S-JQL Reference

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

> Parts of this article assume that you are familiar with Advanced Searching capability of JIRA.

List of Structured JQL topics:

Multiple instances of items

If there are multiple instances of an item in the structure, some of these instances might match the query, and some might not.

Consider the following structure:

```
TS-239
  TS-42
TS-123
  TS-239
```

Here, issue TS-239 is present two times — one at the root position, and another under another issue. Query `root` will match the first instance but not the second one.

This difference is visible when you are filtering in the Structure Widget (see Filter). However, structure() JQL function matches an issue if *at least one* of its instances in the structure matches the S-JQL query. In this example, `issue in structure(root)` will return `TS-239, TS-123.`

Constraints

Structure query consists of *constraints*. A constraint matches items 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 matches items that satisfy a condition — regardless of their relative positions to other items.

JQL constraint

JQL constraint matches all issues in the structure that satisfy a JQL query. To specify it, specify the JQL query enclosed in square brackets:

```
[status = Open]
```

`leaf` and `root`

This basic constraint matches items that are located at special positions within the structure.

```
leaf
```

```
root
```

The first constraint matches items at the bottom level of the hierarchy, i.e., items that do not have children (sub-items).
The second constraint matches items at the top level of the hierarchy, i.e., items that do not have a parent.

Specific issue

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.

You can specify a comma-separated list of issue keys:

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

One issue key:

```
TS-129
```

Issue ID (or a list of them):

```
19320
```

Function constraint (folder, item)

Functions in S-JQL play the same role as in JQL: it is an extension point, so any vendor can develop their own functions to match items in a custom way.

Structure comes bundled with a few functions: *folder* (matching all folders or folders by name) and *item* (matching all items of the specified type or items by name).

Syntax

A function constraint has a *name* and zero or more *arguments*, depending on the function you are using:

```
folder(Urgent)
```

In the example above, function name is *folder* and its argument is *Urgent.*

You can insert any amount of spaces around the name and arguments:

```
folder ( Urgent )
```

Multiple function arguments should be separated by commas:

```
item(Status, In Progress)
```

If an argument contains commas or parentheses, you need to enclose it in "double quotes" or 'single quotes':

```
item(Status, "Done, Sealed, and Delivered")
folder("NU (non-urgent) issues")
```

The former example matches Status items in structure that are named *Done, Sealed, and Delivered.* If this name wasn't enclosed in quotes, the query would mean that function *item* is given four arguments: *Status*, *Done*, *Sealed* and *and Delivered*.
The latter example matches folders named *NU (non-urgent) issues.* If quotes were not used, the query would be incorrect because the first closing parenthesis would be understood as the end of *folder*'s arguments.

If your argument contains quotes, you need to use another type of quotes to enclose it. Suppose that you need to match a version named *3.0, 3.0.1 "Armageddon"*:

```
item(version, '3.0, 3.0.1 "Armageddon"')
```

You can also escape the quotes using backslash (\). Suppose that the version is named *3.0 Beta 1 "Armageddon's Near":*

```
item(version, '3.0 Beta 1 "Armageddon\'s Near"')
```

If you need to use backslash character on its own, you can escape it with another backslash (\\). Suppose that you need to match a folder named *\ (backslash) and related characters*:

```
folder ('\\ (backslash) and related characters')
```

Note that if you don't need to enclose your argument in quotes, then you don't need to escape quotes or backslashes contained within it:

```
folder (Joe's)
folder ( \ )
```

Finally, if there's only one argument and the argument doesn't contain spaces (or is enclosed in quotes), you can omit the parentheses:

```
folder Urgent
folder "Not urgent"
```

folder()

This function matches folder items in the structure, optionally filtering them by name.

Without arguments, this function matches all folders:

```
folder()
```

With one argument, this function matches folders by name (that you see in the *Summary* column). A folder is matched if its name *starts with* the text specified in the first argument. Difference between capital and small letters is ignored.

For example, the following queries match folders named `My issues`, `Issues for Carol`, and `Non-issues`; and do not match folders named `Issuing` or `Issuance`:

```
folder issue
folder Issue
```

If you specify several words separated by spaces, `folder` will match only folders containing all of these words.

If you're familiar with how Text Search in structure works, then it's useful to think of this argument in the same way as of the simple query. The only difference is that `folder` doesn't recognize issue keys.

There's an advanced matching option for those who like to use *regular expressions.*

To tell `folder` that you are specifying a regular expression, enclose it in slashes (`/`):

```
folder /i.*ue/
```

If the argument starts with a slash but doesn't end with a slash, regular expression matching doesn't occur, and it's matched as a simple text. If you need to write a simple text search where a text starts and ends with a slash, escape the leading slash with a backslash (`\`):

```
folder \/???/
```

The query in the example above matches folder `/???/`.

Another advanced topic is how to query for the exact word (e.g., match `issue` but not `issues`).

This is called *strict searching*. Strict searching is turned on when the *search text* starts and ends with a double quote ("). Note, however, that quotes are stripped off from function arguments, since quoting is also used to allow specifying spaces or parentheses in the search text. Thus you'll need to enclose the search text in single quotes (').

```
folder '"issue"'
```

item()

This function matches items of the specified type in the structure, optionally filtering them by name. It is a generalization of `folder()` function to other item types.

The function takes two arguments: *item type* and *name* (optional). The second argument works in the same way as the argument for `folder()` function.

You can reference either standard item types (provided by Structure plugin) or item types provided by third-party plugins.

If you need to match items of all types, use asterisk (*). The following query finds all items that have the word “Infrastructure” in their Summary, regardless of their type:

```
item(*, Infrastructure)
```

Structure provides the following item types:

```
issue
project
version
project-component
issuetype
status
resolution
priority
label
user
group
date
cf-option
folder
memo
generator
loop-marker
sprint
missing
tempo-account (when Tempo Timesheets plugin is available)
sd-request-type (when Jira Service Desk plugin is available)
```

Structure.Pages plugin provides the following item types:

```
page
```

Item types provided by third-party plugins are specified similarly. Here's how `item()` function looks up item types:

1. It tries to interpret *type name* argument as referring to an item type provided by Structure and looks it up in the list above.
2. If not found, it looks at all item types provided by all plugins (including Structure itself) and checks if the type name *ends with* the specified text *as a word*. “As a word” means that `page` will match Confluence page item type, but `age` won't. More specifically, the considered word boundaries are hyphen (-), underscore (_) and colon (:).
3. It is an error to specify item type ambiguously, i.e. if there are two item types matching the description. The following forms of *item type* argument allow to specify item type more precisely.
    - Fully qualified item type name, e.g. `com.almworks.jira.structure:type-issue` or `com.almworks.structure.pages:type-confluence-page`.
      More generally, the form is `<plugin key>:<type name>`.
    - Shortened form of the fully qualified item type name, e.g., `structure:issue` or `pages:page`.
      More generally, the form is `<plugin key part>:<type name part>`.
      When `item()` function looks up item type for the argument, and the argument contains colon (:), the function first tries to interpret is as a fully quailified name. Only if nothing is found, it tries to interpret it as a shortened form.

Don't confuse “*matching items of some type*” and “*matching issues that have field value equal to that item*”. For example, `item(status, Open)` matches *status Open*, not *issues with status Open*. If you need the latter, use JQL constraint: `[status = Open]`.

Empty constraint

An empty constraint matching no items:

```
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 items that the original constraint doesn't:

```
not root
```

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

You can always enclose a constraint in parentheses to ease understanding. So, all items in the structure except issues `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 items that satisfy a condition. A relational constraint matches items *related to* items that satisfy a condition. *Related* corresponds to a relationship between positions of items 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 items that have `TS-129` as a child (sub-item).

Relational constraint has the form `relation operator subConstraint`. Here, `subConstraint` is a constraint on the relatives of items 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 items 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`: item is a child (sub-item) of another item in the structure.
- `parent`: item is a parent of another item in the structure.
- `descendant`: item is a descendant (sub- or sub-sub-...-item) of another item in the structure.
- `ancestor`: item is an ancestor (parent, parent-of-parent, or parent-of-parent-...-of-parent) of another item in the structure.
- `sibling`: item is a sibling of another item in the structure. Two items are considered siblings if they are under the same parent item.
- `prevSibling`: item is a previous (preceding) sibling of another item in the structure.
  item *A* is a preceding sibling of item *B* if it is a sibling of *B* and *A* is higher than *B* (*A* comes before *B*.)
- `nextSibling`: item is a next (following) sibling of another item in the structure.
  item *A* is a following sibling of item *B* if it is a sibling of *B* and *A* is lower than *B* (*A* comes after *B*.)
- `self` and `issue` are relations of an item (or an issue) to itself. Their role is explained later, in the `self` and `issue` relation section, because at first one has to learn how operators and sub-constraints work.
  There are also combinations of `issue` and `self` with all other relations, listed for completeness below:

`childOrSelf`	`childOrIssue`
`parentOrSelf`	`parentOrIssue`
`descendantOrSelf`	`descendantOrIssue`
`ancestorOrSelf`	`ancestorOrIssue`
`siblingOrSelf`	`siblingOrIssue`
`prevSiblingOrSelf`	`prevSiblingOrIssue`
`nextSiblingOrSelf`	`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 item.

   For example, consider

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

   Here, `relation` is `child`, so an item's relative in question is its child in the structure. Thus, an item 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 item is matched if it *is not related to* any item matching `subConstraint`.

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

   For example, consider

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

   An item matches if *no child is `TS-129` nor `TS-239`*. So, this constraint matches all items that either have no children or do not have any of these two items 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 item with no relatives is not matched by the transformed query.
>
> For example,
>
> ```
> child in (not (TS-129, TS-239))
> ```
>
> matches all items that have at least one child that is neither `TS-129` nor `TS-239`. That is, the only items that are not matched are leaves and those that have only `TS-129` or `TS-239` as children.

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

   For example, consider

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

   An item 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 items in the structure and put each of them, one by one, in column **item**. For each item, we take all of its relatives and put each of them, one by one, in column **relative**. Thus we get pairs of items. 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	item	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 items on the second level of the hierarchy. To select items 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 items. It is used as a sub-constraint to match items 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 items that have no children (equivalent of `leaf`)
`child is not empty`	matches all items that have at least one child (equivalent of `not leaf`)
`child of empty`	matches all items that are not children of other items (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 item if it is the only child of its parent.

^ up to the list of S-JQL topics

`self` and `issues` relations: adding sub-constraint matches to the result set

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

```
self in [status = Open]
```

is equivalent to

```
[status = Open]
```

Similarly,

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

is equivalent to

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

When combined with another relation, `self` allows to add the items 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 `descendantOrSelf`:

```
descendantOrSelf of TS-129
```

`issue` relation

`issue` is a special case of `self` relation that only matches issues. For instance, if on the top level of the structure you have folders and issues, and you want to hide all folders, you can write this:

```
descendantOrIssue of root
```

This query matches all top-level issues and all their sub-items.

^ 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]
```

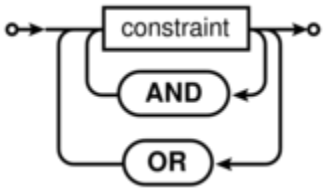
^ up to the list of S-JQL topics

Railroad diagrams

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

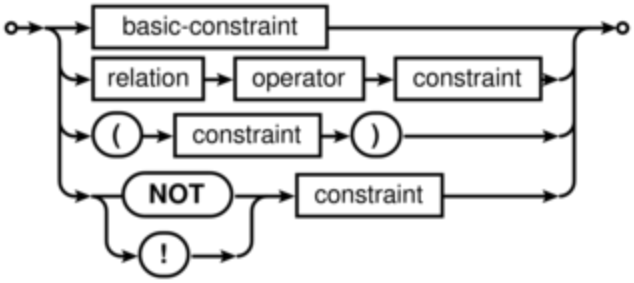
S-JQL keywords are not case-sensitive, and all underscores in keywords are optional.

structure-query

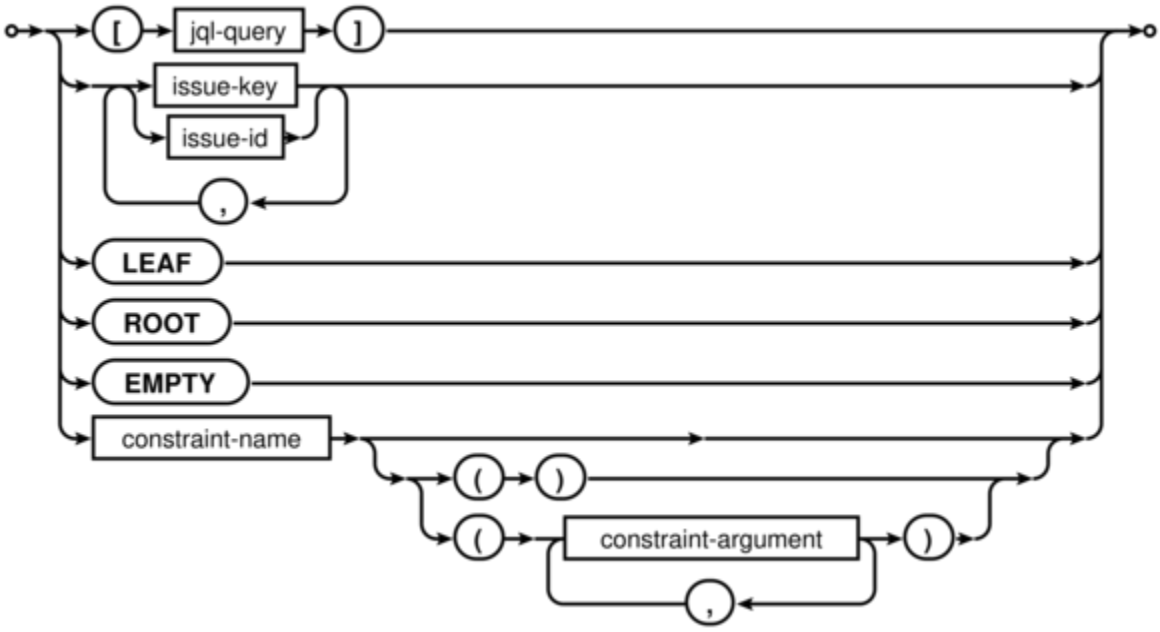


S-JQL admits using && and & in place of `AND`, as well as || and | in place of `OR`.

constraint



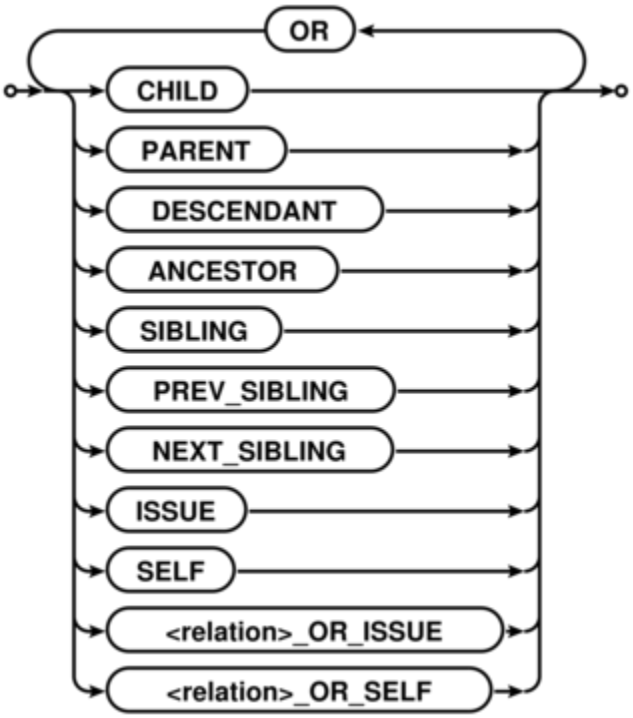
basic-constraint



- `jql-query` is any valid JQL query.
- `issue-key` is any valid JIRA issue key.
- `issue-id` is any valid JIRA issue ID.
- `constraint-name` is the name of the function constraint: either bundled with Structure (`folder`, `item`, or `row_id`) or provided by a Structure extension (plugin).
- `constraint-argument` is one of the following:
  - either a sequence of non-whitespace characters
  - or quoted text (inside "double quotes" or 'single quotes'), where quotes can be escaped via backslash: `\"`, `\'`; backslash itself can be escaped: `\\`.

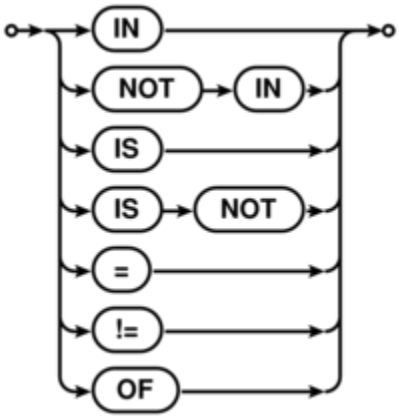
See also Function constraint - Syntax.

relation



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

operator



^ up to the list of S-JQL topics