Expr Language

Expr Language (pronounced like "expert" without the "t") is a simple language that lets you specify an "expression", or a formula, which is calculated for an issue or another item.

Expr can be used in:

- Formula Columns - the expression is calculated for each visible row in the displayed structure or query result.
- Automation - formulas can be used to filter a structure (Filter by Attribute), sort the structure (Sort by Attribute), or group issues (Group by Text Attribute) based on the results of a formula.
- Effectors - the results of a formula can be written to a Jira field using the Attribute to Issue Field Effector.

Expr fundamentals are easy to learn, and yet the language is powerful enough to address complex needs. The following guide will cover the basic requirements of the Expr language.

- Language Components
- Value Types
- Basic Constructs
    - Variables
    - Function Calls
    - Chained Function Calls
    - Numbers and Text Strings
    - Operations
    - Property Access
    - Conditional Expression
- Advanced Constructs
    - Local Variables
    - Aggregate Functions
    - User Functions
    - Embedded Queries (JQL and S-JQL)
    - Comments
- Additional Resources

For a more in-depth study, see our Expr Language Reference and Formula Reference Documentation.

You can view examples of Expr formulas in Sample Formulas or by adding bundled formulas to your structure. To see the formula, simply open the column options panel.

Language Components

An expression may contain one or more of the following:

- Variables - these are mapped to *attributes*, such as issue fields, custom progress, or a value from another column.
- Arithmetic and logical operations - add, subtract, multiply, divide, or compare items.
- Numbers
- Text strings
- Function calls - apply specific calculations to the provided arguments and return a result to be used in the expression.
- Property access - get the value of a particular item property-, such as an author of a work log entry.
- Conditional ("IF") expressions - calculate different results based on a specified set of criteria.

There are also more advanced constructs:

- Aggregate Functions - calculate an aggregate (such as the sum or average) of an expression's values calculated for multiple items in the structure.
- Local Variables - introduce a value and reuse it multiple times in the formula.
- User Functions - define a function, or a functional expression, to be reused in the formula or applied to an array of values.
- JQL and S-JQL queries inside a formula - condition the results based on whether an item matches a query.
- Comments - document larger formulas.

Value Types

With Expr you can build formulas that operate on:

- Basic values – numbers and text, which are either a part of the formula, or read from a simple attribute or Jira field, such as an issue's Summary or Story Points.
- Items – values representing a particular object, such as User, Issue, Worklog, Status and others, and typically read from the corresponding Jira field.
- Array values – a sequence of values, allowing you to represent multi-value fields such as Fix Versions, or multiple Entities, in the case of the `work logs` attribute.
- User function values – representing a piece of formula that is typically applied to each element in an array, for example, for filtering.
- Key-value maps – in rare occasions, these values are produced by the supplied functions like GROUP.
- Undefined – a special value that means "nothing" or "no value".
- Error values – produced if there was a problem calculating a formula.

Normally, you don't need to worry about the value types when writing a formula. The language engine will try to make sense of the formula and convert the values as needed. For more complex formulas, or if something doesn't work as expected, see Expr Function Reference for the expected types for each function.

Basic Constructs

Variables

Variables are user-defined names, which represent *attributes*, such as:

- Jira issue fields
- Calculated attributes like Progress
- Structure-specific attributes like Item type
- Attributes provided by other Jira apps
- Another formula
- Values from another Structure column

Naming Variables

Variables can contain letters (English only), numbers or underscore ("_") characters. Variables cannot contain spaces, and the first character must be a letter or an underscore.

Examples:

- `priority`
- `sprintName`
- `remaining_estimate`
- `abc11`

As you write your formula, Structure attempts to map your variables to well-known attributes. For example, the "remaining_estimate" variable above will automatically be mapped to the Remaining Estimate field. See Mapping Variables for more information.

Variable names are case-insensitive. `Priority`, `priority` and `pRiOrItY` will all refer to the **same** variable.

Function Calls

A function calculates a value based on its arguments and, sometimes, some external aspect. A function call is written as the function name, followed by parentheses, which may or may not contain arguments.

Examples:

- `SUM(-original_estimate, remaining_estimate, time_spent)`
- `CASE(priority, 'High*', 5, 1)`
- `TODAY()`

Function names are case-insensitive. You can write `TODAY()` or `Today()`.

There are 100+ standard functions available with Structure – see Expr Function Reference for a complete list.

Function arguments may be separated by comma (,) or semicolon (;). But in every function call within a formula, you need to use either all commas or all semicolons.

Chained Function Calls

The chained notation allows you to easily apply a sequence of functions to a value, simply by listing each function one after the other, separated by a (.) dot.

- Standard notation: `F3(F2(F1(x)))`
- Chain notation: `x.F1().F2().F3()`

When you use the chain notation, the value that comes before the dot becomes the first argument for the function. If the function takes multiple arguments, the rest of the arguments must be written in parentheses.

For example:

```
created.FORMAT_DATETIME("yyyy").CONCAT(" year issue")
```

In this example, FORMAT_DATETIME takes the date value in "created" and formats it based on the argument in parenthesis ("yyyy"). CONCAT takes the result from FORMAT_DATETIME and joins it with " year issue".

Numbers and Text Strings

Numbers

Formulas support whole numbers, decimals, or fractions. Commas, spaces, locale-specific, percentage, currency or scientific formats are not supported.

Recognized as a number	Not recognized as a number
`0`	`0,0`
`1000`	`1,000`
`1234567890123456`	`1 100 025`
`11.25`	`1.234e+04`
`.111`	`($100)`

You can write a number that is written with a locale-specific decimal and thousands separator as a text value, and it will be automatically converted to a number if needed. For example:

- `"1 122,25" * 2  2244.5`

Text Strings

Text strings are a sequence of characters enclosed either in single (') or double quotes ("). Examples:

- `'a text in single quotes may contain " (a double quote)'`
- `"a text in double quotes may contain ' (a single quote)"`
- `""`

Everything within a text string is retained exactly when the expression is evaluated or displayed, except for the following:

- A sequence of two backslashes (`\\`) is converted to a single backslash (`\`).
- A sequence of a backslash and a single quote (`\'`) is converted to a single quote character (`'`) for text values enclosed in single quotes.
- A sequence of a backslash and a double quote (`\"`) is converted to a double quote character (`"`) for the text values enclosed in double quotes.

Text Snippets

Text Snippets allow you to generate strings using variables and expressions. This is particularly helpful in formulas that utilize wiki markup.

When using text snippets:

- The snippet should be enclosed with """ (three double quotes, at the beginning and at the end)
- The expression portion of the snippet is introduced using the '$' symbol and should be enclosed in braces { }

```
""" $var1 + $var2 = ${var1 + var2} """

""" this $glass is half-${IF optimist: 'full' ELSE: 'empty'} """
```

Operations

Expr provides basic arithmetic operations, comparisons, text operations and logical operations.

Operations	Comments
`+ - * /`	Basic operators. When used, the value is converted to a number. Follows the general precedence rules for arithmetic, so (2 + 3 * 4 = 14).
`= !=`	Equality and non-equality: if either part of the comparison is a number, the other part is also converted into a number. If both values are texts, then text comparison is used. Text comparison ignores leading and trailing whitespace and is case-insensitive (according to Jira's system locale).

`< <= > >=`	Numerical comparisons. When used, both values are converted to numbers.
`AND`, `OR`, `NOT`	Logical operations.
`CONCAT`	An operation that joins together two text strings. Works similar to the function of the same name: `a CONCAT b` is the same as `CONCAT (a, b)`.
`( )`	Parentheses can be used to group the results of operations prior to passing them to other operations.

Order of Operations

When several types of operations are used, they are done in the following order:

1. Arithmetic operations
2. Text operations (CONCAT)
3. Comparison operations
4. Logical operations.

For detailed specification, see Expr Language Reference.

Property Access

Formulas can get the value of an item's property using the following notation: `object.property`

```
fixVersion.releaseDate  //returns the release date for the fixVersion
```

You can also string multiple property calls together:

```
project.lead.emailAddress  //returns the email address of the lead for the project
```

For a complete list of supported properties, see Item Property Reference.

Conditional Expression

Simple "IF" expressions can be declared using the IF() function, but for more elaborate IF cases, with multiple conditions and/or requiring an ELSE option, a conditional expression can be used.

```
WITH total = x + y:
  IF total > 0:
     x / total
  ELSE : error
```

Note: the ":" after "ELSE" is optional – in the example above, we've included it for readability.

Advanced Constructs

Local Variables

Local variables are helpful when an expression needs to be used in the same formula several times. For example:

```
IF time_spent + remaining_estimate > 0 :
   time_spent / (time_spent + remaining_estimate)
```

You can see that in this formula we are using "`time_spent + remaining_estimate`" twice – once when we check that it's not zero (so we don't divide by zero) and again when we divide by it.

Instead of repeating the expression every time, we can rewrite this formula using the **`WITH`** construct:

```
WITH total_time = time_spent + remaining_estimate :
  IF total_time > 0 :
     time_spent / total_time
```

You can define multiple local variables in succession. You can also use previously defined local variables when defining additional local variables. For example:

```
WITH total_time = time_spent + remaining_estimate :
WITH progress = (IF total_time > 0 : time_spent / total_time) :
  IF(progress > 0.5, "Great Progress!", progress > 0.2, "Good Progress", "Needs Progress!")
```

Note the position of the colon (":") – it must be present where each local variable definition ends.

Aggregate Functions

An aggregate function calculates some aggregate value (like sum or minimum) based on the values in a number of rows, typically for all sub-issues. Aggregate functions are written very similar to standard functions, except they use curly braces: `SUM{x}`.

Examples:

- `SUM { remaining_estimate + time_spent }` – calculates the total effort (estimated and actual) for the issue and all its sub-issues.
- `MAX { resolved_date - created_date }` – calculates the maximum time it took to resolve an issue, among the issue and its sub-issues.

They can also contain **modifiers**, which influence how the aggregation works:

- `SUM#all { business_value }` – this will force the function to include values from all duplicate items in the total. (By default, duplicates are ignored.)

See Aggregate Function Reference for a complete list of available aggregate functions and modifiers.

Any local variables used inside an aggregate function must also be declared inside the function - within the `{ }`.

User Functions

A User Function allows you to define a locally-used function within a formula. User functions can be defined in a similar manner as local variables:

```
WITH square(x) = x * x :
  square(impactField) / square(storyPoints)
```

In this example, the user function is given a name ("square") and then used to perform the same calculation on multiple fields. To learn more, see the language reference.

User Functions for Arrays - using the "$" character

When you need to perform an operation on each element in an array, you can use a user function such as the one above, or simplify it using “$” to indicate each element in the array.

```
worklogs.FILTER($.author = ME())
```

In this example, the "$" tells Structure to apply "author = ME()" to each element in worklogs - if the author is the current user, it returns true and that worklog will be included in the FILTER results.

This method becomes very powerful when you combine multiple user functions together. To learn more, see the language reference.

Embedded Queries (JQL and S-JQL)

You can embed JQL and Structured JQL queries inside an Expr formula, using a construct similar to Aggregate Functions. The result will be a boolean value:

- `1` (true) if the current row matches the query
- `0` (false) otherwise

For example:

```
// Collect total story points from all sub-issues assigned to members of Team2 group, unless the stories are
under folder "Special"
SUM {
    IF JQL { assignee in membersOf("Team2") } :
    IF NOT SJQL { descendant of folder("Special") } :
      storyPoints
}
```

Since JQL is a Jira-based query, it will work only on issues; the result will be `0` on other types of items. S-JQL can be used for more complex queries applicable to the whole structure.

Comments

Comments are helpful when you have a large formula or when a reader might need explanations of what is being calculated. It's a good idea to add comments wherever the formula is not trivial.

- To add a single line of comment, begin the comment with `//`
- To add multiple lines of comment, start the comment with `/*` and end the comment with `*/`

Example:

```
// This is a single-line comment.

/* This is a multi-line comment.
   It can be useful for longer explanations. */
```

Additional Resources

- Sample Formulas
- Wiki Markup in Formula Columns
- Formula Reference Documentation