

This method returns true if the string starts with the input substring.

For this input:

```
"What is your name?".
```

This syntax:

```
{
  "conditions": "input.text.startsWith('What')"
}
```

Results: The condition is `true` .

String.substring(Integer beginIndex, Integer endIndex)

This method gets a substring with the character at `beginIndex` and the last character set to index before `endIndex` . The endIndex character is not included.

For this Dialog runtime context:

```
{
  "context": {
    "my_text": "This is a text."
  }
}
```

This syntax:

```
{
  "context": {
    "my_text": "<? $my_text.substring(5, $my_text.length()) ?>"
  }
}
```

Results in this output:

```
{
  "context": {
    "my_text": "is a text."
  }
}
```

String.toJson()

This method parses a string that contains JSON data and returns a JSON object or array, as in this example:

```
$ ${json_var}.toJson()
```

If the context variable `${json_var}` contains the following string:

```
$ "{ \"firstname\": \"John\", \"lastname\": \"Doe\" }"
```

the `toJson()` method returns the following object:

```
$ {
  "firstname": "John",
  "lastname": "Doe"
}
```

String.toLowerCase()

This method returns the original String converted to lowercase letters.

For this input:

```
"This is A DOG!"
```

This syntax:

```
{
  "context": {
    "input_lower_case": "<? input.text.toLowerCase() ?>"
  }
}
```

Results in this output:

```
{
  "context": {
    "input_lower_case": "this is a dog!"
  }
}
```

String.toUpperCase()

This method returns the original String converted to uppercase letters.

For this input:

```
"hi there".
```

This syntax:

```
{
  "context": {
    "input_upper_case": "<? input.text.toUpperCase() ?>"
  }
}
```

Results in this output:

```
{
  "context": {
    "input_upper_case": "HI THERE"
  }
}
```

String.trim()

This method trims any spaces at the beginning and the end of the string and returns the modified string.

For this Dialog runtime context:

```
{
  "context": {
    "my_text": " something is here  "
  }
}
```

This syntax:

```
{
  "context": {
    "my_text": "<? $my_text.trim() ?>"
  }
}
```

Results in this output:

```
{
  "context": {
    "my_text": "something is here"
  }
}
```

java.lang.String support

In addition to the built-in methods, you can use standard methods of the `java.lang.String` class.

java.lang.String.format()

You can apply the standard Java String `format()` method to text. See [java.util.formatter reference](#) for information about the syntax to use to specify the format details.

For example, the following expression takes three decimal integers (1, 1, and 2) and adds them to a sentence.

```
{
  "formatted String": "<? T(java.lang.String).format('%d + %d equals %d', 1, 1, 2) ?>"
}
```

Result: `1 + 1 equals 2`.

To change the decimal placement for a number, use the following syntax:

```
{
  <? T(String).format('%.2f',<number to format>) ?>
}
```

For example, if the \$number variable that needs to be formatted in US dollars is `4.5`, then a response such as, `Your total is $<?T(String).format('%.2f',$number) ?>` returns `Your total is $4.50.`

Indirect data type conversion

When you include an expression within text, as part of a node response, for example, the value is rendered as a String. If you want the expression to be rendered in its original data type, then do not surround it with text.

For example, you can add this expression to a dialog node response to return the entities that are recognized in the user input in String format:

The entities are `<? entities ?>`.

If the user specifies *Hello now* as the input, then the @sys-date and @sys-time entities are triggered by the `now` reference. The entities object is an array, but because the expression is included in text, the entities are returned in String format, like this:

The entities are 2018-02-02, 14:34:56.

If you do not include text in the response, then an array is returned instead. For example, if the response is specified as an expression only, not surrounded by text.

`<? entities ?>`

The entity information is returned in its native data type, as an array.

```
[
  {
    "entity":"sys-date","location":[6,9],"value":"2018-02-02","confidence":1,"metadata":{"calendar_type":"GREGORIAN","timezone":"America/New_York"}
  },
  {
    "entity":"sys-time","location":[6,9],"value":"14:33:22","confidence":1,"metadata":{"calendar_type":"GREGORIAN","timezone":"America/New_York"}
  }
]
```

As another example, the following \$array context variable is an array, but the \$string_array context variable is a string.

```
{
  "context": {
    "array": [
      "one",
      "two"
    ],
    "array_in_string": "this is my array: $array"
  }
}
```

If you check the values of these context variables in the Try it out pane, you will see their values specified as follows:

\$array: `["one", "two"]`

\$array_in_string: `"this is my array: [\"one\\\", \"two\\\"]"`

You can subsequently perform array methods on the \$array variable, such as `<? $array.removeValue('two') ?>`, but not the \$array_in_string variable.



Important: This documentation is for the **classic Watson Assistant** experience. To see the documentation for the new Watson Assistant, please go [here](#).

Expressions for accessing objects

You can write expressions that access objects and properties of objects by using the Spring Expression (SpEL) language. For more information, see [Spring Expression Language \(SpEL\)](#).

Evaluation syntax

To expand variable values inside other variables or invoke methods on properties and global objects, use the `<? expression ?>` expression syntax. For example:

- Expanding a property

```
"output":{"text":"Your name is <? context.userName ?>"}
```

- Invoking methods on properties of global objects

```
"context":{"email": "<? @email.literal ?>"}
```

Shorthand syntax

Learn how to quickly reference the following objects by using the SpEL shorthand syntax:

- [Context variables](#)
- [Entities](#)
- [Intents](#)

Shorthand syntax for context variables

The following table shows examples of the shorthand syntax that you can use to write context variables in condition expressions.

| Shorthand syntax | Full syntax in SpEL |
|--|--|
| <code>\$card_type</code> | <code>context['card_type']</code> |
| <code>\$(card-type)</code> | <code>context['card-type']</code> |
| <code>\$card_type:VISA</code> | <code>context['card_type'] == 'VISA'</code> |
| <code>\$card_type:(MASTER CARD)</code> | <code>context['card_type'] == 'MASTER CARD'</code> |

You can include special characters, such as hyphens or periods, in context variable names. However, doing so can lead to problems when the SpEL expression is evaluated. The hyphen could be interpreted as a minus sign, for example. To avoid such problems, reference the variable by using either the full expression syntax or the shorthand syntax `$(variable-name)` and do not use the following special characters in the name:

- Parentheses `()`
- More than one apostrophe `''`
- Quotation marks `"`

When you refer to a context variable in a text response or a dialog node condition, you can use the short syntax.

For example, `Hello, $name`. If the `$name` context variable contains `Sam`, then the response is shown as `Hello, Sam`.

If you want to reference a context variable by using the full syntax in a text response, be sure to surround the context variable in `<? ?>`. For example, `Hello, <? context['name'] ?>`.

If you want to reference a context variable that has multiple fields, such as `$context.integrations.chat.browser_info.page_url`. To use the full syntax, specify `<? context['integrations']['chat']['browser_info']['page_url'] ?>`.

Shorthand syntax for entities

The following table shows examples of the shorthand syntax that you can use when referring to entities.

| Shorthand syntax | Full syntax in SpEL |
|--------------------|--------------------------------------|
| <code>@year</code> | <code>entities['year']?.value</code> |

| | |
|-------------------|--|
| @year == 2016 | entities['year']?.value == 2016 |
| @year != 2016 | entities['year']?.value != 2016 |
| @city == 'Boston' | entities['city']?.value == 'Boston' |
| @city:Boston | entities['city']?.contains('Boston') |
| @city:(New York) | entities['city']?.contains('New York') |

In SpEL, the question mark `(?)` prevents a null pointer exception from being triggered when an entity object is null.

If the entity value that you want to check for contains a `)` character, you cannot use the `:` operator for comparison. For example, if you want to check whether the city entity is `Dublin (Ohio)`, you must use `@city == 'Dublin (Ohio)'` instead of `@city:(Dublin (Ohio))`.

Shorthand syntax for intents

The following table shows examples of the shorthand syntax that you can use when referring to intents.

| Shorthand syntax | Full syntax in SpEL |
|--------------------------------------|--|
| <code>`#help`</code> | <code>`intent == 'help'`</code> |
| <code>`!#help`</code> | <code>`intent != 'help'`</code> |
| <code>`NOT #help`</code> | <code>`intent != 'help'`</code> |
| <code>`#help` or `#i_am_lost`</code> | <code>`(intent == 'help' intent == 'I_am_lost')`</code> |

Intents shorthand syntax

Built-in global variables

You can use the expression language to extract property information for the following global variables:

| Global variable | Definition |
|--------------------|--|
| <i>context</i> | JSON object part of the processed conversation message. |
| <i>entities[]</i> | List of entities that supports default access to 1st element. |
| <i>input</i> | JSON object part of the processed conversation message. |
| <i>intents[]</i> | List of intents that supports default access to first element. |
| <i>output</i> | JSON object part of the processed conversation message. |

Accessing entities

The entities array contains one or more entities that were recognized in user input.

While testing your dialog, you can see details of the entities that are recognized in user input by specifying this expression in a dialog node response:

```
<? entities ?>
```

For the user input, *today*, your assistant recognizes the @sys-date system entity, so the response contains this entity object:

```
[
  {
    "entity":"sys-date",
    "location":[0,5],
    "value":"2020-12-30",
```

```
"confidence":1.0,
"metadata":
{
  "calendar_type":"GREGORIAN",
  "timezone":"America/New_York"
},
"interpretation":
{
  "timezone":"America/New_York",
  "relative_day":0,
  "granularity":"day",
  "calendar_type":"GREGORIAN"
}
}
```

If you want to include text in the response, use the `toJson()` method in the expression to cast the returned entities list into a JSON object. For example:

Recognized entities are: `<? entities.toJson() ?>`

When placement of entities in the input matters

When you use the shorthand expression, `@city.contains('Boston')`, in a condition, the dialog node returns true **only if** `Boston` is the first entity detected in the user input. Only use this syntax if the placement of entities in the input matters to you and you want to check the first mention only.

Use the full SpEL expression if you want the condition to return true any time the term is mentioned in the user input, regardless of the order in which the entities are mentioned. The condition, `entities['city']?.contains('Boston')` returns true when at least one 'Boston' city entity is found in all the @city entities, regardless of placement.

For example, a user submits `"I want to go from Toronto to Boston."` The `@city:Toronto` and `@city:Boston` entities are detected and are represented in the array that is returned as follows:

- `entities.city[0].value = 'Toronto'`
- `entities.city[1].value = 'Boston'`



Note: The order of entities in the array that is returned matches the order in which they are mentioned in the user input.

Entity properties

Each entity has a set of properties associated with it. You can access information about an entity through its properties.

| Property | Definition | Usage tips |
|-------------------|--|---|
| <i>confidence</i> | A decimal percentage that represents your assistant's confidence in the recognized entity. The confidence of an entity is either 0 or 1, unless you have activated fuzzy matching of entities. When fuzzy matching is enabled, the default confidence level threshold is 0.3. Whether or not fuzzy matching is enabled, system entities always have a confidence level of 1.0. | You can use this property in a condition to have it return false if the confidence level is not higher than a percent you specify. |
| <i>location</i> | A zero-based character offsets that indicates where the detected entity values begin and end in the input text. | Use <code>.literal</code> to extract the span of text between start and end index values that are stored in the location property. |
| <i>value</i> | The entity value identified in the input. | This property returns the entity value as defined in the training data, even if the match was made against one of its associated synonyms. You can use <code>.values</code> to capture multiple occurrences of an entity that might be present in user input. |

Entity property usage examples

In the following examples, the skill contains an airport entity that includes a value of JFK, and the synonym 'Kennedy Airport'. The user input is `I want to go to Kennedy Airport`.

- To return a specific response if the 'JFK' entity is recognized in the user input, you could add this expression to the response condition:
`entities.airport[0].value == 'JFK'` or `@airport = "JFK"`
- To return the entity name as it was specified by the user in the dialog response, use the `.literal` property: `So you want to go to <?`

```
entities.airport[0].literal?>... or So you want to go to @airport.literal ...
```

Both formats evaluate to `So you want to go to Kennedy Airport...` in the response.

- Expressions like `@airport:(JFK)` or `@airport.contains('JFK')` always refer to the **value** of the entity (`JFK` in this example).
- To be more restrictive about which terms are identified as airports in the input when fuzzy matching is enabled, you can specify this expression in a node condition, for example: `@airport && @airport.confidence > 0.7`. The node will only execute if your assistant is 70% confident that the input text contains an airport reference.

In this example, the user input is *Are there places to exchange currency at JFK, Logan, and O'Hare?*

- To capture multiple occurrences of an entity type in user input, use syntax like this:

```
$ "context":{
  "airports":"@airport.values"
}
```

To later refer to the captured list in a dialog response, use this syntax: `You asked about these airports: <? $airports.join(', ') ?>.` It is displayed like this: `You asked about these airports: JFK, Logan, O'Hare.`

- To capture the literal values for multiple entity mentions, use the following syntax:

```
$ entities['myEntityName'].![literal]
```

Accessing intents

The intents array contains one or more intents that were recognized in the user input, sorted in descending order of confidence.

Each intent has one property only: the **confidence** property. The confidence property is a decimal percentage that represents your assistant's confidence in the recognized intent.

While testing your dialog, you can see details of the intents that are recognized in user input by specifying this expression in a dialog node response:

```
<? intents ?>
```

For the user input, *Hello now*, your assistant finds an exact match with the #greeting intent. Therefore, it lists the #greeting intent object details first. The response also includes the top 10 other intents that are defined in the skill regardless of their confidence score. (In this example, its confidence in the other intents is set to 0 because the first intent is an exact match.) The top 10 intents are returned because the "Try it out" pane sends the **alternate_intents:true** parameter with its request. If you are using the API directly and want to see the top 10 results, be sure to specify this parameter in your call. If **alternate_intents** is false, which is the default value, only intents with a confidence above 0.2 are returned in the array.

```
[{"intent":"greeting","confidence":1},
{"intent":"yes","confidence":0},
{"intent":"pizza-order","confidence":0}]
```

If you want to include text in the response, use the **toJson()** method in the expression to cast the returned intents list into a JSON object. For example:

```
Recognized intents are: <? intents.toJson() ?>
```

The following examples show how to check for an intent value:

- `intents[0] == 'Help'`
- `intent == 'Help'`

`intent == 'help'` differs from `intents[0] == 'help'` because `intent == 'help'` does not throw an exception if no intent is detected. It is evaluated as true only if the intent confidence exceeds a threshold. If you want to, you can specify a custom confidence level for a condition, for example, `intents.size() > 0 && intents[0] == 'help' && intents[0].confidence > 0.1`

Accessing input

The input JSON object contains one property only: the text property. The text property represents the text of the user input.

Input property usage examples

The following example shows how to access input:

- To execute a node if the user input is "Yes", add this expression to the node condition: `input.text == 'Yes'`

You can use any of the [String methods](#) to evaluate or manipulate text from the user input. For example:

- To check whether the user input contains "Yes", use: `input.text.contains('Yes')`.
- Returns true if the user input is a number: `input.text.matches('[0-9]+')`.
- To check whether the input string contains ten characters, use: `input.text.length() == 10`.