Jira-Select Documentation

Release 3.2.0

Adam Coddington

Aug 16, 2023

CONTENTS:

Jira-select is a command-line tool and library that helps you generate the useful insights you need out of Jira.

Jira has its own query language but there are many limitations around what JQL is capable of. Some data is returned in arcane formats (e.g. sprint names are returned as a string looking something like `com.atlassian.greenhopper.service.sprint.Sprint@14b1c359[id=436...]`), data cannot be grouped (there's nothing like SQL's *GROUP BY* statement), and because of that lack of grouping, there are no aggregation functions – no *SUM*-ing story points or estimates per-assignee for you. And if you want to write a custom function for processing a field, well, I'm not even sure where you'd begin. Jira-select makes those things easy.

If you've ever found yourself held back by the limitations of Jira's built-in query language, this tool may make your life easier. Using Jira-select you can perform a wide variety of SQL-like query operations including grouping, aggregation, custom functions, and more.

CHAPTER

ONE

QUICKSTART

First, install this package:

```
pip install jira-select
```

Next you need to configure *jira-select* to connect to your jira instance:

```
jira-select configure
```

Follow the displayed instructions, then, you can open up your shell:

```
jira-select shell
```

From here, you can type out a jira-select query (See *Query Format* for details). The format is inspired by SQL, but isn't quite the same. The following example will return to you a table showing you which issues are assigned to you.

```
select:
- key
- summary
from: issues
where:
- assignee = "your-email@somecompany.com"
- resolution is null
```

The editor uses *vi* bindings by default; so once you're ready to submit your query, press *Esc* followed by *Enter* and after a short wait (watch the progressbars), you'll be shown your results. Press *q* to exit your results.

See the built-in help (*–help*) for more options.

CHAPTER

TWO

QUERY FORMAT

Jira-select queries are written in a YAML format, but using section names inspired by SQL.

Here's a simple example that will return all Jira issues assigned to you:

```
select:
- Issue Key: key
- Issue Summary: summary
from: issues
where:
- assignee = "your-email@your-company.net"
```

Here's a query that uses many more of the possible sections, but know that in real life, you're very unlikely to use them all at once:

```
select:
  My Assignee: assignee
  Key Length: len(key)
from: issues
expand:
- changelog
where:
- project = "MYPROJECT"
order_by:
- created
filter:
- customfield10010 == 140
group_by:
- assignee
having:
- len(key) > 5
sort_by:
- len(key) desc
limit: 100
cap: 10
cache: 86400
```

Below, we'll go over what each of these sections are for in detail.

2.1 Query Structure Overview

Table 1: Jira-select Query Sections

	Use	Type	Evaluated where?	Can use custom functions?	Can use query parameters?
`select`	Defines data to return	List[Expression]	Local	Yes	Yes
`from`	Defines data source	str	*n/a*	*n/a*	**No**
`where`	Remote filtering of results	List[JqlString] (for 'issues') *or* Dict[str, Any] (for 'boards' and 'sprints')	Remote	**No**	**Yes**
`order_by`	Remote ordering of results	List[JqlString]	Remote	**No**	**No**
`filter`	Local pre-grouping filtering of results	List[Expression]	Local	Yes	Yes
`group_by`	Grouping returned rows together	List[Expression]	Local	Yes	Yes
`having`	Local post-grouping filtering of results	List[Expression]	Local	Yes	Yes
`sort_by`	Local sorting of results	List[Expression]	Local	Yes	Yes
`limit`	Remote limiting of the count of results	int	Remote	*n/a*	*n/a*
`expand`	Defines Jira issue field expansions	List[str]	Remote	*n/a*	*n/a*
`cap`	Post-having/group_by limiting of results	int	Local	*n/a*	*n/a*
`cache`	Defines how long to cache Jira results	int	Local	*n/a*	*n/a*

2.1.1 What is a `JqlString`

A `JqlString` is standard Jira JQL. You can find more information about writing JQL in Jira's JQL documentation.

2.1.2 What is an `Expression`

An `Expression` is an expression evaluated by Jira-select. Expressions have access to all functions documented in *Query Functions*. The variables available for use in your expressions can be determined by using `jira-select schema [issues|boards|sprints]`.

Expressions are (with one caveat) valid Python expressions. The single caveat is that you can use curly braces to quote field names. These curly-brace-quoted fields will be replaced with the actual Jira field name before processing the expression in Python.

For example; if you have a custom field named `customfield10010` that has a human-readable name of `Story Points`, you can create an expression like:

```
{Story Points} >= 5
```

this expression will be transformed into:

```
customfield10010 >= 5
```

before evaluating the expression in Python.

2.2 Ubiquitous

2.2.1 select

This section defines what data you would like to include in your report. It should be a dictionary mapping the column name with the expression you would like to display in that column. This section *can* use custom functions (see *Query Functions* for options).

For example:

```
select:
  My Field Name: somefunction(my_field)
```

Note: This section supports a handful of formats in addition to the one discussed here that you may find in some documentation or in other examples including:

You can specify columns as a list:

```
select:
- somefunction(my_field) as "My Field Name"
```

You can specify a single column as a string:

```
select: somefunction(my_field) as "My Field Name"
```

The above formats will be supported for the foreseeable future, but the dictionary-based format discussed outside this box is the preferred format for writing queries.

As a shorthand, if you do not provide a value for your dictionary entry, the dictionary entry's name will be used as the expression for your column:

```
select:
  issuetype:
  key:
  summary:
from: issues
```

In the above example, the fields `issuetype`, `key`, and `summary` will be displayed in columns matching their field name.

If you would like to return *all* fields values, use the expression * as a stirng value to your *select* statement:

```
select: "*"
from: issues
```

Important: Due to yaml parsing rules, the * expression must be quoted.

2.2.2 from

This section defines what you would like to query. The value should be a string.

There are two query sources currently implemented:

- `issues`: Searches Jira issues.
- `boards`: Searches Jira boards.
- `sprints`: Searches Jira sprints.

2.3 Common

2.3.1 where

The `where` section varies depending upon what kind of data source you are querying from.

issues

This section is where you enter the JQL for your query. This should be provided as a list of strings; these strings will be `AND`-ed together to generate the query sent to Jira.

```
where:
- assignee = 'me@adamcoddington.net'
```

You *cannot* use custom functions in this section given that it is evaluated on your Jira server instead of locally.

boards

You can provide key-value pairs to limit the returned boards; the following parameters are allowed:

- `type`: The board type. Known values include 'scrum', 'kanban', and 'simple'.
- `name`: The board name. Returned boards mustinclude the string you provided somewhere in their name.

```
where:
  name: 'My Board'
```

sprints

You can provide key-value pairs to limit the returned boards; the following parameters are allowed:

- `state`: The sprint state. Known values include 'future', 'active', or 'closed'.
- `board_type`: The board type of the board to which this sprint belongs. Known values include 'scrum', 'kanban', and 'simple'.
- `board_name`: The board name of the board to which this sprint belongs. Returned boards mustinclude the string you provided somewhere in their name.

```
where:
  state: 'active'
```

Note: This type of query is slow due to the way Jira's API exposes this type of record. There is no endpoint allowing us to list sprints directly. Instead, we must collect a list of sprints by requesting a list of sprints for each board.

You can improve performance substantially by using the `board_type` or `board_name` parameters to limit the number of boards we will need to request sprints for.

2.3.2 order_by

This section is where you enter your JQL ordeirng instructions and should be a list of strings.

You *cannot* use custom functions in this section given that it is evaluated on your Jira server instead of locally.

2.3.3 group_by

This section is where you can define how you would like your rows to be grouped. This behaves similarly to SQL's `GROUP BY` statement in that rows sharing the same result in your `group_by` expression will be grouped togehter.

For example; to count the number of issues by type that are assigned to you you could run the following query:

```
select:
  Issue Type: issuetype
  Key Length: len(key)
from: issues
where:
- assignee = "your-email@your-company.net"
group_by:
- issuetype
```

Note: When executing an SQL query that uses a `GROUP BY` statement, you will always see just a single value for each column even if that column represents multiple rows' values.

Unlike standard SQL, in Jira-select column values will always contain arrays of values when your column definition does not use a value entered in your `group_by` section. If you are surprised about a particular field showing an array holding values that are all the same, try adding that column to your `group_by` statement, too.

If you would like to perform an aggregation across all returned values, you can provide `True` in your `group_by` statement. This works because, for every row, `True` will evaluate to the same result causing all rows to be grouped together:

```
select:
  Key Length: len(key)
from: issues
where:
- assignee = "your-email@your-company.net"
group_by:
- True
```

You **can** use custom functions in this section.

2.3.4 having

This section is where you can provide filtering instructions that Jql cannot handle because they either require local functions or operate on grouped data.

You **can** use custom functions in this section.

2.3.5 calculate

Perhaps you have an expression you'd like to calculate once and use multiple times across your query (e.g. multiple times across `select` columns, or in both `select` and `filter` at the same time). You can use the `calculate` section for performing those calculations once and then referencing their result in other expressions; for example:

```
select:
  Hours in Progress: round(in_progress_seconds / 3600)
calculate:
  in_progress_seconds: interval_size(interval_matching(issue, status="In Progress") &␣
↪interval_business_hours(parse_date(created))).total_seconds() / 28800
from: issues
filter:
- in_progress_seconds > 60
expand:
- changelog
```

The above example will calculate the total amount of time issues were in progress in hours while excluding results where they were in progress for fewer than sixty seconds.

2.3.6 sort_by

This section is where you can provide sorting instructions that Jql cannot handle because they either require local functions or operate on grouped data.

You **can** use custom functions in this section.

2.3.7 limit

This sets a limit on how many rows will be returned from Jira. See *Query Lifecycle* to understand where this fits in the query lifecycle.

If you would like to limit the count of rows *after* `group_by` and `having` have reduced the count of rows, use `cap` instead.

Note: `limit` is handled by Jira itself, so if you would like to instead limit the number of rows returned after `having` and `grouping` expressions have reduced the row count, use `cap` instead.

2.3.8 cache

This will cache the results returned by Jira for up to the specified number of seconds. This can be very helpful if you are iterating on changes to your `group_by` or `having` sections in that you can make minor changes and avoid the slow process of requesting records from jira after every change.

Note that the cache parameter can be in one of two forms:

```
cache: 86400
```

In this case, we will cache the results for up to 86400 seconds and will also accept an already-stored cached value that is up to that number of seconds old.

```
cache: [300, 86400]
```

In this case, we will cache the results for up to 86400 seconds, but will only accept a cached value that is 300 seconds old or newer.

You can also pass `null` as the second parameter to allow reading from the cache, but disallow writing a new cached value, or pass `null` as the first parameter to disallow using an existing cache, but allowing storing a new value.

Note that to take full advantage of caching, you may also want to use the `filter` feature described below. Using it can let you take better advantage of your cached values.

2.4 Unusual

2.4.1 expand

Jira has a concept of "field expansion", and although by default Jira-select will fetch "all" data, that won't actually return quite all of the data. You can find more information about what data this will return by reading the Jira documentation covering "Search for issues using JQL (GET)".

2.4.2 filter

In most cases, using `where` (pre-grouping/having, processed by Jira) and `having` (post-grouping) are sufficient. But there are scenarios where you might want to filter rows between these two steps. For example:

- Jql doesn't provide the functionality you need for filtering your resultset, but you'll be using a `group_by` statement, too, and thus can't just use `having`; because by that point, the field you need to filter on will have been grouped with others.
- You are using a long cache interval to quickly iterate on your query and do not want to have to update your `where` expression since changing that will cause us to not use the cached results.

In these cases, you can enter the same sorts of expressions you'd use in a `having` statement in your `filter` statement as a sort of local-side equivalent of `where`.

You **can** use custom functions in this section.

2.4.3 cap

This sets a limit on how many rows will be returned, but unlike `limit` is evaluated locally.

This can be used if you want your `having` or `group_by` statements to have access to as much data as possible (and thus do not want to use `limit` to reduce the number of rows returned from Jira), but still want to limit the number of rows in your final document.

CHAPTER

THREE

QUERY LIFECYCLE

Jira-select queries are evaluated in many steps across two phases:

- Remote
  - JQL Query (`where`, `order_by`, and `limit`)
- Local
  - Calculating (`calculate`)
  - Filtering (`filter`)
  - Grouping (`group_by`)
  - Filtering (`having`)
  - Sorting (`sort_by`)
  - Capping count of results (`cap`)
  - Rendering results (`select`)

The steps in the "Remote" section are accomplished entirely by Jira and thus are limited to the capabilities of JQL. The result of this part of the query processor can be cached by using the `cache` query parameter.

The steps in the "Local" section are accomplished on your local machine by Jira-select, and thus can use custom functions.

CHAPTER

FOUR

QUERY FUNCTIONS

Jira-select provides a long list of functions out-of-the-box, and you can add your own if these are not enough.

4.1 Jira

get_issue(*ticket_number: str*) → jira.resources.Issue

Fetch a Jira issue by its issue key (e.g. `MYPROJECT-1045`).

This will return a `jira.resources.Issue` object; you can access most fields via its `fields` property, eg:

```
get_issue(field_holding_issue_key).fields.summary
```

get_issue_snapshot_on_date(*issue: jira.resources.Issue*) → jira_select.types.IssueSnapshot:

Reconstruct the state of an issue at a particular point in time using the issue's `changelog`.

You will want to pass the literal value `issue` as the first parameter of this function. Jira-select provides the `jira.resources.Issue` object itself under that name, and this function will use both that object and the changes recorded in the `changelog` field for getting an understanding of what the issue looked liked at a particular point in time.

This function requires that you set the query `expand` option such that it includes `changelog` for this to work correctly – if you do not do that, this function will fail.

```
select:
  snapshot: get_issue_snapshot_on_date(issue, parse_datetime('2022-01-01'))
from: issues
expand:
- changelog
```

The returned snapshot is *not* a `jira.resources.Issue` object, but instead a `jira_select.types.IssueSnapshot` object due to limitations around what kinds of data can be gathered from the snapshot information. The most important difference between a `jira_select.types.IssueSnapshot` and a `jira.resources.Issue` object is that the `jira_select.types.IssueSnapshot` object is a simple `dict[str, str]` object in which the value of the `dict` entries is always the `str`-ified field value.

sprint_name(*sprint_blob: str*) → Optional[str]

Shortcut for returning the name of a sprint via its ID. Equivalent to calling `sprint_details(sprint_blob).name`.

sprint_details(*sprint_blob: str*) → Optional[jira_select.functions.sprint_details.SprintInfo]

Returns an object representing the passed-in sprint blob.

Jira returns sprint information on an issue via strings looking something like:

```
com.atlassian.greenhopper.service.sprint.Sprint@14b1c359[id=436,rapidViewId=153,
↪state=CLOSED,name=MySprint,goal=Beep Boop,startDate=2020-03-09T21:53:07.264Z,
↪endDate=2020-03-23T20:53:00.000Z,completeDate=2020-03-23T21:08:29.391Z,
↪sequence=436]
```

This function will extract the information found in the above string into a `jira_select.functions.sprint_details.SprintInfo` object allowing you to access each of the following properties:

- `id`: Sprint ID number
- `state`: Sprint state
- `name`: Sprint name
- `startDate`: Sprint starting date (as datetime)
- `endDate`: Sprint ending date (as datetime)
- `completeDate`: Sprint completion date (as datetime)

get_sprint_by_id(*id: int*) → Optional[jira.resources.Sprint]

This function will request the information for the sprint specified by `id` from your Jira server and return it as a `jira.resources.Sprint` object.

get_sprint_by_name(*board_name_or_id: Union[str, int]*, *sprint_name: str*) → Optional[jira.resources.Sprint]

This function will request the information for the sprint matching the specified name and belonging to the specified board. This will be returned as a `jira.resources.Sprint` resource.

field_by_name(*row: Any*, *display_name: str*) → Optional[str]

Returns value for field having the specified display name.

Note: You probably do not nee to use this function. We provide another, simpler, method for getting the value of a field by its human-readable name– just place the human-readable name in between curly braces in your query expression; eg:

```
select
  Story Points: "{Story Points}"
from: issues
```

Note: You will almost certainly want to provide the parameter named `issue` as the first argument to this function. Jira-select provides the raw row data to functions under this variable name.

In Jira, custom fields are usually named something like `customfield_10024` which is, for most people, somewhat difficult to remember. You can use this function to get the field value for a field by its display name instead of its ID.

Example:

```
select
  - field_by_name(issue, "Story Points") as "Story Points"
from: issues
```

estimate_to_days(*estimate_string: str*, *day_hour_count=8*) → Optional[float]

Converts a string estimation (typically stored in `timetracking.originalEstimate`) into an integer count of days.

The `timetracking.originalEstimate` field contains values like `1d 2h 3m`; using this function will transform such a value into `1.25625`.

flatten_changelog(*changelog*) → List[jira_select.functions.flatten_changelog.ChangelogEntry]

Converts changelog structure from your returned Jira issue into a flattened list of `jira_select.functions.flatten_changelog.ChangelogEntry` instances.

Note: You must use the `expand` option of `changelog` for Jira to return to you changelog information in your query; eg:

```
select:
  changelog: flatten_changelog(changelog)
from: issues
expand:
- changelog
```

If you do not provide the necessary `expand` option, this function will raise an error.

Every member of the returned list has the following properties:

- `author` (str): Author of the change
- `created` (datetime.datetime): When the change took place
- `id` (int): The ID of the changelog entry
- `field` (str): The name of the field that was changed
- `fieldtype` (str): The type of the field that was changed
- `fromValue` (Optional[Any]): The original value of the field. Note that the original Jira field name for this is `from`.
- `fromString` (Optional[str]): The original value of the field as a string.
- `toValue` (Optional[Any]): The final value of the field. Note that the original Jira field name for this is ``to`.
- `toString` (Optional[str]): The final value of the field as a string.

This returned list of records can be filtered with `simple_filter` to either find particular entries or filter out rows that do not have an entry having particular characteristics.

get_linked_issue_keys(*issue: jira.resources.Issue*, *link_type: str | None = None*) → list[str]:

Return a list of issue keys that are related to the relevant issue via the specified relation type (e.g. `causes`, `is associated with`, etc.).

You will want to pass the literal value `issuelinks` as the first parameter of this function. This will provide this function with the list of issuelinks your issue has.

If `link_type` is unspecified, all related issue keys will be returned.

For example, to find the keys for all issues that were caused by a particular issue, you could run the following query:

```
select:
  caused_bugs: get_linked_issue_keys(issuelinks, 'causes')
from: issues
where:
- type = 'Bug'
```

4.2 Subquery

subquery(*subquery_name*, ***params*) → Any:

Runs a subquery by name with the provided parameters.

For example: you can get the time intervals during which an issue and its associated subtasks were in the "In Progress" status with a query like so:

```
select:
  self_and_child_intervals_in_progress: interval_matching(issue, status="In Progress
↪") | union(subquery("children", key=issue.key))
from: issues
subqueries:
   children:
      select:
        in_progress_intervals: interval_matching(issue, status='In Progress')
      from: issues
      where:
      - parent = "{params.key}"
      expand:
      - changelog
expand:
- changelog
```

Your specified `**params` will become available to the subquery via `{params.*}`; in the above example, `{params.key}` will be set to the value of the outer query's `issue.key`.

Unless specifically specified, a subquery will use the same cache settings as the parent query.

> **Warning:** If you would like your subquery's cache to be effective, only pass simple values in `**params`.
>
> The string representation of an object is used for calculating cache keys, and many objects include information in their default string representations that vary between instantiations. If things like, for example, the memory address of an object appears in its string representation, the cache key will never match, and the cached value will not be used.
>
> A common way that this problem might occur is if you were to pass the entire `issue` object to the subquery. Instead of passing the entire `issue` object to the subquery, pass simple values from it as shown in the example above.

4.3 Time Analysis

interval_matching(*issue*, ***query_params: dict[str, Any]*) → portion.Interval

See *simple_filter function* for information about how to specify `query_params`.

Will return an interval covering segments in which the provided issue matches the conditions specified by `query_params`.

> **Note:** Contrary to what you might guess, a single *portion.Interval* object can represnt multiple ranges of time.

Note that *portion.Interval* objects can be used with logical operations like |, &, and -.

interval_size(*interval: portion.Interval*) → datetime.timedelta

For a provided interval, return the total amount of time that the interval's segments span.

interval_business_hours(*min_date: datetime.date | None = None*, *max_date: datetime.date | None = None*, *start_hour: int = 9*, *end_hour: int = 17*, *timezone_name: str | None = None*, *work_days: Iterable[int] = (1, 2, 3, 4, 5)*) → portion.Interval:

Returns an interval having segments that correspond with the "business hours" specified by your paramters.

This is particularly useful when used in conjunction with *interval_matching* and *interval_size* above for determining the amount of time an issue was actively in a particular state, for example:

```
select:
  total_time_in_progress: interval_size(interval_matching(issue, status="In Progress
↪") & interval_business_hours(parse_date(created)))
from: issues
```

This will find all segments of time during which the relevant issue was in the "In Progress" status during business hours, then return the amount of time that those segments spanned.

Note: A naive implementation of this sort of time analysis might use actual, raw clock time, but consider the following two situations:

- MYPROJECT-01 moves from "To Do" into "In Progress" at 4:55PM, just five minutes before the end of the day, then the next day moves from "In Progress" into "Done" at 9:05AM, five minutes after the beginning of the next day.
- MYPROJECT-02 moves from "To Do" into "In Progress" at 10:00AM and in the same day from "In Progress" into "Done" at 4:00PM.

Clearly, MYPROJECT-02 was being "worked on" for more time than MYPROJECT-01, but let's see how various algorithms might measure that time.

If we use raw clock time:

- MYPROJECT-01: 16.2h (81 times more than the actual working time)
- MYPROJECT-02: 6h

If we only measure time happening between 9A and 5P:

- MYPROJECT-01: 0.2h (the actual working time)
- MYPROJECT-02: 6h (the actual working time)

Of course, this does introduce one inaccuracy that may, depending on how predicable your team's working hours are, make this behavior undesirable: time spent working on an issue outside of business hours isn't counted. Typically, though, the amount of time an issue might be worked on outside those hours will be much smaller than the amount of excess time using raw clock time directly would count.

- `min_date`: The minimum date to add the business hours of to your interval. By default, 365 days before now.
- `max_date`: The (exclusive) maximum date to add the business hours of to your interval. By default: tomorrow.
- `start_hour`: The work day starting hour. Defaults to 9 (i.e. 9 AM)
- `end_hour`: The work day ending hour. Defaults to 17 (i.e 5 PM)
- `timezone_name`: The timezone to interpret `start_hour` and `end_hour` in.

- **work_days: The days of the week to count as work days; 0 = Sunday,**
  1 = Monday… 6 = Saturday.

4.4 Data Traversal

extract(*field: Iterable[Any]*, *dotpath: str*) → Iterable[Any]

For every member of `field`, walk through dictionary keys or object attributes described by `dotpath` and return all non-null results as an array.

Note: Although this will work, it is not necessary to use this for traversing into properties of grouped rows. If your selected field is an object having a value you'd like to select, you can simply use dotpath traversal to reach the value you'd like.

This function works for both dictionaries and objects.

flatten_list(*field: List[List[Any]]*) → List[Any]

For a list containing a lists of items, create a single list of items from the internal lists.

The above is a little bit difficult to read, but in principle what this function does is convert values like:

```
[[1, 2, 3], [4, 5, 6]]
```

into a single list of the shape:

```
[1, 2, 3, 4, 5, 6]
```

4.5 Dates

now(***replacements*) → datetime.datetime

Return "now" as a timezone-aware `datetime.datetime` object.

Additional parameters can be passed via keyword arguments; these values will be applied to the `datetime.datetime` object via its `replace` method. See Python's documentation for for more information .

If you would like to obtain a timezone-unaware datetime object, pass `tzinfo=None` as a keyword argument.

timedelta(*days=0*, *seconds=0*, *microseconds=0*, *milliseconds=0*, *minutes=0*, *hours=0*, *weeks=0*) → datetime.timedelta

Returns a `datetime.timedelta` object.

This object can be used in math with existing ``datetime.datetime``objects.

datetime(*year*, *month*, *day*, *hour=0*, *minute=0*, *second=0*, *microsecond=0*, *tzinfo=None*, ***, *fold=0*) → datetime.datetime

Returns a `datetime.datetime` object.

To obtain a `date` object, call `.date()` on the return value of this function.

parse_datetime(*datetime_string: str*, **args*, ***kwargs*) → datetime.datetime

Parse a date string into a datetime object.

This relies on *python-dateutil*; there are many additional options available that you can find documented here.

parse_date(*date_string: str*, **args*, ***kwargs*) → datetime.date

Parse a date string into a date object.

This relies on *python-dateutil*; there are many additional options available that you can find documented here.

4.6 Intervals

empty_interval() → portion.Interval

closed_interval() → portion.Interval

open_interval() → portion.Interval

openclosed_interval() → portion.Interval

closedopen_interval() → portion.Interval

4.7 Json

json_loads(*json: str*) → Union[Dict, List]

Parse a JSON string.

json_dumps(*obj: Union[Dict, List]*) → str

Encode a dictionary or list into a JSON string.

4.8 Math

4.8.1 Basic

See more in information in Python's Documentation.

abs(*value: float*) → str

max(*values: List[Any]*) → Any

min(*values: List[Any]*) → Any

pow(*base: float*, *exponent: float*, *mod: Optional[int]*) → float

round(*value: float*, *ndigits: int = 0*) → float

sum(*values: List[Any]*) → Any

4.8.2 Averages & measures of central location

See more in information in Python's Documentation.

mean(*values: List[Any]*) → Any

fmean(*values: List[Any]*) → float

Requires Python 3.8

geometric_mean(*values: List[Any]*) → float

Requires Python 3.8

harmonic_mean(*values: List[Any]*) → Any

median(*values: List[Any]*) → Any

median_low(*values: List[Any]*) → Any

median_high(*values: List[Any]*) → Any

median_grouped(*values: List[Any]*, *interval: int = 1*) → Any

mode(*values: List[Any]*) → Any

multimode(*values: List[Any]*) → List[Any]

Requires Python 3.8

quantiles(*values: List[Any]*, *n=4*, *method=Literal['exclusive', 'inclusive']*) → Iterable[Iterable[Any]]

Requires Python 3.8

4.8.3 Measures of spread

See more in information in Python's Documentation.

pstdev(*values: List[Any]*, *mu=Optional[float]*) → Any

pvariance(*values: List[Any]*, *mu=Optional[float]*) → Any

stdev(*values: List[Any]*, *xbar=Optional[float]*) → Any

variance(*values: List[Any]*, *xbar=Optional[float]*) → Any

4.8.4 Numeric Representation

See more in information in Python's Documentation.

bin(*value: int*) → str

hex(*value: int*) → str

oct(*value: int*) → str

ord(*value: str*) → int

4.9 List Operations

union(*iterable: Iterable[X]*) → X

4.10 Types

See more in information in Python's Documentation.

bool(*value: Any*) → bool

int(*value: Any*) → int

set(*value: Any*) → set

str(*value: Any*) → str

tuple(*value: Any*) → tuple

type(*value: Any*) → str

4.11 Sorting

See more in information in Python's Documentation.

reversed(*iterable: List[Any]*) → Iterable[List[Any]]

sorted(*iterable: List[Any]*) → Iterable[List[Any]]

4.12 Filtering & Mapping

simple_filter(*iterable: Iterable[Any]*, ***query_params: Dict[str, Any]*) → Iterable[Any]

simple_filter_any(*iterable: Iterable[Any]*, ***query_params: Dict[str, Any]*) → Iterable[Any]

These functions provide you with a simple way of filtering lists using a syntax reminiscent of Django's ORM query filter parameters.

- `simple_filter`: All provided `query_params` must match for the row to be returned.
- `simple_filter_any`: At least one provided param in `query_params` must match for the row to be returned.

For example; to find issues having become resolved between two dates, you could run a query like the following:

```
select: "*"
from: issues
filter:
- simple_filter(
      flatten_changelog(changelog),
      field__eq="resolution",
      toValue__ne=None,
      created__lt=parse_datetime("2020-02-02"),
```

(continues on next page)

(continued from previous page)

```
      created__gt=parse_datetime("2020-02-01"),
  )
expand:
- changelog
```

Consult the documentation for QueryableList for information about the full scope of parameters available.

4.12.1 Python Builtin Functions

See more in information in Python's Documentation.

filter(*callable: Callable, Iterable[Any]*) → Iterable[Any]

map(*callable: Callable, Iterable[Any]*) → Iterable[Any]

4.13 Logic Shortcuts

See more in information in Python's Documentation.

all(*iterable: List[Any]*) → bool

any(*iterable: List[Any]*) → bool

4.14 Counting

See more in information in Python's Documentation.

len(*iterable: List[Any]*) → int

You might be tempted to use `count()` given how we share many patterns with SQL, but *this* is what you actually want to use.

4.15 Ranges

See more in information in Python's Documentation.

range(*stop: int*) → Iterable[int]

range(*start: int*, *stop: int*) → Iterable[int]

range(*start: int*, *stop: int*, *step: int*) → Iterable[int]

4.16 Random

See more in information in Python's Documentation.

random() → float

randrange(*stop: int*) → int

randrange(*start: int*, *stop: int*) → int

randrange(*start: int*, *stop: int*, *step: int*) → int

randint(*low: int*, *high: int*) → int

choice(*Sequence[Any]*) → Any

CHAPTER

FIVE

QUERY PARAMETERS

When writing some queries that you'd like to reuse later, you may find a reason to want to pass-in a parameter at query runtime instead of altering the query directly. You can use query parameters for that.

For a contrived example, the below query will require that you specify a query parameter `project` that will be used when interpreting the query.

```
select:
  Issue Key: key
from: issues
where:
- project = "{params.project}"
- updated > "2023-01-01"
```

Note: See the "Can use query parameters?" section of *Query Structure Overview* for information about where these may be used.

You can specify the parameters to use via the `--param` command-line argument like so:

```
jira-select run-query --param="project=MYPROJECT" my-query.yaml
```

CHAPTER

SIX

HOW TO

6.1 Use Functions

Your `select`, `having`, `group_by`, and `sort_by` sections have access to a wide range of functions as well as to the full breadth of Python syntax. If the built-in functions aren't enough, you can also just write your own and either register them at runtime or make them persistently available via a setuptools entrypoint.

See *Query Functions* for a complete list of built-in functions.

6.2 Format data using functions

```
select:
  Status: status
  Summary: summary
  Story Points: "{Story Points}"
  Spring Count: len(customfield_10010)
  Sprint Name: sprint_name(customfield_10010[-1])
from: issues
```

In the above example, two of the displayed columns are processed with a function:

- *Sprint Count*: Will render the number of array elements in the field containing the list of sprints in which this issue was present.
- *Sprint Name*: Will show the name of the last sprint associated with the displayed issue.

6.3 Filter results using functions

```
select:
  Status: status
  Summary: summary
  Story Points: "{Story Points}"
from: issues
having:
  # The quoting below is required only because the first character of line
  # being a double-quote causes YAML parsers to parse the line differently
  - '"Sprint #19" in sprint_name(customfield_10010[-1])'
```

In the above example, the issues returned from Jira will be compared against each constraint you've entered in the `having` section; in this case, all returned issues not having the string "Sprint #19" in the name of the last sprint associated with the displayed issue will not be written to your output.

Note: `having` entries are processed locally instead of on the Jira server so filtering using *having* entries is slower than using standard Jql due to the amount of (potentially) unnecessary data transfer involved. It is recommended that you use `having` only when your logic cannot be expressed in standard Jql (i.e. in the `where` section).

6.4 Group results & calculate aggregates

You can group and/or aggregate your returned rows by using `group_by`:

```
select:
  Status: status
  Count: count(key)
from: issues
group_by:
  - status
```

You'll receive just a single result row for each status, and a count of how many records shared that status in the second column.

6.5 Sort results using functions

You can order your entries using any expression, too:

```
select:
  Status: status
  Count: count(key)
from: issues
group_by:
  - status
sort_by:
  - count(key) desc
```

This will sort all returned tickets, grouped by status, in descending order from the status that has the most tickets to the one that has the fewest.

Note: The `sort_by` section is evaluated locally instead of by your Jira server. In situations where your expression can be evaluated in Jql, you will have faster performance using the `order_by` section.

6.6 Limit the number of returned results

You can limit the number of results returned by adding a `limit` to your query:

```
select:
  Key: key
  Status: status
  Summary: summary
from: issues
where:
  - assignee = "me@adamcoddington.net"
limit: 10
```

Be aware that this limit is handled by Jira; so only the first N records will be available for downstream steps in the *Query Lifecycle*.

6.7 Expand Jira Issue Fields

You can ask Jira to expand issue fields by adding an `expand` element to your query:

```
select:
  Key: key
  Status: status
  Summary: summary
from: issues
expand:
  - transitions
```

The meaning of these expansions is defined by Jira; you can find more information in Jira's documentation.

CHAPTER

SEVEN

EXAMPLES

7.1 Finding all issues assigned to a particular user

```
select: "*"
from: issues
where:
- assignee = "some-user@some-company.com"
```

7.2 Summing the number of story points assigned in a particular sprint

```
select:
  Total Story Points: sum({Story Points})
from: issues
where:
- project = "MYPROJECT"
group_by:
- True
having:
- '"My Sprint Name" in sprint_name({Sprint}[-1])'
```

In Jira, your "Story Points" and "Sprint" fields may have any number of names since they're "Custom Fields" – their real names are things like `customfield10024` and `customfield10428`, but may vary instance to instance. You can use the field name directly in your query, but if you know only the "human-readable" name for your field, you can provide it in brackets as shown above with – *{Story Points}* and *{Sprint}*.

The `where` limitation here is used solely for reducing the number of records needing to be downloaded, and can be omitted if you are willing to wait.

The `group_by` expression here is to make all of your rows be grouped together so the `sum` operation in your `select` block will operate over all of the returned rows. `True` is used because that expression will evaluate to the same value for every row.

In the `having` section, you can see a fairly complicated expression that takes the last sprint associated with each returned issue, looks up that sprint's name and compares it with the sprint name you are looking for. We're using the `in` python expression here because I can't remember the full name, but I can remember part of it. You'll notice that the line is quoted; that's necessary only because the yaml parser interprets a line starting with a double-quote a little differently from one that does not. Try running the query without quoting the string to see what I mean.

7.3 Summing the total estimated size of issues per-person for a given sprint

```
select:
  Assignee: assignee
  Total Size: sum(map(estimate_to_days, timeestimate.originalEstimate))
from: issues
where:
- project = "MYPROJECT"
group_by:
- assignee
having:
- '"My Sprint Name" in sprint_name({Sprint}[-1])'
```

See *Summing the number of story points assigned in a particular sprint* for an explanation of the `having` section here.

In Jira, estimations are stored in the `timeestimate.originalEstimate` field, but since we've grouped our rows by assignee, `timeestimate` represents an array of objects instead of a single object holding the `originalEstimate` we want.

If we were to stop here, we would receive an array of strings looking something like:

```
["1d", "4h", "2d", "30m"]
```

but, we want to be able to sum these estimates, so we'll `map` each of those through the `estimate_to_days` function. This will create an array like this:

```
[1, 0.5, 2, 0.625]
```

An array isn't quite what we want – we want the total number of days – so we use the `sum` function to get that.

See *Query Functions* for more information.

7.4 Summing story points of issues resolved during a particular sprint

```
select:
  Assignee: assignee
  Story Points: sum({Story Points})
from: issues
where:
- project = 'My Project'
filter:
- simple_filter(
    flatten_changelog(changelog),
    created__gt=parse_datetime(get_sprint_by_name("Board Name", "Sprint Name").
↪startDate),
    created__lt=parse_datetime(get_sprint_by_name("Board Name", "Sprint Name").endDate),
    field__eq="resolution",
    fromValue__eq=None,
    toValue__ne=None
```

(continues on next page)

(continued from previous page)

```
  )
group_by:
- assignee
expand:
- changelog
```

The most important section in the above is in `filter`; here you'll see that we're using the `simple_filter` function for filtering the (flattened) list of changelog entries to those changelog enttries that were created during the sprint and indicate that the field `resolution` was changed from `None` to something that is not `None`.

For a row to be returned from `filter`, each expression should return a truthy value. So rows that do not have a corresponding changelog entry matching the above requirements will be omitted from results.

7.5 Summing worklog entries

```
select:
  Total Seconds: sum(extract(flatten_list(worklogs.worklogs), "timespentSeconds"))
from: issues
group_by:
- True
```

Worklog entries on issues are shaped like this for every row (unnecessary fields omitted):

```
{
   "total": 1,
   "worklogs": [
      {"timespentSeconds": 60},
      {"timespentSeconds": 100},
   ]
}
```

So, if we were to just select `worklogs.worklogs` we'd receive an array of results in this shape:

```
[
   [
      {"timespentSeconds": 60},
      {"timespentSeconds": 100},
   ],
   [
      {"timespentSeconds": 50},
   ]
]
```

The value we need is nested deeply in there, so we should first try to flatten the list of lists using `flatten_list`; if we do that, our list will become:

```
[
   {"timespentSeconds": 60},
   {"timespentSeconds": 100},
   {"timespentSeconds": 50},
]
```

We're still not quite there – the value under `timespentSeconds` still needs to be `extract``ed from the inner objects using ``extract`; if we do that we receive:

```
[
   60,
   100,
   50
]
```

We finally have something summable & can wrap that set of calls with `sum` giving us an answer of `210`.

The `group_by` expression here is to make all of your rows be grouped together so the `sum` operation in your `select` block will operate over all of the returned rows. `True` is used because that expression will evaluate to the same value for every row.

CHAPTER

EIGHT

COMMAND-LINE

8.1 *jira-select shell [–editor-mode=MODE] [–disable-progressbars] [–output=PATH] [–format=FORMAT] [–launch-default-viewer]*

Opens an interactive shell (a.k.a repl) allowing you to interact with Jira and see your query results immediately afterward.

This is a lot like the "shell" you might have used for postgres, mysql or sqlite. Except that this one syntax highlights your query and has tab completion.

- `--editor-mode=MODE`: Set the editor mode to use; options include `vi` and `emacs`. The default value for this can be set in your configuration file by setting `shell.emacs_mode` to `True` or `False`. See `--help` if you're not sure where your configuration file is.
- `--disable-progressbars`: By defualt, a pretty progressbar is displayed to provide an indication of how long you might have to wait for results. Using this option will disable this progressbar.
- `--output=PATH`: Instead of writing output to a temporary file, write output to the specified file path. This is useful if you're using the `--launch-default-viewer` option to work around OS-level security limits around what processes can read temporary files.
- `--format=FORMAT`: By default, the output is generated in `json` format, but you can select a different output format by setting `FORMAT` to `csv`, `html`, `table` or `json`.
- `--launch-default-viewer`: Display the generated output in your system's default viewer for the relevant filetype. You may need to use this argument if you are running on an operating system in which Visidata is not available (e.g. Windows when not running under Windows Subsystem for Linux).

8.2 *jira-select run FILENAME [–format=FORMAT] [–output=PATH] [–view] [–launch-default-viewer]*

Executes query specified in FILENAME and returns results in the specified format.

- `--format=FORMAT`: Sets the output format; options include `json` (default) `csv`, `html` and ``table`.
- `--output=PATH`: Sets the output path. If unspecified, the output will be written to stdout.
- `--view`: Open the appropriate viewer to view your query results after the query has completed.
- `--launch-default-viewer`: Display the generated output in your system's default viewer for the relevant filetype.

8.3 *jira-select install-user-script SCRIPT [–overwrite] [–name]*

Installs a python script into your user scripts directory. User scripts can be used to extend the functionality of jira-select by letting you write functions that can be available during your query operation. See *Direct Registration* for more information about how to use this.

- `SCRIPT`: Path to the python script (or `-` to import from stdin) to add to your user scripts directory.
- `--overwrite`: By default, an error will be returned if your query script matches the name of an existing script. Use this command-line argument if you would like to overwrite a script having the same name.
- `--name`: By default, the name will match the incoming filename (if it's available). Use this to override that behavior.

8.4 *jira-select build-query [–output=PATH]*

Allows you to interactively generate a query definition file.

- `--output=PATH`: Sets the output path. If unspecified, the output will be written to stdout.

8.5 *jira-select configure*

Allows you to interactively configure jira-select to connect to your Jira instance.

8.6 *jira-select setup-instance*

Configures an instance via the standard command-line arguments. See `--help` for more information. This is intended to be used programmatically; if you are a human, you probably want to use `configure` instead.

8.7 *jira-select –instance-name=NAME remove-instance*

Removes configuration for the instance having the specified name.

8.8 *jira-select show-instances [–json]*

Displays for you which instances are currently configured for use with jira-select.

- `--json`: Instead of displaying results in a pretty-printed table, export the results as json.

8.9 *jira-select store-password USERNAME*

Allows you to store a password for USERNAME in your system keychain.

- `USERNAME`: The username to store a password for.

8.10 *jirafs-select functions [–having=EXPRESSION] [SEARCH_TERM [SEARCH_TERM...]]*

Displays functions available for use in a query.

- `--having=EXPRESSION`: A `having` expression to use for filtering displayed results. The provided fields for filtering are `name` and `description`.
- `SEARCH_TERM`: A search term to use for filtering results. The term is case-insensitive and must be present in either the function name or description to be displayed.

8.11 *jira-select schema [issues|boards|sprints] [–having=EXPRESSION] [SEARCH_TERM [SEARCH_TERM...]] [–json]*

Displays fields available for querying a given data source.

- `--having=EXPRESSION`: A `having` expression to use for filtering displayed results. The provided fields for filtering are `id`, `type`, `description`, and `raw`.
- `SEARCH_TERM`: A search term to use for filtering results. The term is case-insensitive and must be present in either the function name or description to be displayed.
- `--json`: Instead of displaying results in a pretty-printed table, export the results as json.

8.12 *jira-select run-script FILENAME [ARGS...]*

Executes the `main(**kwargs)` function in the specified filename, passing it two keyword arguments:

- `args`: An array of extra arguments.
- `cmd`: The command class (via which you can access configuration, your jira instance, and other utilities).

This function is intended for use in ad-hoc scripting needs. If you are the sort of person to be running complex queries against your Jira instance, you're also likely to be the sort of person who will occasionally write an import script for ingesting issues into Jira. This utility function allows you to do that more easily by letting you lean on the Jira settings you've already configured jira-select to use.

Important: If you want to future-proof your script, be sure that the signature of your `main` function accepts `**kwargs` even if your signature already captures `args` and `cmd` explicitly. New keyword arguments may be added at any time.

Example contenst of a user script named `my_file.py`:

```
def main(args, cmd, **kwargs):
   print(f"Extra args: {args}")
   print(cmd.jira)
```

Running this file with:

```
jira-select run-script my_file.py --extra --args
```

Will print:

```
Extra args: ['--extra', '--args']
<jira.client.JIRA object at 0x7fc0a47e7e80>
```

CHAPTER

NINE

WRITING YOUR OWN PLUGINS

Jira-select relies on setuptools entrypoints for determining what functions, commands, and formatters are available. This makes it easy to write your own as long as you're familiar with python packaging, and if you're not, you can also register functions at runtime.

9.1 Commands

To write your own commands, you need to:

1. Create a class that is a subclass of `jira_select.plugin.BaseCommand`. This command:
   - Must implement a `handle` function.
2. Register that class via a setuptools entrypoint.
   - Your entrypoint should be in the `jira_select.commands` section.
   - The name of your entrypoint will become the command's name.

9.2 Functions

For functions, you have two choices:

1. You can create and install a user script into your user functions and within that script register a function using the method described in *Direct Registration* below.
3. If you plan to distribute your function on PyPI or would like for it to be installable generally, you can create an entrypoint; see *Entrypoint* below for details.

9.2.1 Direct Registration

1. Create a function in a python file somewhere.
2. Wrapping that function in `jira_select.plugin.register_function`.
3. Install that user script using the *install-user-script* command.

For example, if you have a file named `my_user_function.py` in your current directory with the following contents:

```
from jira_select.plugin import register_function
```

(continues on next page)

(continued from previous page)

```
@register_function
def my_important_function(value):
    """Returns length of `value`

    This function isn't doing anything useful, really, but
    you could of course make it useful if you were to write
    your own.

    """
    return len(value)
```

you could install it with:

```
jira-select install-user-function my_user_function.py
```

and after that, you will have access to `my_important_function` in a query like:

```
select:
- my_important_function(key)
from: issues
```

9.2.2 Entrypoint

1. Create a class that is a subclass of `jira_select.plugin.Function`. This command:
    - Must implement a `__call__` function.
2. Register that class via a setuptools entrypoint.
    - Your entrypoint should be in the `jira_select.functions` section.

9.3 Formatters

To write your own formatter, you need to:

1. Create a class that is a subclass of `jira_select.plugin.BaseFormatter`. This command:
    - Must implement a `writerow` function.
    - Must implement a `get_file_extension` classmethod returning your format's file extension.
    - May implement an `open` method for any setup functionality.
    - May implement an `close` method for any teardown functionality.
2. Register that class via a setuptools entrypoint.
    - Your entrypoint should be in the `jira_select.formatters` section.

CHAPTER

TEN

TROUBLESHOOTING

10.1 After running a query in jira-select's `shell` subcommand, the output results are printed directly to the screen instead of opening in a spreadsheet viewer

The viewer you see being used on in the demo gif is called Visidata, and unfortunately it isn't available on all platforms. You do have a few options, though:

1. You could use the `--format=table` command-line argument to tell jira-select to print your output to the screen in a fancy table mode.
2. You could ask jira-select to open the query results in your system's defualt viewer using the `--launch-default-viewer` command-line argument. On Windows, you will also need to specify an output path explicitly to make this work by using `--output=/some/path/to/write/output/to.csv`.
3. If you're running on Windows, you could install this under "Windows Subsystem for Linux" so that you can use the default viewer (visidata). See more information here: Windows Subsystem for Linux Installation Guide for Windows 10.
4. You could use the `run-query` subcommand instead of `shell`. This particular subcommand is a lot less fancy than `shell`, though.

10.2 Sometimes filtering using `having` (or sorting using `sort_by`) on a value I see in the output doesn't work; why not?

Oftentimes the data returned from Jira for a particular field is not a simple string or number and is instead a complex object full of other information. In those cases, we show the most reasonable string value was can obtain from the object instead of showing you the whole object.

To use such objects in `having` or `sort_by` expressions, you should convert them into a reasonable simple data type using one of the functions in *Types*.

If you're curious about which fields we're transforming into strings behind-the-scenes, try wrapping your column in `type` to see the data's actual type.

If you want to see the data that is being hidden by the above transformations – for example: for `issuetype` – you can access the raw Jira object via the `raw` property of the field; e.g.

```
select:
  Raw Issue Data: issuetype.raw
from: issues
```

10.3 I can't connect because my Jira instance uses a self-signed certificate

Don't worry, there are two command-line arguments you can use for configuring certificate verification:

- `--disable-certificate-verification`: For the brave. This will entirely disable certificate verification for this instance when configuring it as well as for all future connections to it.
- `--certificate=/path/to/certificate`: For the people who have a security team watching what they're doing. This will ask jira-select to use a particular self-signed certificate.

These are overrides available for all commands (not just `configure`) so these arguments can only be used between `jira-select` and the command you're running (probably only `configure` as when you use them with `configure` those settings will be recorded in your configuration's settings for the future):

```
jira-select --disable-certificate-verification configure
```

10.4 When attempting to use a field's human readable name in curly braces, I get a Parse Error

YAML, the file format we use for queries in jira-select, has some parsing rules that will make it interpret a line starting with a quote, curly brace, bracket, or other reserved characters differently from other lines.

In cases like those, you should just wrap your whole query expression in quotes; for example:

```
select:
  Story Points: '{Story Points}'
from: issues
```

CHAPTER

ELEVEN

REFERENCE

11.1 API Reference

11.1.1 jira-select

jira_select package

Subpackages

jira_select.commands package

Submodules

jira_select.commands.build_query module

jira_select.commands.configure module

jira_select.commands.show_instances module

jira_select.commands.functions module

jira_select.commands.run module

jira_select.commands.schema module

jira_select.commands.shell module

jira_select.commands.store_password module

Module contents

jira_select.formatters package

Submodules

Module contents

setup module

tests package

Submodules

tests.base module

tests.test_query module

tests.test_utils module

Module contents

CHAPTER

TWELVE

INDICES AND TABLES

- genindex
- modindex
- search

INDEX

C

D

E

F

G

H

I

J

L

M

N

O

P

Q

R

S

T

U

V