Data types (Jira expressions)

On this page

General Information

Throughout the documentation we refer to **data types** that can be used in **Jira expression**. The most commonly use data types are listed below. Apart from this list, learn more about data types in Jira expressions in the general documentation.

Data type	Description	Example
NUMBER	This type represents numeric values.	1, 1.1, -1.1, .1, -.1
NUMBER LIST	This type represents a collection of numeric values. The size may vary from 0 to any number of numeric values. It is used to read the value of a numeric field in a selection of issues. You can also use literals like `[1, 2, 3]` .	[1, 2, 3]
TEXT	This type represents any kind of text or character string including all kinds of select and multi-select fields.	"Hello world"
TEXT LIST	This type represents a collection of textstring values. The size may vary from 0 to any number of string values. It is also used to read the value of a string field in a selection of issues. You can also use literals like `["string_A", "string_B", "string_C"]` .	["string_A", "string_B", "string_C"]
BOOLEAN	A **logical**, or **boolean,** value of `true` or `false`.	true

Numbers

NUMBER

All numbers in Jira expressions are double-precision 64-bit IEEE 754 floating points. The usual set of mathematical operations is available.

Strings can be converted to numbers with the **Number** function. For example:

```
Number('1') + Number('2') == 3
```

⚠️ Note that if a string cannot be parsed as number, the function returns *NaN* (Not a Number).

Texts

TEXT

Texts are based on the JavaScript String object.

Currently supported properties and functions are:

Function	Output	Returned value
`length`	NUMBER	The text **length**
`trim()`	TEXT	**Removes whitespaces** from beginning and end
`toLowerCase()`	TEXT	Returns the same string with all characters in **lowercase**
`toUpperCase()`	TEXT	Returns the same string with all characters in **uppercase**
`split(string?)`	TEXT LIST	**Splits** the string with the given **separator**

`replace(string, string|string => string)`	TEXT	**Replaces** all occurrences of the first argument with the second argument, which can also be a function that accepts the matched part
`match(string)`	TEXT LIST	**Finds all matches** of the given regular expression in this string
`includes(string)`	BOOLEAN	Returns **true** if this string contains the given string, **false** otherwise
`indexOf(string)`	NUMBER	Returns the **index** of the **first occurrence** of the given string in this string, or -1
`slice(number, number?)`	TEXT	Returns a **substring** of this string, according to the given arguments

Rich text

This object represents fields with rich text formatting. Currently it allows to retrieve only plain text, but in the future it will also contain Atlassian Document Format.

- `plainText` : The plain text stored in the field (TEXT).

To access such a property you can easily call it by using an expression like:

- `issue.description.plainText`

Dates

DATE

This object is based on the JavaScript Date API.

Read more about dates and times in Jira expressions in the general documentation.

Jira expressions provide these **additional functions**:

Function	Output	Returned value
`toString()`	STRING	Returns a **string** in the human-readable format, according to the current user's locale and timezone
`toCalendarDate()`	CALENDAR DATE	Transforms this into a **calendar date**, according to the current user's locale and timezone
`toCalendarDateUTC()`	CALENDAR DATE	Transforms this into a **calendar date** in the **UTC** timezone
`plusMonths(number)`	DATE	Returns a date with the given number of **months added**
`minusMonths(number)`	DATE	Returns a date with the given number of **months removed**
`plusDays(number)`	DATE	Returns a date with the given number of **days added**
`minusDays(number)`	DATE	Returns a date with the given number of **days removed**
`plusHours(number)`	DATE	Returns a date with the given number of **hours added**
`minusHours(number)`	DATE	Returns a date with the given number of **hours removed**
`plusMinutes(number)`	DATE	Returns a date with the given number of **minutes added**
`minusMinutes (number)`	DATE	Returns a date with the given number of **minutes removed**

Constructors

- **`new Date()`**: Creates a date that represents the current time.
- **`new Date(number)`**: Creates a date based on a number of milliseconds that elapsed since the Unix epoch.
- **`new Date(string)`**: Creates a date based on a string in the ISO 8601 format (for example, `2008-09-15T15:53:00+05:00`). The current user's timezone is used if none is included in the string.

Calendar Date

CALENDAR DATE

A time-zone agnostic Date with the same set of methods, but limited only to year, month, and day.

Constructors

- **`new CalendarDate(string)`**: Creates a calendar date based on a string in the **yyyy-MM-dd** format. For example, `2018-09-15`.

Lists

STRING LIST NUMBER LIST

Lists are a basic building block of Jira expressions. By design, the language does not support imperative constructs, so instead of writing loops, you need to employ the functional style of processing lists with lambda functions.

For example, to return the **number of comments** with contents **longer than 100 characters:**

1. first **map** the comments to their texts
2. then **filter** them to leave only those long enough
3. and finally get the **length** of the resulting list:

```
issue.comments
     .map(c => c.body.plainText)
     .filter(text => text.length > 100)
     .length
```

You can access individual elements of a list by using an index, e.g. `issue.attachments[0].author.displayName` returns the name of the author of the issue's first attachment.

The following properties and functions are available for lists:

Function	Output	Returned value
`length`	NUMBER	Returns the **number of items** stored in the list
`map(Any => Any)`	TEXT LIST	**Maps all items** in the list to the result of the provided function
`sort((Any, Any) => Number)`	TEXT LIST	Returns the **list sorted** by the natural ordering of elements or by the optional comparison function
`filter(Any => Boolean)`	TEXT LIST	Leaves only items that do satisfy the given function, that is, for which the given function returns **true**
`every(Any => Boolean)`	BOOLEAN	Checks if all elements in the list **satisfy** the given **predicate**
`some(Any => Boolean)`	BOOLEAN	Checks if the list contains at least one element that **satisfies** the given **predicate**
`includes(Any)`	BOOLEAN	Checks if the **given argument** is stored in the list
`indexOf(Any)`	NUMBER	Returns the **index of the first occurrence** of the item in the list, or -1
`slice(Number, Number?)`	TEXT LIST	Returns a **portion of the list**, with the index starting from the first argument (inclusive), and ending with the second one (exclusive). The second argument is optional, if not provided, all remaining elements will be returned. Negative numbers are allowed and mean indexes counted from the end of the list

`flatten()`	TEXT LIST	**Flattens** a multi-dimensional list
`flatMap(Any => Any)`	TEXT LIST	**Maps** all items in the list and **flattens** the result
`reduce(Any => Any, Any?)`	TEXT LIST	**Aggregates** all elements of the list using the function provided in the first argument. The operation starts from the first element of the list, unless the initial value is provided in the optional second argument. If the list is empty and no initial value is given, an error will be returned.

Maps

MAP

If the returned property value is a JSON object, it will be converted to a **Map**.

- Static or dynamic member access can be used to retrieve values from a map. For example, `map.key` is the same as `map['key']`.
- Values can also be accessed using the **`get()`** method. For example, `map.get('key')`.
- Both of these methods will return **`null`** if there is no mapping for the given key.

To create a new map, write **`new Map()`**. Object literals are also evaluated to the **Map** object. For example, `{ id: issue.id, summary: issue.summary }` will evaluate to a map with two keys: *id* and *summary*.

Apart from static and computed member access, the following methods are available for maps:

- **`get(string)`**: Returns the value mapped to the given key, or `null` (Any).
- **`set(string, Any)`**: Returns a new map that has all entries from the current map, plus the first argument mapped to the second (Map).
- **`entries()`**: Returns a list of all entries in this map, each entry returned as a two-element list of key and value (List<[String, Any]>).

Constructors

- **`new Map()`**: Creates an empty map. Equivalent to **{}**.

Optional chaining

Accessing properties in a Jira expression may fail, for example, where:

- the left-hand side of the operation is `null`. For example, in the expression a.b where the value of `a` is `null`.
- the property does not exist.

In expressions where such strict rules are not desired, use the optional chaining operator `?.`. This operator behaves in the same way as regular member access, but with one crucial difference: when accessing the property fails, `null` is returned.

Examples:

- `issue.properties?.myProperty?.a?.b`—this expression returns `null` if there is no `myProperty` defined in the issue, or if there is no `a.b` path in the value of the property.
- `issue?.customfield_10010`—this expression returns `null` if the custom field doesn't exist.

The operator can also be used in combination with computed member access, for example: `issue?.[fieldName]`.

Boolean

BOOLEAN

There are two boolean values: **`true`** and **`false`**.

The usual set of logical operators, with behavior following the rules of classical boolean algebra, is available:

Operator	Example
conjunction	`a && b`
disjunction	`a || b`
negation	`!a`