S-JQL Reference

`structure()` JQL Function Reference

To specify a structure condition in JQL, use the following format:

issue in structure(*structureName*$_{opt}$, *structureQuery*$_{opt}$**)**

Function arguments:

structure Name	*Optional*	The name of the structure. If you omit the structure name, system-wide Default Structure will be searched. Remember to enclose the name in double quotes ("") if it contains spaces or non-letters.
structure Query	*Optional*	Use this parameter to select only a part of the structure. This parameter specifies a *Structure Query* in a language similar to JQL, *Structured JQL*, which is discussed below.

You can use structure ID instead of the structure name. You can see structure ID in the URL of the Structure Board if you open **Manage Structure** page and click structure name.

If a user does not have access to structure, they will not be able to create new queries with the `structure()` function and existing queries will have `structure()` function return an empty set. However, the user will still see `structure()` function offered in the JQL completion drop-down.

Structured JQL Language Reference

Structure query is a hierarchical condition on the issues added to the structure. Structure query is expressed in the Structured JQL language (S-JQL), described in this section.

This reference assumes that you are familiar with Advanced Searching and Advanced Searching Functions capabilities of JIRA.

List of Structured JQL topics:

- Constraints
- Basic constraint
- Negation
- Relational constraint
  - Relations
  - Operators
  - Sub-constraints
  - `issue` relation: adding sub-constraint matches to the result set
- Combining constraints with Boolean operators
- Quoting structure query argument in the `structure()` JQL function
- Backward compatibility with `structure()` JQL function prior to Structure 2.4
- Railroad diagrams

Constraints

Structure query consists of *constraints*. Each constraint matches some issues in the structure. In the simplest case, the whole structure query consists of a single constraint; for now, we will consider only this case.

There are two types of constraints: *basic* and *relational* constraints.

^ up to the list of S-JQL topics

Basic constraint

A basic constraint is one of the following:

1. A JQL query enclosed in square brackets:

```
[status = Open]
```

This kind of basic contraint matches all issues in the structure that satisfy the JQL query.

2. Issues having special positions within the structure:

```
leaf
```

or

```
root
```

The first constraint matches issues at the bottom level of the hierarchy, i.e., issues that do not have children.
The second constraint matches issues at the top level of the hierarchy, i.e., issues that do not have a parent.

3. A comma-separated list of issues:

```
TS-129, TS-239
```

or just a single issue:

```
TS-129
```

You can specify issue key, as above, or issue ID:

```
19320
```

This kind of basic constraint matches just the referenced issues. If some of the issues are not contained within the structure, they are ignored. If none of the issues are contained within the structure, the constraint matches no issues.

4. An empty constraint matching no issues:

```
empty
```

This constraint plays the same role as JQL's `EMPTY` keyword. It is intended to be used as a sub-constraint in relational constraints, which are discussed further.

^ up to the list of S-JQL topics

Negation

Any constraint, basic or relational, can be negated using keyword `NOT`. This produces a constraint that matches all issues that the original constraint doesn't:

```
not root
```

matches all issues that are not top-level issues in the structure.

You can always enclose a constraint in parentheses to ease understanding. So, all issues in the structure except `TS-129` and `TS-239` are matched by this structure query:

```
not (TS-129, TS-239)
```

^ up to the list of S-JQL topics

Relational constraint

A basic constraint matches issues that satisfy a condition. A relational constraint matches issues *related to* issues that satisfy a condition. *Related* corresponds to a relationship between issues in the structure, like parent-child.
For example,

```
TS-129
```

is a basic constraint that matches a single issue `TS-129`;

```
child in TS-129
```

is a relational constraint matching an issue such that its child is `TS-129`.

Relational constraint has the form `relation operator subConstraint`. Here, `subConstraint` is a constraint on the relatives of issues to be matched; other parts of relational constraint are discussed in the following sections.

Note that the form of relational constraint is similar to the form of JQL clause, `field operator value`.
Indeed, let's describe in English a JQL query `type in (Epic, Story)`: it matches issues having *type* that is *in* values *Epic, Story*.
Now, let's describe in English a structure query `parent in [type = Epic]`: it matches issues having *parent* that is *in* constraint "`type = Epic`".
As you can see, the form that can be used to describe the structure query is similar to that of JQL.

^ up to the list of S-JQL topics

Relations

S-JQL has the following relations:

- `child`: issue is a child (sub-issue) of another issue in the structure.
- `parent`: issue is a parent of another issue in the structure.
- `descendant`: issue is a descendant (sub- or sub-sub-...-issue) of another issue in the structure.
- `ancestor`: issue is an ancestor (parent, parent-of-parent, or parent-of-parent-...-of-parent) of another issue in the structure.
- `sibling`: issue is a sibling of another issue in the structure. Two issues are considered siblings if they are under the same parent issue.
- `prevSibling`: issue is a previous (preceding) sibling of another issue in the structure.
  Issue *A* is a preceding sibling of issue *B* if it is a sibling of *B* and *A* is higher than *B* (*A* comes before *B*.)
- `nextSibling`: issue is a next (following) sibling of another issue in the structure.
  Issue *A* is a following sibling of issue *B* if it is a sibling of *B* and *A* is lower than *B* (*A* comes after *B*.)
- `issue` is a relation of an issue to itself. Its role is explained later, in the `issue` relation section, because first one has to grok how operators and sub-constraints work.
  There are also combinations of `issue` with all other relations, listed for completeness below.
- `childOrIssue`
- `parentOrIssue`
- `descendantOrIssue`
- `ancestorOrIssue`
- `siblingOrIssue`
- `prevSiblingOrIssue`
- `nextSiblingOrIssue`

Those familiar with XPath may have recognized these relations; indeed, they work like the corresponding XPath axes.

^ up to the list of S-JQL topics

Operators

These are the operators used in S-JQL:

```
IN, NOT IN, IS, IS NOT, =, !=, OF
```

`operator` specifies how `subConstraint` is applied to `relation`:

1. `IN`, `IS`, and = put constraint on the relatives of a matched issue.

   For example, consider

   ```
   child in (TS-129, TS-239)
   ```

Here, `relation` is `child`, so an issue's relative in question is its child in the structure. Thus, an issue matches if *at least one of its children is `TS-129` or `TS-239`*.

There is no difference between these three operators, unlike JQL. Different forms exist to allow for more natural-looking queries with some sub-constraints.

2. `NOT IN`, `IS NOT`, and `!=` are negated versions of `IN`, `IS`, and =. That is, an issue is matched if it *is not related to* any issue matching `subConstraint`.

As an important consequence, issue that has no relatives is matched.

For example, consider

```
child not in (TS-129, TS-239)
```

An issue matches if *no child is `TS-129` nor `TS-239`*; thus, this constraint matches all issues that either have no children or do not have any of these two issues among their children.

Using one of these operators in a relational constraint is the same as using `IN` (or `IS`, or =) and negating the whole relational constraint. Thus, the constraint above is equivalent to

```
not (child in (TS-129, TS-239))
```


But, using one of these operators is **very not** the same as using operator `IN` and negating `subConstraint`!

First, *having relatives other than X* is not the same as *not having relatives X*. Think of it as of relationships in a human family: having a relative other than brother (e.g., a sister) is **not** the same as not having a brother, because one may have both a sister and a brother. Second, an issue with no relatives is not matched by the transformed query.

For example,

```
child in (not (TS-129, TS-239))
```

matches all issues that have at least one child that is neither `TS-129` nor `TS-239`. That is, the only issues that are not matched are leaves and those that have only `TS-129` or `TS-239` as children.

3. `OF` matches the relatives of issues that satisfy `subConstraint`.

For example, consider

```
child of (TS-129, TS-239)
```

An issue matches if *it is a child of either `TS-129` or `TS-239`*.

To have a model of how operators `IN` (`IS`, =) and `OF` work and to understand the difference between them, consider the table below. Suppose that we take all issues in the structure and put each of them, one by one, in column **issue**. For each issue, we take all of its relatives and put each of them, one by one, in column **relative**. Thus we get pairs of issues. We examine each pair, and if one of the components satisfies *subConstraint*, we add the other component to the result set. Which component is added, depends on the operator:

operator	issue	relative
`in`	*add to result set*	*satisfies* `subConstraint`
`of`	*satisfies* `subConstraint`	*add to result set*

One may note that for any relation, there is a corresponding "inverse": for example, `child` is the inverse of `parent`, and vice versa. A relational constraint that uses operator `IN` (`IS`, `=`) is equivalent to a relational constraint that uses an inverse relation with operator `OF`. That is,

```
child in (TS-129, TS-239)
```

is the same as

```
parent of (TS-129, TS-239)
```

Again, different forms of expressing the same constraint exist to allow for more natural-looking queries.

^ up to the list of S-JQL topics

Sub-constraints

Any constraint can be used as a sub-constraint, whether basic, relational, or a combination of those.
For example,

```
child of root
```

selects issues on the second level of the hierarchy. To select issues on the third level of the hierarchy, you can once again use relation `child` and the previous query as `subConstraint`:

```
child of (child of root)
```

There is a special basic constraint, `empty`, which matches no issues. It is used as a sub-constraint to match issues that have no relatives as per `relation`

For example, let's take relation `child` and see what the corresponding relational constraints with different operators mean.

`child is empty`	matches all issues that have no children (equivalent of `leaf`)
`child is not empty`	matches all issues that have at least one child (equivalent of `not leaf`)
`child of empty`	matches all issues that are not children of other issues (equivalent of `root`)

Of course, using `leaf` or `root` is more convenient, but you can apply `empty` to any other relation. For instance, `sibling is empty` matches an issue if it is the only child of its parent.

^ up to the list of S-JQL topics

`issue` relation: adding sub-constraint matches to the result set

A relational constraint with relation `issue` behaves exactly as its sub-constraint, possibly negated if operator `NOT IN` (`IS NOT`, `!=`) is used.
Thus,

```
issue in [status = Open]
```

is equivalent to

```
[status = Open]
```

Similarly,

```
issue not in [status = Open]
```

is equivalent to

```
not [status = Open]
```

When combined with another relation, `issue` allows to add the issues matched by `subConstraint` to the resulting set. For example,

```
descendant of TS-129
```

returns all of the children of `TS-129` at all levels, but does not return `TS-129` itself. To add `TS-129`, use `descendantOrIssue`:

```
descendantOrIssue of TS-129
```

^ up to the list of S-JQL topics

Combining constraints with Boolean operators

We can now define a structure query as a *Boolean combination of constraints*, that is, a structure query consists of constraints connected with `AND` and `OR`. When two constraints are connected with `AND`, together they will match issues that are matched by both constraints. This allows you to limit the results. Likewise, when two constraints are connected by `OR`, together they will match issues that are matched by at least one of the constraints. This allows you to expand the results.

Note that `AND` has higher precedence than `OR`. That means that the Structure query

```
leaf or (parent of leaf) and [status = Open]
```

matches all issues that are either leaves, or are parents of leaves in status *Open*. In order to also constrain leaf issues to be in the status *Open*, you need to use parentheses:

```
(leaf or (parent of leaf)) and [status = Open]
```

^ up to the list of S-JQL topics

Quoting structure query argument in the `structure()` JQL function

When specifying structure query as a parameter of the `structure()` JQL function, you should enclose it in "double quotes" or 'single quotes' if it contains spaces or non-letters. Please note that if you are using quotes of one kind, you cannot use quotes of the same kind in the inner JQL constraint.

This query will not parse:

```
issue in structure("My personal structure", "child of [Status = "Awaiting Deployment"]")
```

You should use single quotes in the inner JQL constraint instead:

```
issue in structure("My personal structure", "child of [Status = 'Awaiting Deployment']")
```

If some values in the inner JQL constraint contain quotes, you should escape them with a backslash:

```
issue in structure("My personal structure", "child of [fixVersion = 'funky\"Version']")
```

^ up to the list of S-JQL topics

Backward compatibility with `structure()` JQL function prior to Structure 2.4

Prior to Structure 2.4, `structure()` JQL function did not take structure query as an argument; you could specify only one issue key or ID, and you would get the referenced issue along with all of its children at all levels. As you might have noticed, this old-style usage can be interpreted as a structure query, but according to the rules of S-JQL, it would return just the referenced issue without its children. To maintain backward compatibility, any structure query in Structure 2.4 that consists of a single basic constraint that references issues by their keys or IDs matches not only these issues, but all of their children as well.

That means that if you were using JQL of the form

```
issue in structure("My personal structure", TS-129)
```

then in Structure 2.4 this query will still return `TS-129` and all of its children at all levels (provided that `TS-129` is added to the structure.)

If this backward compatibility bites you (if, say, you need to check whether an issue is added to a structure), prepend the structure query with `issue in`:

```
issue in structure("My personal structure", "issue in TS-129")
```

This JQL will match only `TS-129` if it is in the structure.

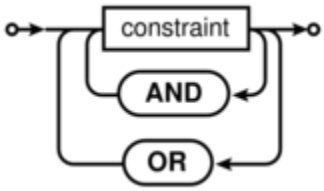
^ up to the list of S-JQL topics

Railroad diagrams

As a final piece of reference, here's the S-JQL syntax in the form of railroad diagrams.

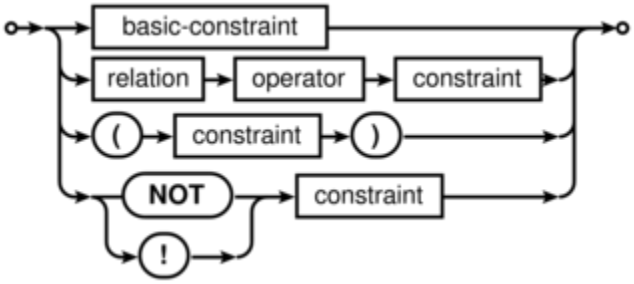
S-JQL keywords are case-insensitive, and all underscores in keywords are optional.

structure-query

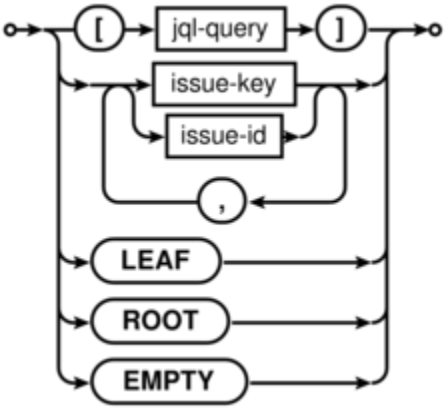


S-JQL admits using && and & in place of `AND`, as well as || and | in place of `OR`.

constraint

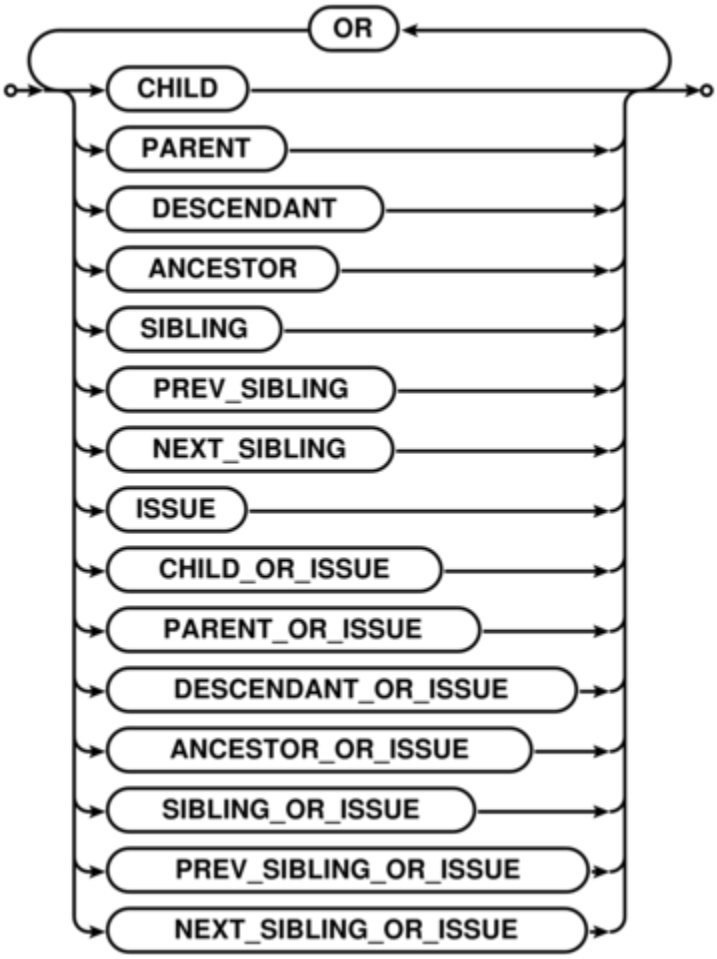


basic-constraint



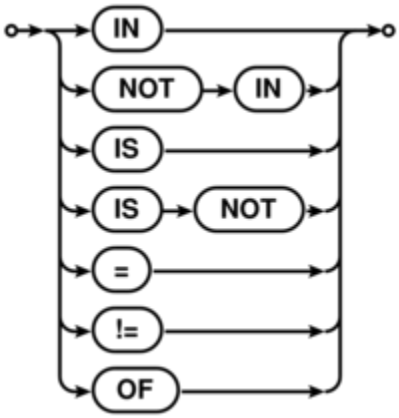
- `jql-query` is any valid JQL query subject to the quoting restrictions.
- `issue-key` is any valid JIRA issue key.
- `issue-id` is any valid JIRA issue ID.

relation



S-JQL admits using `||` and `|` in place of `OR`.

operator



^ up to the list of S-JQL topics