ZABBIX

ZQL – The Query Language

Petite et dabitur vobis quaerite et invenietis …

Vladimir Ulogov

PREFACE

What are the ZQ and ZQL ?

When the ZQ stands for “Zabbix Query”, “ZQL” is the “Zabbix Query Language”, which is the interactive and batch query language and Unix command line tool, for translating ZQL to a subset of the Zabbix API calls.

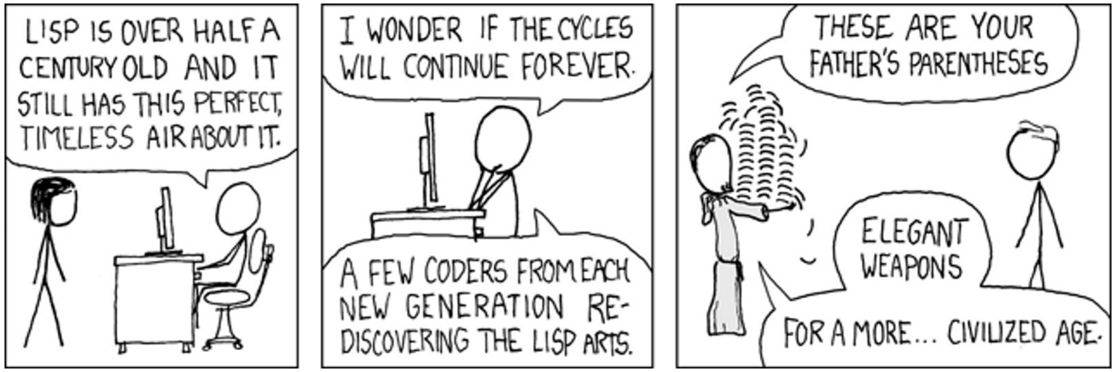
Is that a server, database library, or what ?

It it nether. ZQ is the Python module, which is compiled into a machine code for your platform using Cython. ZQL is a interpreted, stack-based language, loose dialect of the LISP, implemented inside this module with few external dependencies. ZQ/ZQL distribution also provides you a command-line tool ZQL, which act as standalone interpreter and a query executor.

ZABBIX

PREFACE

So, what kind of language is ZQL ?


LISP IS OVER HALF A CENTURY OLD AND IT STILL HAS THIS PERFECT, TIMELESS AIR ABOUT IT.
I WONDER IF THE CYCLES WILL CONTINUE FOREVER.
A FEW CODERS FROM EACH NEW GENERATION RE-DISCOVERING THE LISP ARTS.
THESE ARE YOUR FATHER'S PARENTHESES
ELEGANT WEAPONS
FOR A MORE... CIVILIZED AGE.

ZQL is a built around stack operations and LISP-based extendable interpreted language implemented inside the Python interpreter.

Vladimir Ulogov

PREFACE

Is that subset of SQL ?

NO
SQL

ZQL have no relation with Structured Query Language and never will be, thanks to it’s implementation.

ZABBIX

PREFACE

Show me some example of what’s ZQL is looks like ….

As you wish, here is an example of the correct ZQL, which connects to the server, grabs the list of the hosts, filter the hosts which do have a proxy configuration, merge proxy information and display output as JSON

```
(ZBX NEW “zabbix”) (Hosts) (Filter TRUE [proxy_hostid Ne 0]) (Proxies) (Merge proxy_hostid proxyid) (Out) (Pretty_Json)
```

SCEPTIC

- Are you serious !? How can anybody understand this mumbo-jumbo !

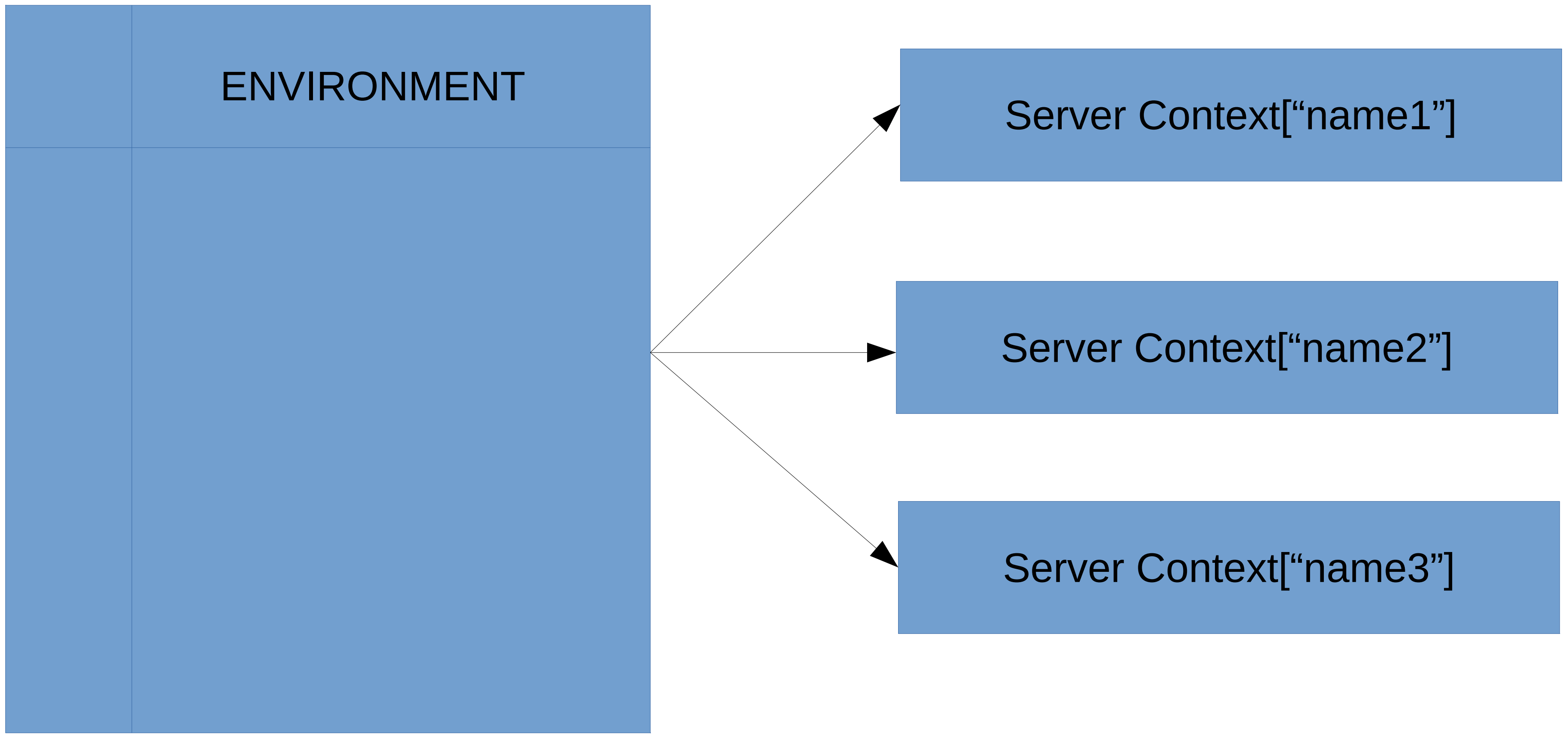
- Well, in fact, this is very simple, and powerful language. Once you’ve get the grasp on few basic concepts, you will find, that ZQL is quite powerful for both, simple day-to-day maintenance and reporting operations and more sophisticated configuration and infrastructure analysis.

YOU CAN DO IT

ZABBIX

Servers and Context

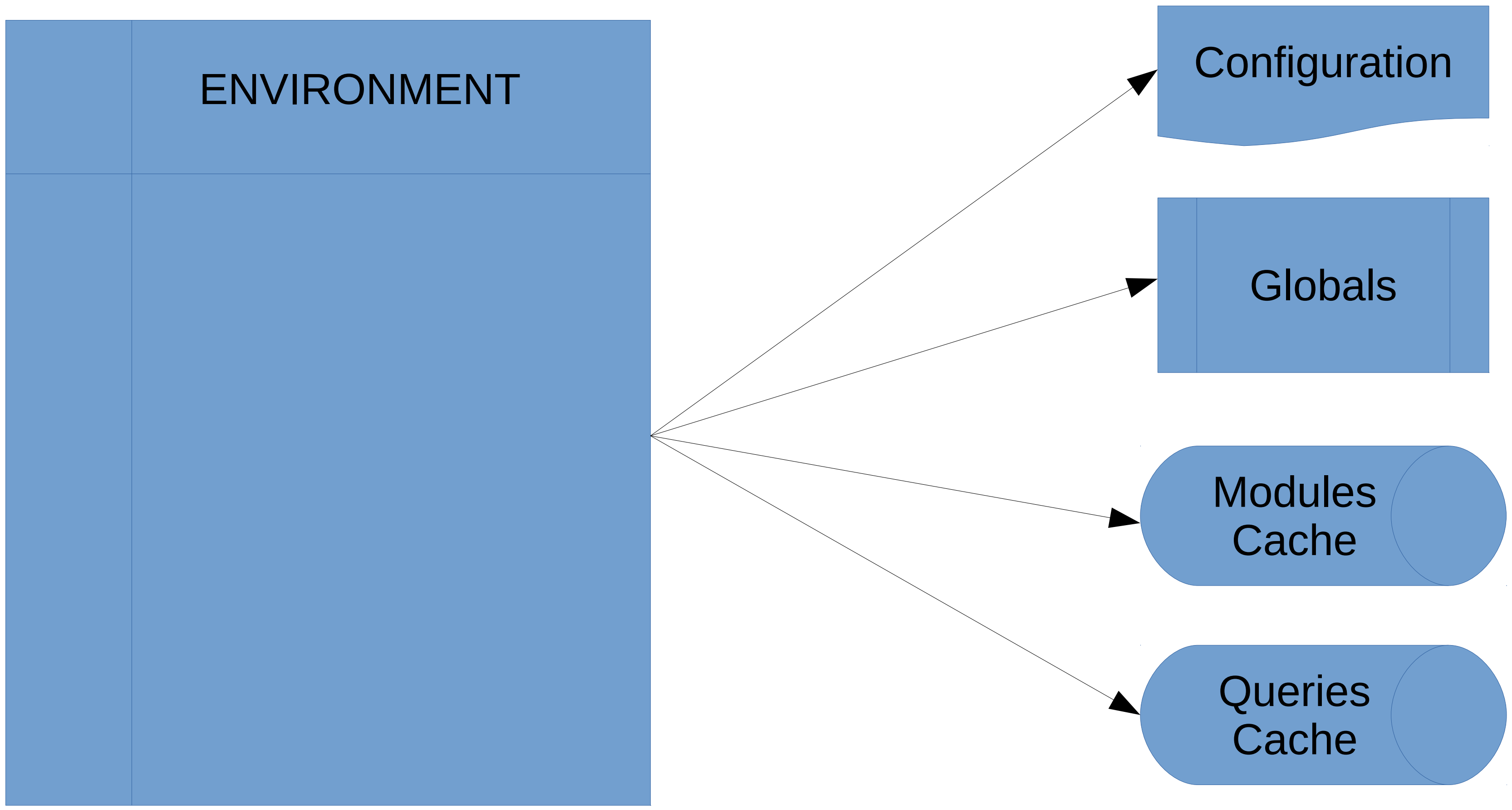
First, you have to understand the Concept of the Environment and Context.



ZABBIX

Servers and Context

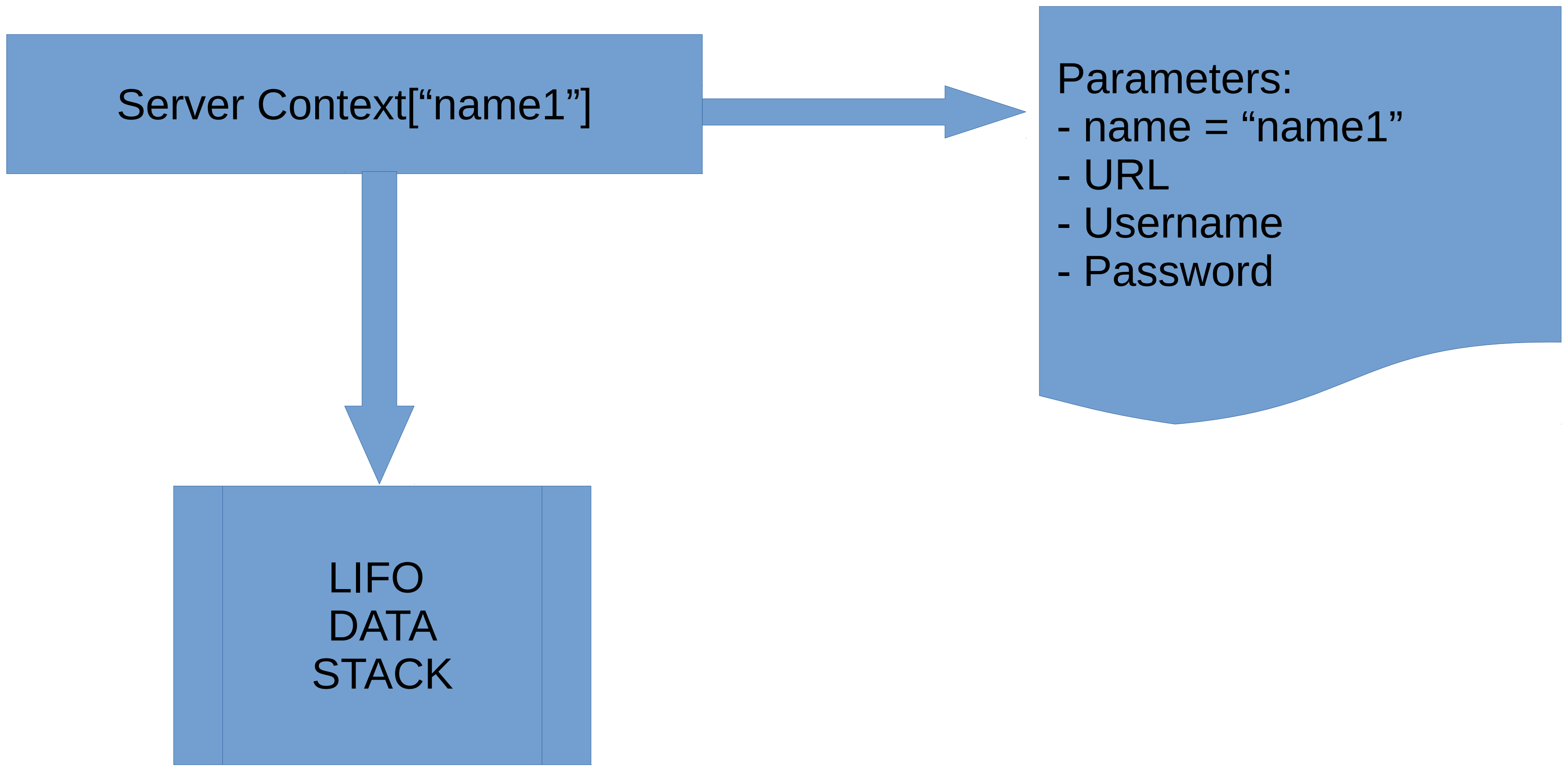
First, you have to understand the Concept of the Environment and the Context.



ZABBIX

Servers and Context

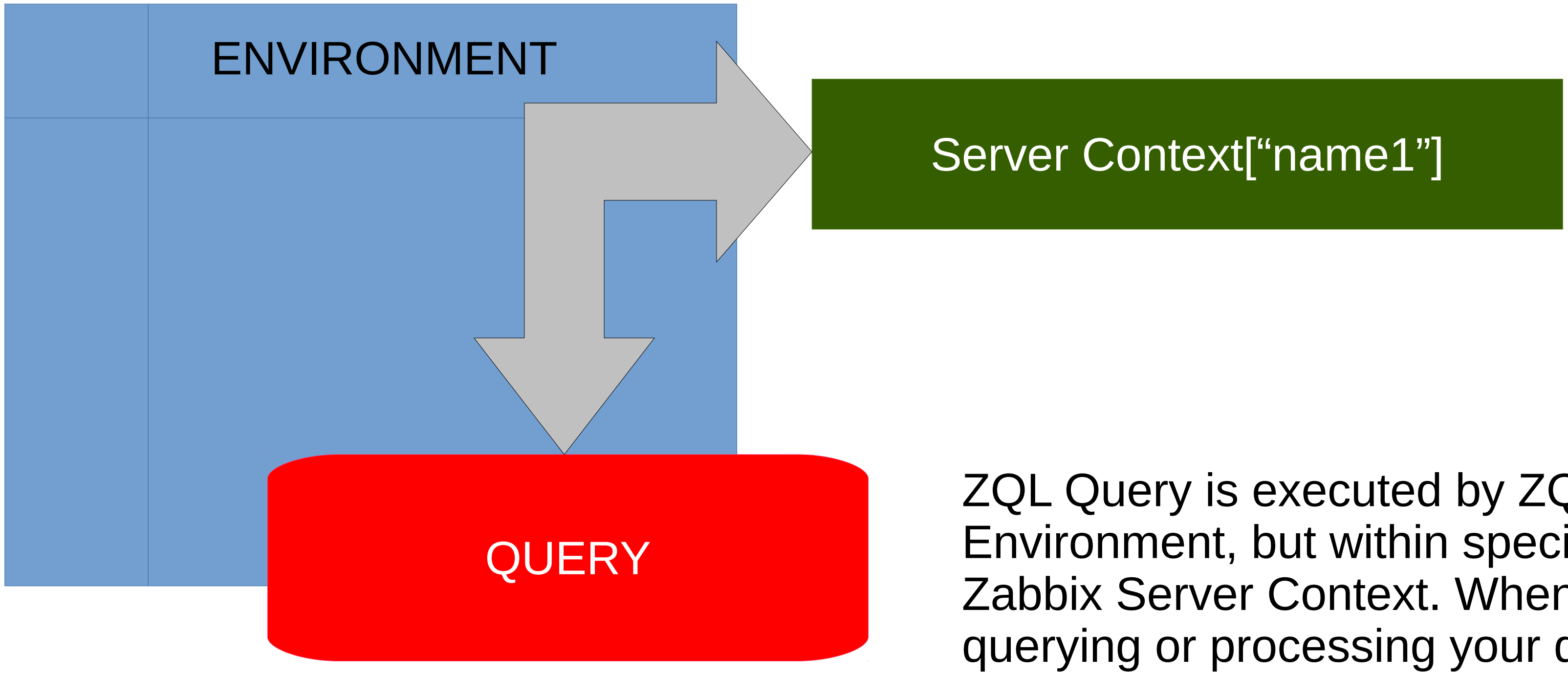
Each server context points at the specific Zabbix server and the attributes of the context helps ZQL environment to establish connection and to obtain and process the data.



ZABBIX

Servers and Context

First, you have to understand the Concept of the Environment and Context.



ZQL Query is executed by ZQ Environment, but within specific Zabbix Server Context. When you are querying or processing your data, you are receiving the reference on the server Context, which contains the reference to the Environment and ZQL Shell.

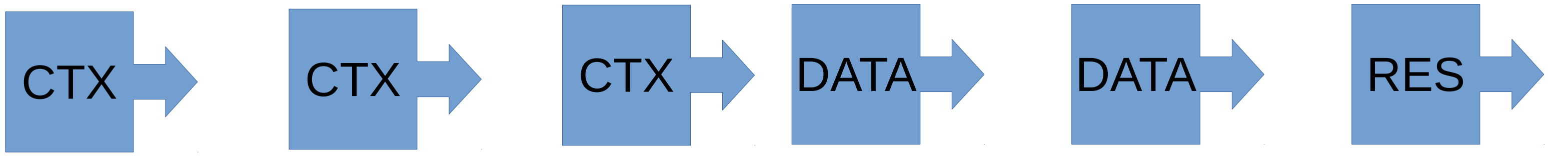
ZABBIX

Query structure

ZQL query consists of the “words”.

- Each word is a single S-Expression: (x y z)
- The “x” is the first element of an expression and the “name” of the word.
- The words in the query are called sequentially.
- Return value of the previous S-Expression (or “the word”) is passed to the next word down to the pipeline as indirectly defined first parameter of the function, which defining the word.
- Return value of the last word will be the return value of the ZQL query.
- There are four types of the words:
  - Context manipulating words;
  - Stack manipulating words;
  - Data manipulating words.
  - Function manipulating words (we will talk about them separately).

(Context) (Stack …) (Stack …) (Data) (Data …) (Data …)



Query structure

Wait ! But this is not a LISP S-Expressions !

(LISP)

And you are right. The S-Expression is a notation of the nested tree structure (list) data, for the Lisp programming language, used for source code and source data.

That’s is why I call it a “Query Language”, or “Zabbix Query Language” to link the purpose of this language with the specific application domain. The ZQL query is consisting of the words, each of this words is expression and it is passing data from word to word through the pipeline. This shall be very familiar for everyone, who is using Unix. ZQL is essentially is:

(ZQL)

```
cat /data/file | grep “pattern” | wc -l
```

ZABBIX

Query structure

Let us look at the relationship between nested s-expressions in LISP and ZQL pipelines:

(ZQL)

You have a query like this one in ZQL pipeline format:

(word1) (word2) (word3)

(LISP)

The same query in s-expression will look like this:

(word3 (word2 (word1)))

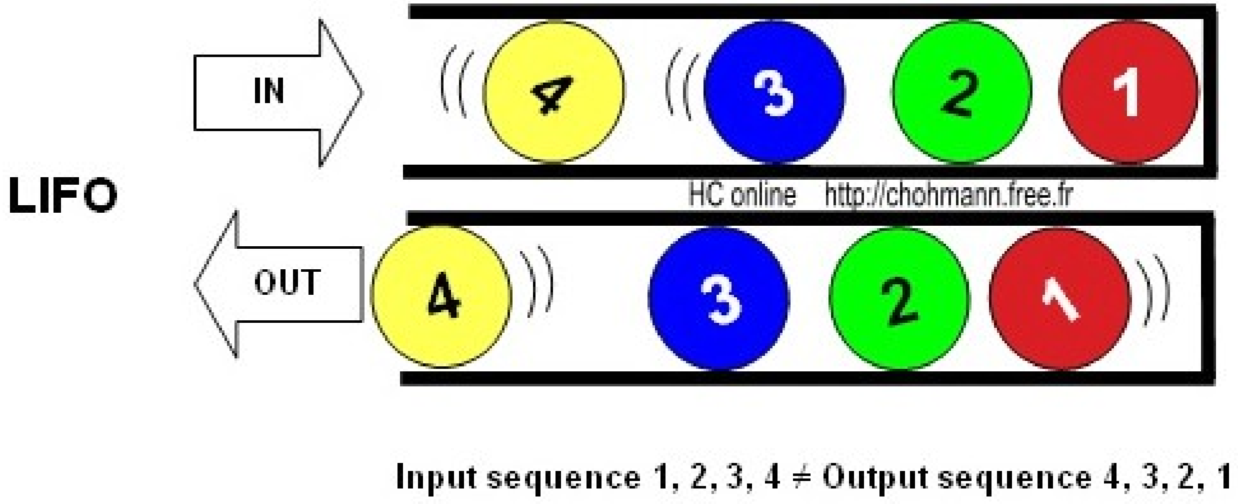
Further, I will be focusing on ZQL pipeline query format, but in ZQ module documentation and ZQL command-line tool documentation, I will show you, how you can send a Queries to the Zabbix Lisp-Style… Stay tuned.

ZABBIX

Stack

In the beginning, I was talked about a stack. Yes, ZQL Query processor is a nearly full implementation of the stack machine. All “context” and “stack” words, responsible for data acquisition and manipulations operates with that LIFO stack.

The “context” and “stack” words are passing the reference to a “context”. The context is a data structure inside the ZQ Environment, which corresponds to the “Zabbix Server context”



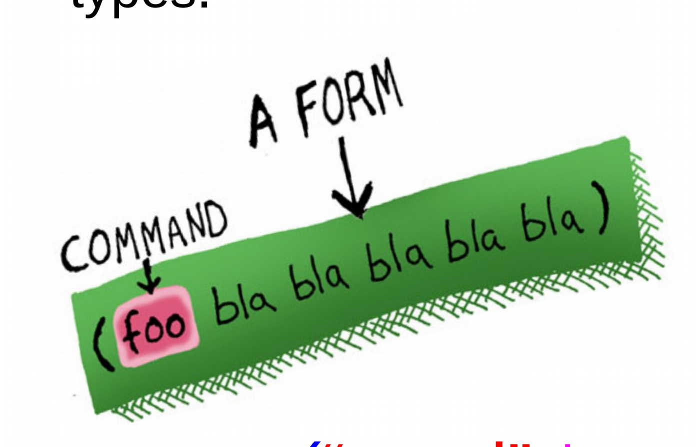
LIFO
IN
OUT
4
3
2
1
HC online http://chohmann.free.fr
Input sequence 1, 2, 3, 4 ≠ Output sequence 4, 3, 2, 1

and a set of the methods for the stack and data manipulation. You can push your data to the stack, pull the data from the stack, peek the data (i.e. get the reference on the top element of the stack without pulling it from the stack), and so on...

ZABBIX

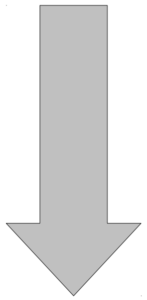
About words

Since we’ve started to talk about the “words”, it is about a right time to talk about word format, positional parameters, symbols, strings and other data types.



First, let's talk about how word is represented in ZQL:

(“word” *positional arguments **keyword arguments)



(Filter Eq :name “www.example.com”)

ZABBIX

About words

Next, let's talk about ZQL data types which you are going to use in positional and keyword arguments. And the first and most simplest are the SYMBOLS

Symbol – is an object with a simple string representation that (by default) is guaranteed to be interned; i.e., any two symbols that are written the same are the same object in memory (reference equality).
Symbol – is a pre-defined constant. They are ether pre-defined in ZQL core, or you can assign on-the fly by using (Setv …) word. We will talk more about this later. Symbols are alpha-numeric, case-sensitive, and can not begin with number or special character and must be pre-defined before first use in your Environment as part of the Environment Global dictionary. Otherwise, your query will throw an error.

Example of the symbols:

NONE, TRUE, FALSE, Ne, Eq ...

ZABBIX

About words

Strings, are also important datatype. You will deal a lot with strings, when querying and analyzing data from Zabbix

String – is a list of unicode characters, enclosed in quotas (“). If you are using quota as a part of the string, the quota must be escaped with forward “\”.

Example:

“Hello world!”
“Yes, I can contain the \”quotas\””

ZABBIX

About words

Numbers, in ZQL are corresponded to the numeric types in Python. There are integers, which are 64-bit longs (sys.maxint = `9223372036854775807 on my host`), and the floats.

Integers – numeric values which may be prefixed with "+" or "-"

Float – floating point numbers.

Example:

Integer: 42
Float: 3.14

ZABBIX

About words

Lists, in ZQL are the same as in the Python. This is a collection of the elements, which is not required to be the same type, and unlike in Dictionaries, you must address an element of the List by it's index. List indexes beginning with “0”. You can define list by listing it's elements, separated by one or more spaces, surrounded by [and]

Example:

```
[ 1 2 “answer is 42”
    [ “yes you can define list”
      “inside list”
      3.14
      “and mix types”
    ]
]
```

Please note, unlike in Python, where elements in the list are “comma” (“,”) separated, list elements in ZQL are space separated. If you “fall towards Python-way”, you'll get a nice cryptic traceback. I told ya….

ZABBIX

About words

Dictionaries, in ZQL like a Lists do have the same meaning as in Python. They are Hash Tables. Moreover, ZQL dictionaries represented by Python dictionaries on low level. And like with Lists, there is a variation on how you define the Dictionary.

So, as I mentioned earlier, the Dictionary is a Hash Table, means that there is a Key and the Value. And you are addressing the Value by it's corresponded Key. You can create a ZQL Dictionary by defining space separated pairs. The first element of the pair will be the Key, and the second element, will be the Value. And of course, there must be an even number of elements at all time in the dictionary, as, remember, one of them representing the Key, and the second – Value. And dictionary elements are enclosed in { and }

Please note, definitions of the Dictionaries in ZQL are different from the Python. If you'll try to go your "good ol' Python ways", you'll get a traceback. I told ya again ….

About words

Example:

```
{
  “Key” “Value”
  “Answer” 42
  “Pi” 3.14
}
```

… again, please note that the elements of the Dictionary are space separated.

ZABBIX

About words

Positional parameters is a values (each of them could be an s-expression), which are passed to the ZQL word. Positional parameters are separated with one or more spaces.

```
(word TRUE “String” 42 3.14 (Time.time))
```

If word requires some positional parameters and you do not provide them in your query, defaults may be used, if they are specified by your word developer. If the word do require parameter and you didn’t provide it and there is no default specified, the ZQL will throw an exception.

ZABBIX

About words

Keyword parameters is a values (each of them could be an s-expression), which are passed to the ZQL word in the form of Dictionary, which is a collection of the Key→Value pairs. Keyword parameters is the Dictionary, but you passing them not like dictionary we've just discussed. Keyword parameters are passed as Key→Value pairs, where the key is prefixed with ":". There must be even number of the keyword parameters elements. Otherwise, you know what will happens [*]

```
(word :answer 42 :pi 3.14 :Key "Value")
```

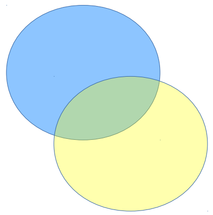
If keyword parameters are required, but not specified, this is up to word Developer on how to react on this situation In most cases, defaults will be engaged. Or you'll get an exception.

[*] *An exception, of course ….*

ZABBIX

Context words

In the beginning, thy shall be a “context word”.



“Context word” serves as important “capital letter” of your ZQL query. It creates and passes the first context. There are special case, where you can skip the first context word, but in order to do so, you do need to prepare your context in a special way. I will explain, how to do this later. Also, in the future releases of ZQ/ZQL I will provide you “better polished” mechanism on how you can switch server context within a single query “on the fly”.

There is currently only one “context word” is useful for the user, is (ZBX …)

As you already figure out, the ZQL words are context sensitive. So, the (ZBX) is (ZBX), not (Zbx) or (zbx).

YES

NO

ZABBIX

Context words

The first context-related word which is very useful to the ZQL user is (ZBX). And never forget, the name of the word is case sensitive

(ZBX NEW “*<server>*” “*<environment>*”)

This “word” takes three positional parameters:

- Reference to the previous context. At this moment, we are only accepting the symbol NEW;
- Name of the Zabbix server context in the environment. So, if you load the references to the multiple Zabbix servers in your Environment, this is the chance to provide the name in your Query to set with which server you will be working;
- Name of the environment in the “multi-environment” application. For the ZQL command-line tool (of version 0.5 and earlier), there is only one environment, and it’s name is “default” and default value for the third parameter is “default”.

(ZBX) word returns the reference to the Zabbix server context.

ZABBIX

Context words

The second context-related word will be very useful to you, if you are planning to execute queries from the zq.so module, or from some script, outside of the context of the zql command-line tool.

(ZBX-> *{references to the configuration file})

This “word” takes multiple positional parameters:

- The references to the Zabbix Server ZQL configuration file.

(ZBX->) word returns the reference to the last Zabbix server context from the last reference.

ZABBIX

Context words

We've just mentioned some “references to the Zabbix Server ZQL configuration”. What is that ?

```
[zabbix]
url=http://192.168.101.23/zabbix
username=Admin
password=zabbix
sender=192.168.101.23
sender_port=10051
```

This is a .ini formatted text file, containing the information on how ZQL shall connect to your Zabbix Server. The section name will become “name of the server” and the following parameters will hold the configuration for a Zabbix API.

Parameter	Description
url	URL of yor Zabbix server
username	Zabbix Username
password	Zabbix Password
sender	IP address of Zabbix Trapper, or the Proxy
sender_port	Zabbix Trapper or Zabbix Proxy port.

ZABBIX

Context words

Q: And what is the "reference" ?

A: Reference is the instruction on where ZQL can get the data for that resource.

There are two type of the references:

+{full or relative path} – reference to a file on local filesystem.

@{URL} – reference by URL.

Examples:

+/usr/local/etc/default/locahzabbix.ini

@http://192.168.101.23/zql/localzabbix.ini

ZABBIX

Stack manipulating words

There are two types of the words which is manipulating the stack. The words which performs the data acquisitions and place acquired data on the stack. And the words” which have nothing to do with data acquisition, they just manipulating data which is already in the stack.

ZQL is trying to be safe with the data. If in doubt, it will tend to do “be safe then sorry” and perform as little destructive operations with the data as possible. Of course, there are some words, like (Delete) which are very destructive by-design.

And yes, if you are using 'em, you are asking for it.

Stack manipulating words accepts context as the first parameter. But you do not have to pass it. ZQL runtime will do it for you. And stack manipulating words must return the current context. Please remember this, if you are planning to implement your own words in ZQL extension modules.

Getting data from Zabbix

The next series of the words, will do the Data Acquisition from the Zabbx server and the results will be pushed to the Stack. Keyword arguments (denoted as **keyword arguments) which you can pass to those words, will be passed to the corresponded Zabbix API *.get method. This feature can help you to create much more efficient and fine-tuned queries, but do expects some knowledge of the Zabbix API. By default, it will return **ALL** entries. For some of the entities, this is usually, not a big deal. But if you do not specify keyword parameters and you do have a 50000 hosts, you'll got 'em all. You've been warned. In the future versions, I'll implement some caching to help to save some network traffic and cut the load on the Zabbix servers at the same time.

Getting data from Zabbix

(Hosts **keyword arguments)

This keyword will connect to the Zabbix server defined by the current context, requests the list of the Hosts, and places result in the stack as a hash table, where the Key will be 'HOST' and the Value is information about hosts as it returned by ZabbixAPI[(*)]

Example query:

```
(ZBX) (Hosts)
```

This query will return the server context and hosts information will be pushed to the Stack.

[(*)] *List of the dictionaries.*

Getting data from Zabbix

(Interfaces **keyword arguments)

This keyword will connect to the Zabbix server defined by the current context, requests the list of the Hosts, and places result in the stack as a hash table, where the Key will be 'INTERFACE' and the Value is information about the host interfaces as it returned by ZabbixAPI(*)

Example query:

(ZBX) (Interfaces)

This query will return the server context and host interfaces information will be pushed to the Stack.

(*) *List of the dictionaries.*

ZABBIX

Getting data from Zabbix

(Hostgroups **keyword arguments)

This keyword will connect to the Zabbix server defined by the current context, requests the list of the Host Groups, and places result in the stack as a hash table, where the Key will be ‘HOSTGROUPS’ and the Value is information about the host groups as it returned by ZabbixAPI(*)

Example query:

```
(ZBX) (Hostgroups)
```

This query will return the server context and host groups information will be pushed to the Stack.

(*) *List of the dictionaries.*

Getting data from Zabbix

(Templates **keyword arguments)

This keyword will connect to the Zabbix server defined by the current context, requests the list of the Templates, and places result in the stack as a hash table, where the Key will be ‘TEMPLATE’ and the Value is information about the Templates as it returned by ZabbixAPI(*)

Example query:

(ZBX) (Templates)

This query will return the server context and templates information will be pushed to the Stack.

(*) *List of the dictionaries.*

ZABBIX

Getting data from Zabbix

(Items **keyword arguments)

This keyword will connect to the Zabbix server defined by the current context, requests the list of the Items, and places result in the stack as a hash table, where the Key will be 'ITEM' and the Value is information about the Items as it returned by ZabbixAPI(*)

Example query:

```
(ZBX) (Items)
```

This query will return the server context and items information will be pushed to the Stack.

(*) *List of the dictionaries.*

ZABBIX

Getting data from Zabbix

(Triggers **keyword arguments)

This keyword will connect to the Zabbix server defined by the current context, requests the list of the Triggers, and places result in the stack as a hash table, where the Key will be 'TRIGGER' and the Value is information about the Triggers as it returned by ZabbixAPI(*)

Example query:

```
(ZBX) (Triggers)
```

This query will return the server context and triggers information will be pushed to the Stack.

(*) *List of the dictionaries.*

Getting data from Zabbix

(Proxies **keyword arguments)

This keyword will connect to the Zabbix server defined by the current context, requests the list of the Proxies, and places result in the stack as a hash table, where the Key will be ‘PROXY’ and the Value is information about the Proxies as it returned by ZabbixAPI(*)

Example query:

(ZBX) (Proxies)

This query will return the server context and proxies information will be pushed to the Stack.

(*) *List of the dictionaries.*

Getting data from Zabbix

(Applications *args **keyword arguments)

This keyword will connect to the Zabbix server defined by the current context, requests the list of the Applications, and places result in the stack as a hash table, where the Key will be 'PROXY' and the Value is information about the Applications as it returned by ZabbixAPI[(*)]

Example query:

(ZBX) (Applications)

This query will return the server context and applications information will be pushed to the Stack.

[(*)] *List of the dictionaries.*

Data, data and more data

```
(Filter COMPARATIVE_FUNCTION
                          *[optional positional filtering parameters]
                            optional keyword filtering parameters
)
```

The word (Filter) will pull the first element from the Stack[*], apply the filtering conditions to the data and push the result back to Stack with the data keys unmodified. This is as close to the SQL “select” query as you can get. Bear in mind, ZQL performs filtering while you data is in memory.

(*) as you remember, the Stack is a LIFO stack, means the single (Filter) call always operates with the last element which is pushed to the Stack.

ZABBIX

Data, data and more data

Before discussing of what is the COMPARATIVE FUNCTION, let me introduce you to the format of the positional filtering parameters. Positional filtering parameters consisting of the one or more 3-element lists. The format of the single filtering element is:

[<keyword in the dataset> COMPARATIVE_FUNCTION <value>]

The subelements of the filtering element are space-separated and enclosed in [ ]. So if you remember slide about Datatypes, you will recognize a three-element List.

Your data is usually returned by Zabbix as the List of the Dictionaries, containing key-value pairs. <keyword> parameter, or the first element of the filtering element will define the key which will refer the value from the dataset which we will test with the help of the comparator. The <value>, or the third element in the filtering element, will define “comparative or base value” which will be used during the filtering. And finally, <COMPARATIVE_FUNCTION>, or the second item of the filtering element, is a name of the ether “lambda expression” or the “function” which will be called with first parameter is the context, second parameter is the data from the dataset and third parameter is a “base value” from the filtering element. If COMPAARATIVE_FUNCTION returns “True”, then data element is passed the filter, otherwise, this dictionary will be removed from the derived dataset.

ZABBIX

Data, data and more data

COMPARATIVE FUNCTIONS

Name	Description
Eq	One parameter is equal to another
Ne	One parameter is not equal to another
Lg	Data from dataset is smaller than the <value>
Lge	Data from dataset is smaller or equal than the <value>
Gt	Data from dataset is larger than the <value>
Gte	Data from dataset is larger or equal thean the value
Match	Data from dataset is matching the base value

Data, data and more data

Examples:

[“hostid” Eq 10084] or [hostid Eq 10084]

Filter data after the (Hosts), filter-out everything but the record where key “hostid” is 10084. You can use second form, because “hostid” also defined as a symbol

[“proxyid” Ne 0] or [proxyid Ne 0]

Filter data after the (Hosts), filter-out all records for which value of the key “proxyid” is not 0. This filter-expression will remove all the hosts which is not handled by the proxy.

Data, data and more data

Keyword-based filtering expressions.

:key <value>

Since, the keywords are nothing but the key-value pairs, here where the first parameter of the (Filter) “word”, the COMPARATIVE_FUNCTION comes to play. So, what is the COMPARATIVE_FUNCTION passed as first parameter ? The reference to the function or lambda-expression passed as the second parameter will be used by all keyword-based expressions. So the expressions:

(Filter TRUE [“hostid Eq 10084])

and

(Filter Eq :hostid 10084)

are essentially the same. In fact, you can mix positional and keyword-based filtering expressions in the same “word”. When processed, positional filters comes first in the order in which they are positioned and keyword-based expressions comes second. But bear in mind. The first parameter of the (Filter...), the COMPARATIVE_FUNCTION is applicable **ONLY** to the keyword-based filters. If you are not using keyword-based filter expressions, you still **can not omit it**. Pass the TRUE symbol.

ZABBIX

Data, data and more data

Examples:

```
(Hosts) (Filter TRUE [ “status” Ne 0]) (Out)
```

This query will return you the list of the hosts which status is not 0

```
(Hosts) (Filter Eq :proxy_id 0) (Out)
```

This query will return you the list of the hosts which is not handled by the proxy

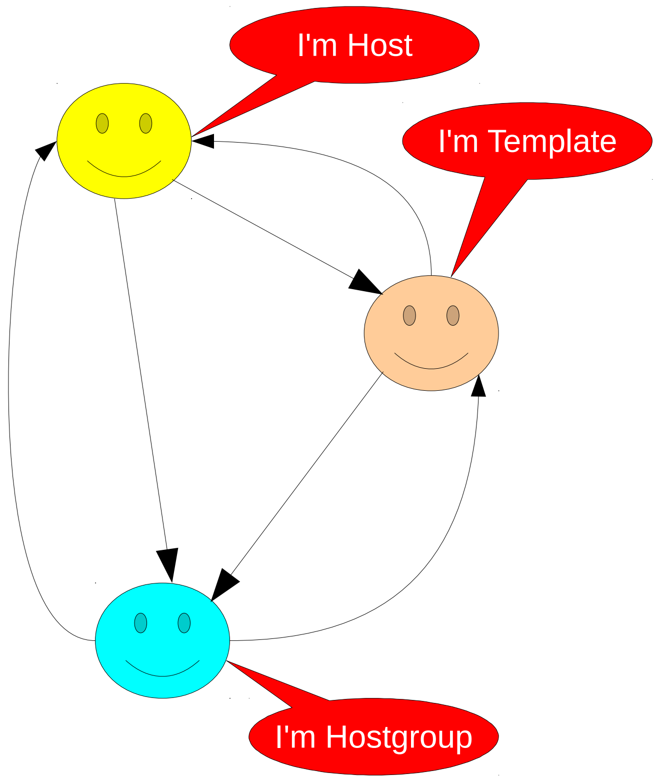
```
(Hosts) (Filter Eq [proxy_hostid Eq 0] :host "Zabbix server")
    (Out) (Pretty_Json)
```

This query return you the information about host “Zabbix server” if it is not handled by a proxy.

ZABBIX

Data, data and more data

One of the most powerful features of ZQL is a (Join …) word. This is as close to a JOINT SELECT from the relational world as you can get in ZQL. So, what is the (Join…) in the nutshell ?



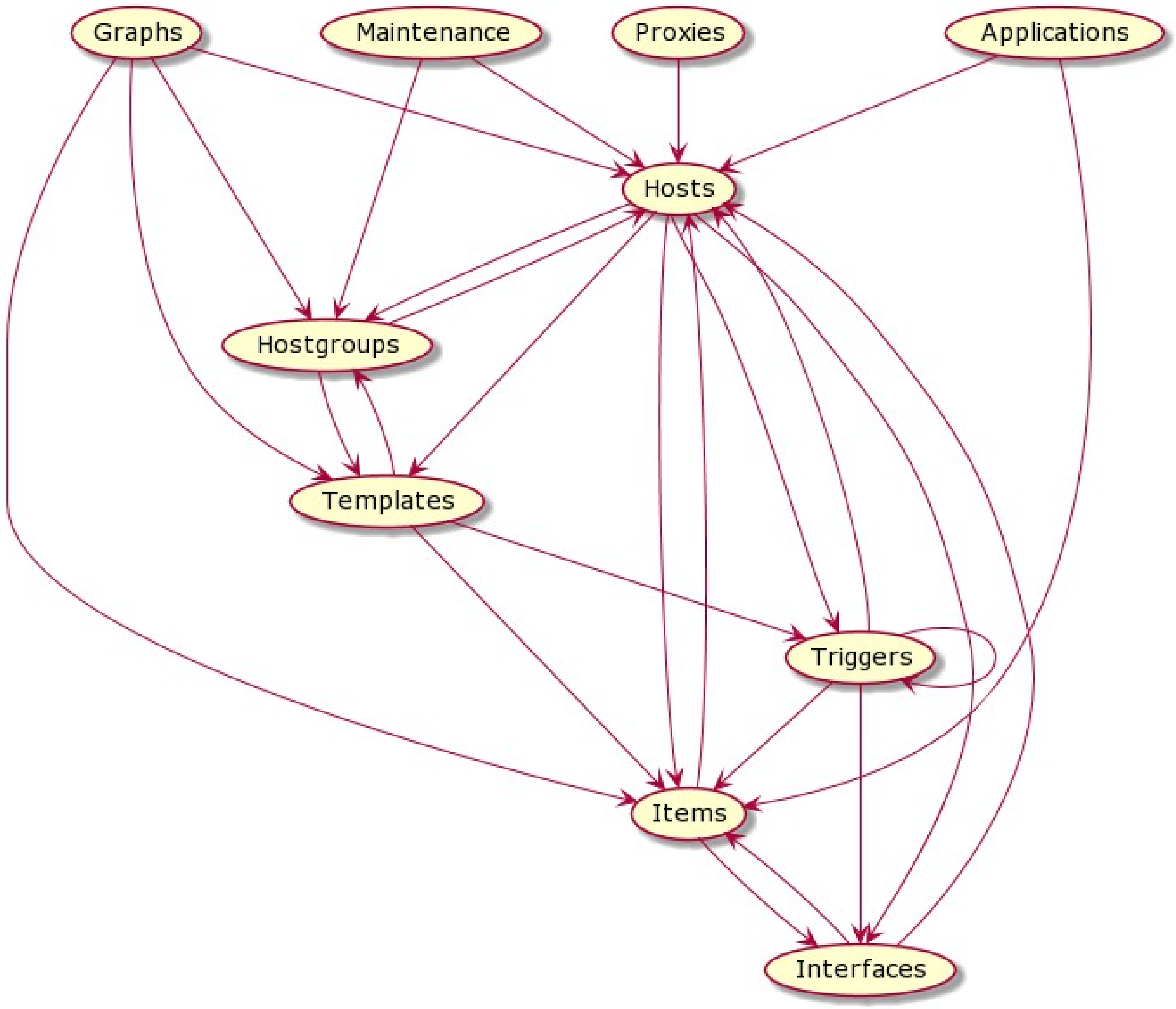
Each and every Zabbix configuration or data entity that you can query, do have the references to another entities. Hostgroups refering Hosts which are members of the Hostgroups, Interfaces refering Hosts, Template refering Hosts, Templates and so on…

So, what (Join…) is doing, it is pulling the element from the Stack, performing the query for the elements to which this element is referring to, and push information about that elements, the ones which has been discovered to the Stack.

This is exactly like performing the query, limited to certain subset of the elements.

ZABBIX

Data, data and more data



This nice diagram shows the relationship between Zabbix entities and entities to which they can refer to.

For example, using (Join…) it will be trivial to find which hosts belongs to certain Hostgroup, or which Items trigger are referring to or which Triggers Trigger is depend on.

So, one-level (Join…), as I hope is not a mystery anymore. But how to, for example, get a list of the Items, handled by specific Proxy ? Or which Templates could be involved with downtime of the Hosts, caused by scheduled Maintenance ? Can (Join…) be recursive ?

ZABBIX

Data, data and more data

No, (Join…) can not be recursive, but you can create a chain of the (Join…) calls. All thanks to the ZQL pipeline design, you can travel across your configuration any way you like. So, let's bring one of the before-mentioned examples to life.

```
(ZBX)
  (Proxies :proxyid 10)
  (Join)
  (Join
     :TEMPLATE False
     :HOSTGROUPS False
     :INTERFACE False
     :TRIGGER False
  )
(Out)
```

First, let's select Specific proxy with proxyid 10

First (Join) doesn't do much, but it will give us the list of (Hosts) handled by this (Proxy) on the Stack

And the second (Join) Will take inormation About (Hosts) from the Stack and will give us Only (Items) back to Stack

And then (Out)

ZABBIX

Data, data and more data

But how come that I never discuss about how you can limit the (Join...) ? Let's talk about that.

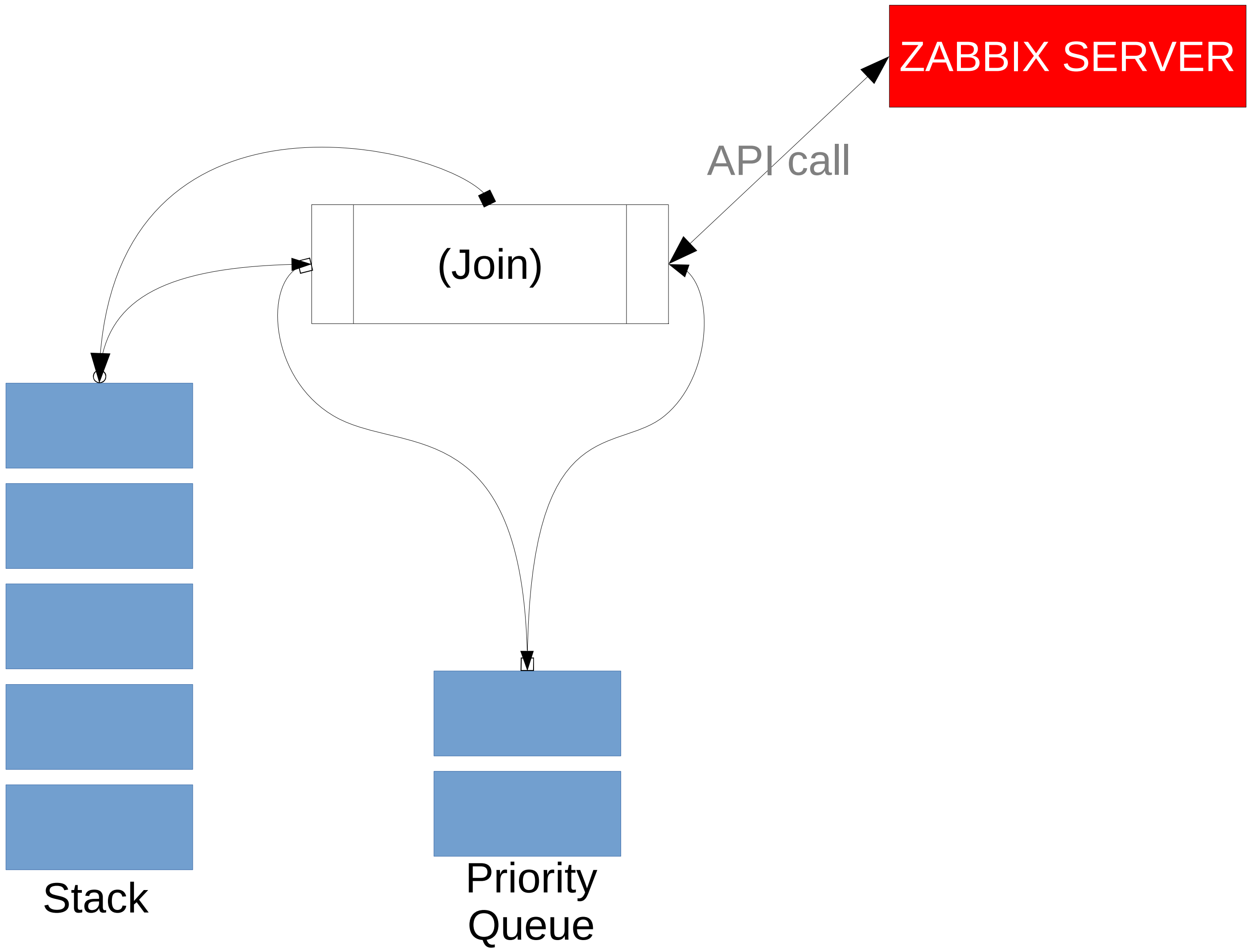
And before we will discuss this topic, we will briefly talk about the mechanism of (Join…) operation. (Join…) will pull data from the Stack until ether one condition happens: Stack become empty, or non-supported element is pulled. If we pull non-supported element, we will push it back to Stack and will commence processing.

But while we are pulling the data, we will push retrieved supported elements into in-memory priority Queue. If you are interested about ordering principle, please take a look at the source code or ask me. So, once we reach the point when we start processing the data, we start to pull the data elements from the priority Queue, analyze those elements and query Zabbix Server and push the results back to stack.

By default, (Join…) will process all supported types of an entities. You can select behavior of (Join…) by passing keyword parameter :{ENTITY NAME} True/False. If entity name is not passed, or if passed and the value is True, then this entity types will be processed. If it is passed and st to False, it will be skipped.

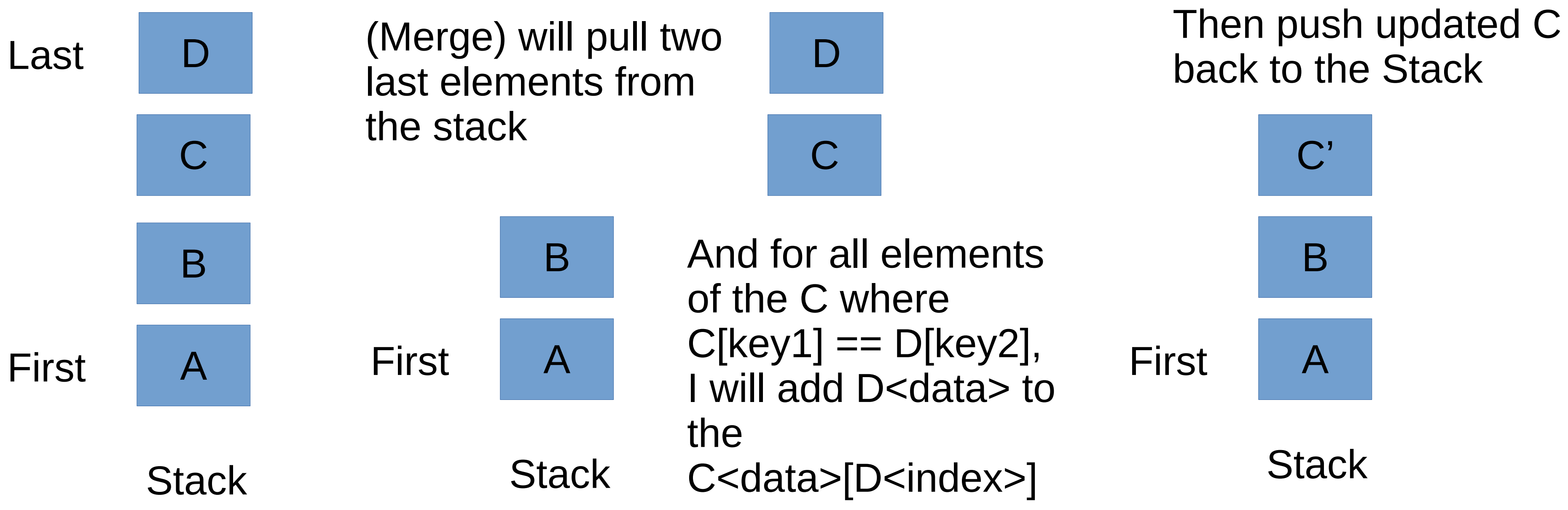
ZABBIX

Data, data and more data



Stack, stack and more Stack....

ZQL is a Stack-oriented language, where the words (they are s-expressions) evaluates sequentially, in a “pipeline” manner. ZQL words ether operates with the data or with the stack or with the functions. So far we've covered some data acquisitions and manipulation words. Now, it is a time to discuss the stack operations. And we will start with most complicated “word” - (Merge <key1> <key2>)



(Merge) will pull two last elements from the stack

And for all elements of the C where C[key1] == D[key2], I will add D<data> to the C<data>[D<index>]

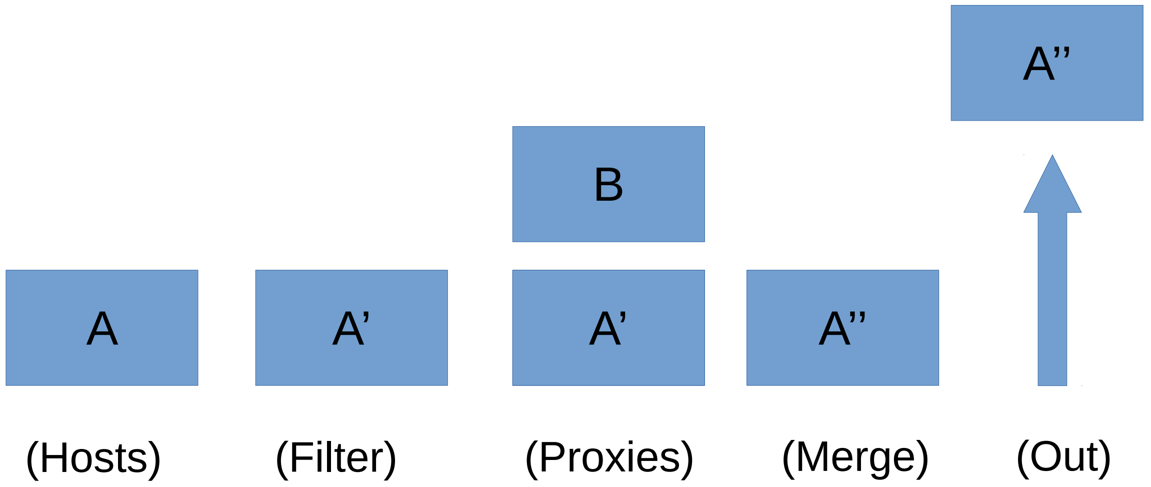
Then push updated C back to the Stack

ZABBIX

Stack, stack and more Stack....

Example:

```
(Hosts) (Filter TRUE [proxy_hostid Ne 0])
    (Proxies) (Merge proxy_hostid proxyid)
        (Out)
```

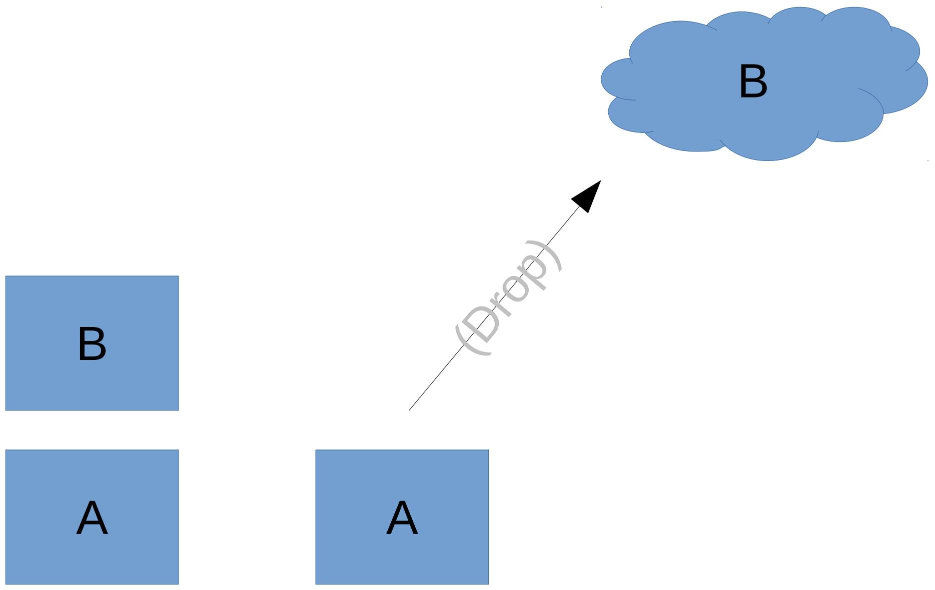


ZABBIX

Stack, stack and more Stack....

(Drop)

Discards the element from the top of the stack.

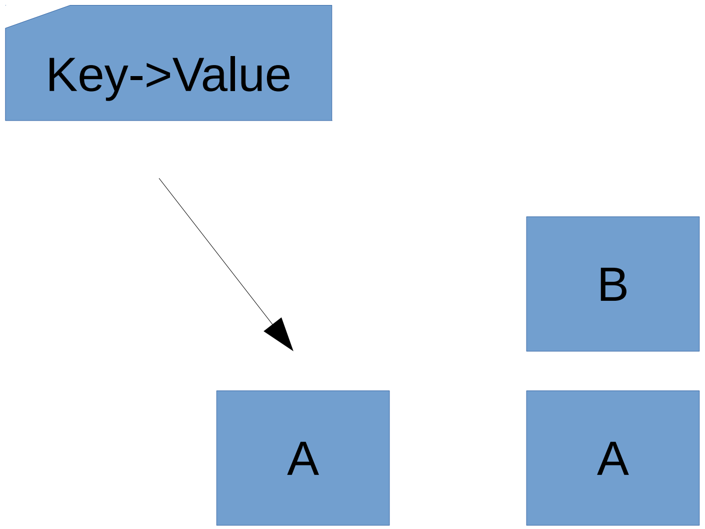


ZABBIX

Stack, stack and more Stack....

(Push <key> <value>)

Push key-value pair to the top of the stack.

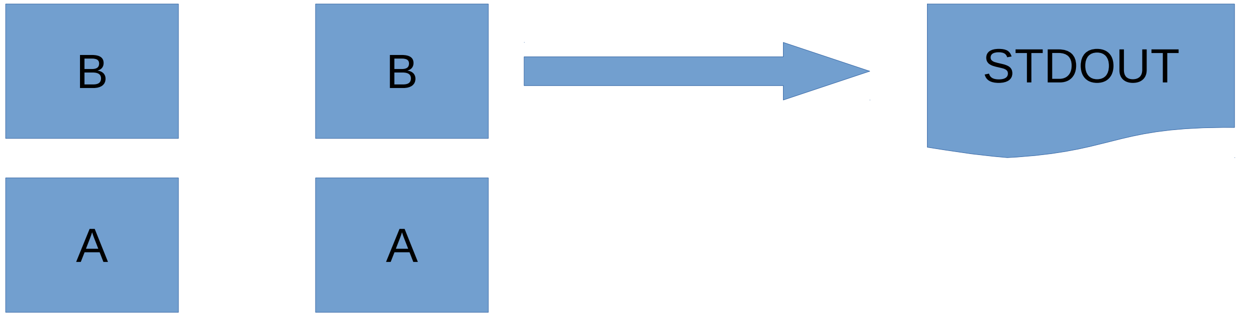


ZABBIX

Stack, stack and more Stack....

(Peek)

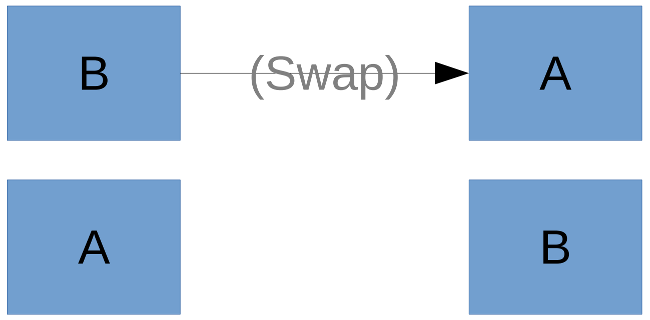
Print the value of the last element of the stack on the STDOUT without removing it from the stack



Stack, stack and more Stack....

(Swap)

Swap the top element of the Stack with the previous to the top one.



ZABBIX

Yes, we do have GLOBALS

In ZQL, we do have a Dictionary of the Global Variables, defines as per Environment. In the multi-Environment setup (which is not fully supported yet), you will be able to define a unique GLOBALS per environment. The first and most important word to deal with the Global Variables is (Setv ...)

```
(Setv {variable name} {value} **{keyword parameters})
```

This word will permit to define a single or multiple Global Variables. You can ether specify the “classic” name→ value, like this:

```
(Setv “ANSWER” 42)
```

Or set th multiple variables, by using keyword parameters:

```
(Setv :ANSWER 42 :Pi 3.14 :Hello “World”)
```

Words, words and more words ....

$$x = f(y)$$

During the next chapter, we will discuss some elements of the functional programming, available for ZQL user. So far, we've discuss data manipulation and storage, now let's take a look on how you can pass and use the references to the functions.

Functions in ZQL is as the name suggests, **a functions**. Executable code, which can accept some parameters and return the value. Functions are defined ether directly, or through the Zabbix Module System (which we will discuss later)

ZABBIX

Words, words and more words ....

First and simplest pair of functional words, are the (Do…) and (Do! …). The format of the arguments for those words are the same, but return value is different. (Do …) will return a “context” and (Do! ...) will return the result returned by the function

```
(Do {Key for the Stack} {Reference to the Function} *{arguments for the function} **{keyword parameters for the function})

(Do! {Key for the Stack} {Reference to the Function} *{arguments for the function} **{keyword parameters for the function})
```

Example:

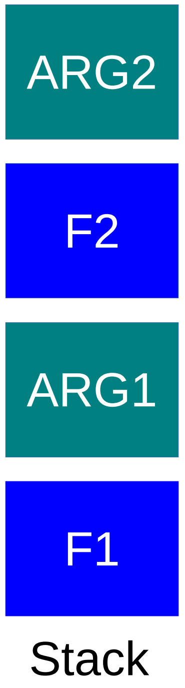
```
(ZBX)
    (Do “TIMESTAMP” Time.time())
(Out)
```

This ZQL code will push a current timestamp to the Stack, with the key “TIMESTAMP”

ZABBIX

Words, words and more words ....

The next group of functional words operating with execution context and arguments previously pushed to the Stack. When ZQL engine commence execution, it will pull the arguments and the execution contexts (the references to the function) from the stack and execute each function with each discovered argument. Order in which you will push functions and arguments to the Stack doesn't matter. ZQL will properly pass the reference to the "context" as the first argument of the function.

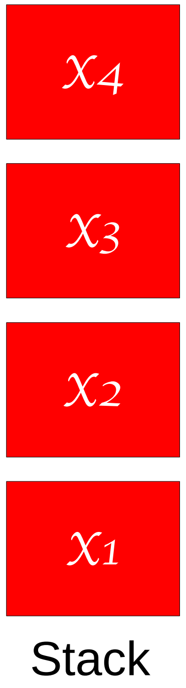


$X_1 = F_1(ARG_1)$

$X_2 = F_1(ARG_2)$

$X_3 = F_2(ARG_1)$

$X_4 = F_2(ARG_2)$



Words, words and more words ….

As you see, the first order of business is to push “execution environment” or information about function that you are planning to execute to the Stack. And here is the word:

```
(F→ *{positional arguments} **{keyword arguments})
```

Positional arguments is a sequence of the two-element Lists. First element of the List is the name of the function, the second is the reference to the function.

Keyword arguments is a pairs, where the “Key” is a name of the function, and Value is a reference to the function. You can push information about functions using ether way or mix them. But remember, I do not check for duplicates.

Example:

```
(ZBX)
  (F→ [“Demo.PrintArgs” Demo.PrintArgs])
```

ZABBIX

Words, words and more words ....

The second “order of business” is to push information about arguments to the Stack. And of course, here is the word:

(Args→ *{positional arguments} **{keyword arguments})

Positional arguments is a set of values, which will be passed to the function that you are executing as positional arguments.

Keyword arguments will be passed to the function as keyword arguments.

Example:

```
(ZBX)
  (F→ [“Demo.PrintArgs” Demo.PrintArgs])
  (Args→ “Hello word” (3.14,) :answer 42)
```

ZABBIX

Words, words and more words ....

And the finally, when both execution context and arguments are in stack, let's execute the functions:

```
(F*  **{keyword arguments})
```

Keyword arguments will be added to all keyword arguments pushed by (Args→ …). In case of the conflict, key-value from (F* …) overpower the key-value from the (Args→ ...)

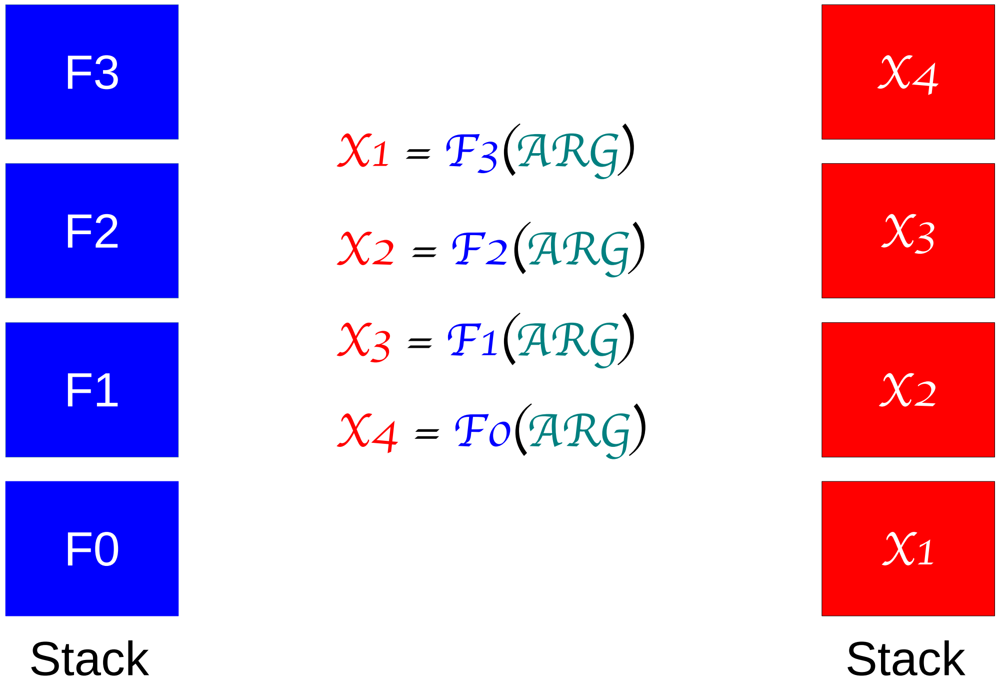
Example:

```
(ZBX)
  (F→ [“Demo.PrintArgs” Demo.PrintArgs])
  (Args→ “Hello word” (3.14,) :answer 42)
  (F* :default “This is default”)
(Out)
```

ZABBIX

Words, words and more words ....

Another functional word is (F …) which is variation of the (F* …). The key difference is, if (F* …) uses execution context and arguments from the Stack, (F …) uses function arguments passed as arguments, but still pulling execution context from the Stack and pushing result to the Stack.



$X_1 = F_3(ARG)$

$X_2 = F_2(ARG)$

$X_3 = F_1(ARG)$

$X_4 = F_0(ARG)$

ZABBIX

Words, words and more words ....

Format of the (F …) word is:

(F *{positional arguments} **{keyword arguments})

And passed positional and keyword arguments will be used with the pulled execution context.

Example:

```
(ZBX)
    (F→ :Demo.PrintArgs Demo.PrintArgs)
    (F “Hello world” :answer 42)
(Out)
```

ZABBIX

Words, words and more words ....

And the last, but certainly not least functional word is (F! …) . This word is different, and honestly, I've created it for the single case that I've had. I do not know if you ever going to need it, but anyway.

This word isn't taking the reference to the context, it is taking the reference to the function and it is not returning a context, it's returning the data.

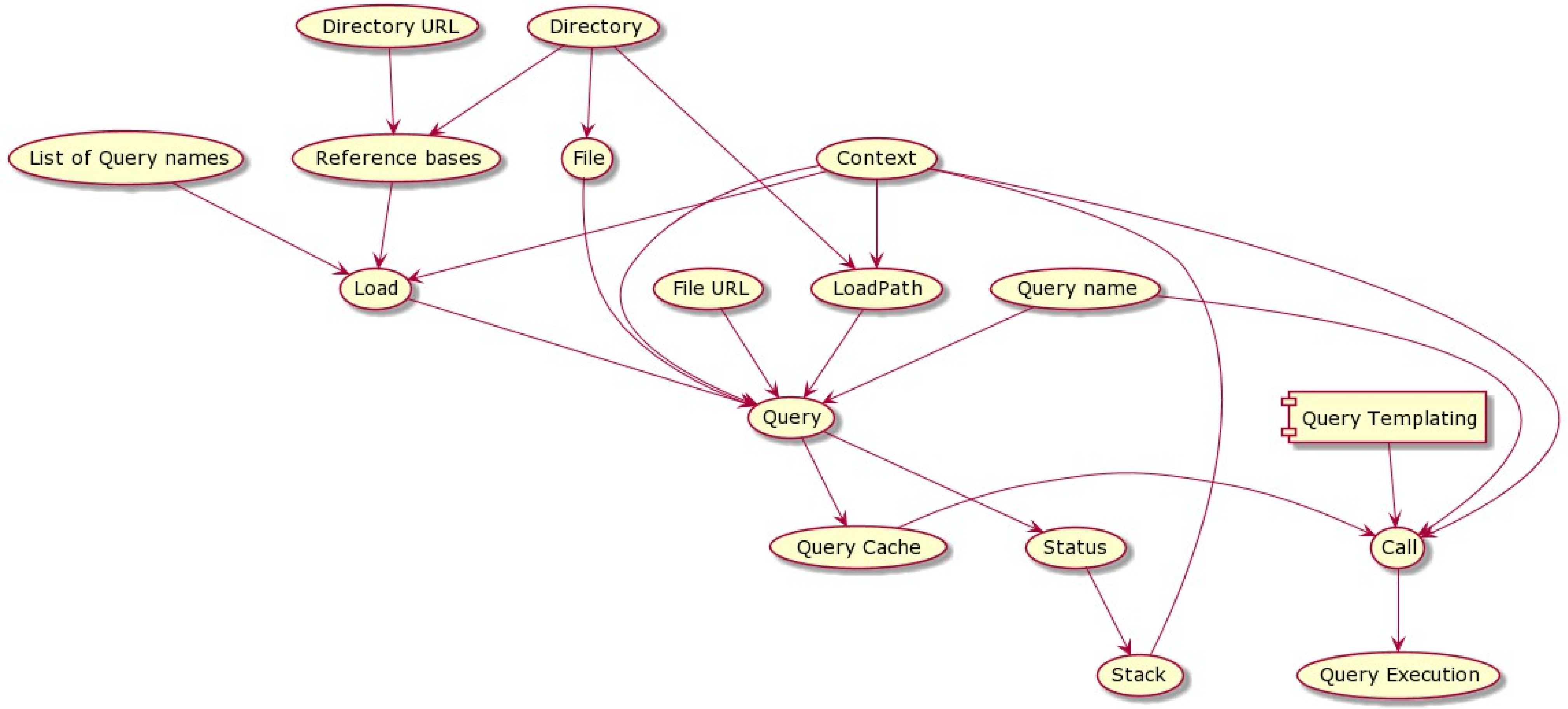
```
(F! {reference to the function} *{positional arguments} **{keyword arguments})
```

```
(ZBX)
    (Value Version.Version)
(F! )
```

ZABBIX

Subqueries

Subquery, that is the ZQL Query, that you are calling within the existing context of your current query. Before you can use that query, you shall load it in the special Query cache. Query cache is a Dictionary, and the elements of the Query Cache are never expired. It is a several words which will help you to deal with subqueries. Let me show you the relationship between those words and Query cache and Query execution.



ZABBIX

Subqueries

Although, this diagram looks complicated, but idea behind tha diagram is very simple. The cornerstone of the ZQL subquery mechanism is word (Query ...)

```
(Query {query name} {query full reference})
```

This query will take a content of the query, referred by the reference(*) and load this query into ZQL Environment Query Cache under the name passed as first parameter. The Query could be ether the raw query, or Query Template. We will talk about Query Templates in a few minutes.

```
(Query “GetAllHosts” “@http://www.example.com/zql/GetAllHosts.zql”)
```

In this example, ZQL will download the query from the specific URL and save it in the Query Cache under name “GetAllHosts”.

(*) The references are the strings, representing the location of the object. If string prefixed with “+”, the rest of the string is a path to the local file. If reference prefixed with “@”, the rest of the string is URL.

Subqueries

Although loading queries through the (Query …) word is possible, but it is will make your code very cluttered. We want something more convenient. And here it is:

```
(Load *{query names} ...)
```

This query will take a list of the Query names, passed as positional parameters and will try to load them from the list of the base references, pre-loaded into your environment. For more details about “reference bases”, look in the “ZQL command-line tool” documentation for the information about parameter –ref-base.

```
(Load “GetAllHosts” )
```

In this example, ZQL scan the list of the “reference bases”. For the existence of the query with that name and load it and install it into a Query Cache.

ZABBIX

Subqueries

(Load …) is very powerful when it comes to the loading the number of the queries from the "reference bases", but sometimes, we are having pretty simple situation, when we do have a local directory, filled with the "*.zql" files and we do need to load all of them. The word (LoadPath …) will help you to do that.

```
(LoadPath *{list of the directories} **{keywords})
```

First, please note, we are passing the list of the directory path's, not the list of the references. So, you do not have to add "+" prefix. Next, each path, can contain a macro, which will be expanded with the value passed through the keywords.

```
(LoadPath
    "$ZQ_LIB/zql"
    "$LOCAL/etc/zql"
    "/var/data/zql"
      :ZQ_LIB "/usr/local/lib/zq"
      :LOCAL "/usr/local"
)
```

ZABBIX

Subqueries

So far, we've discussed how to load the query into a Query Cache. But query sitting in the cache isn't very useful for us. Not very useful, at all…. In order to call the query which was previously stored in the Cache, you shall use the word (Call ….)

```
(Call {Query Name} **{keywords})
```

The first parameter is a name of the Query which were loaded in the Cache. Since the query itself could be a query template, the keywords will provide the values for the macro expansion. Let say, we have the loaded query which looks like this:

```
(ZBX) (Hosts :hostid $HOSTID) (Out)
```

As you see, this Query, instead of value, refering the Macro name $HOSTID. Let assume, that this query will be loaded with the name GetHostByID

```
(Call "GetHostByID" :HOSTID 10084)
```

In this example, we will expand the Macro in the template with the name GetHostByID "on the fly" (means, Query Cache is unchanged) and execute the following Query:

```
(ZBX) (Hosts :hostid 10084) (Out)
```

ZABBIX

Yes, we do have GLOBALS

And it's counterpart, the “border word” - (Getv ...)

```
(Getv {variable name} *{keyword parameters})
```

I did used the definition “border word” for the (Getv …), because this word will return the value of the variable. Another behavior of which you shall be aware of, is that the Global Variable are “read once” by-default. Means, once (Getv …) returns value, Global Variable is removed. You can change that behavior and keep the variable by passing :keep True to the (Getv …) call.

Example:

```
(ZBX) (Setv :ANSWER 42) (Getv “ANSWER”)
```

ZABBIX

Words of logic

ZQL is not “your typical programming language”. While ZQL have a both, LISP and Forth features by design and it built on top of the Python, it is still “domain-specific language”. It is precisely, “a Query Language” where the problems of getting and processing the data are more important than feature richness for the algorithm's implementation.

There are two classes of the logic words:

The words operating with the “status” data pushed to the Stack. Those words are (Status …) (IfTrue …) (IfFalse ...)

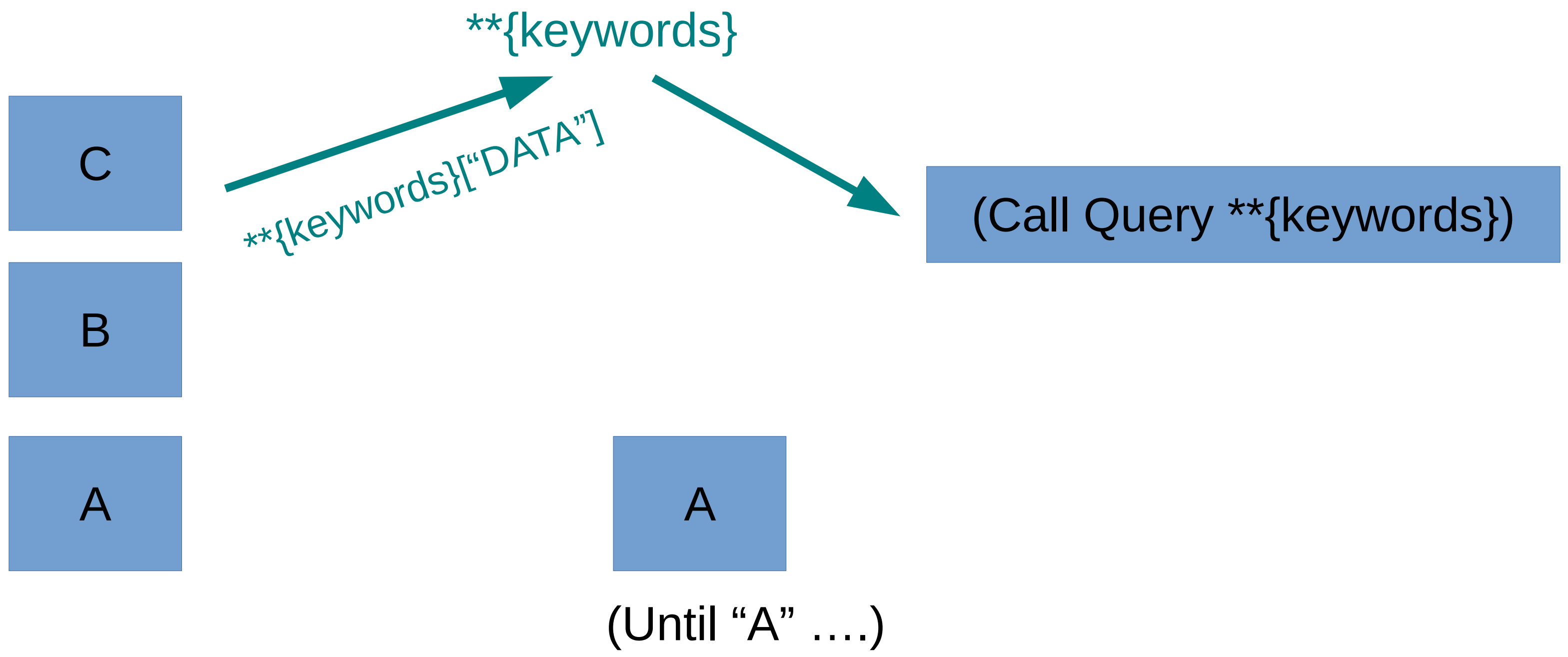
And the the conditional word, performing operation with the data stored in the Stack. And this word is (Until ...)

ZABBIX

Words of logic

(Until {key} *{query names} **{keywords})

This word pulls the data from the stack, until the key of the data matches the key passed as first parameter. Then it is trying to execute (Call …) over the list of the pre-loaded queries, and passing the keyword parameters to the (Call …) The data from the Stack is passed to the key “DATA” of the keywords.

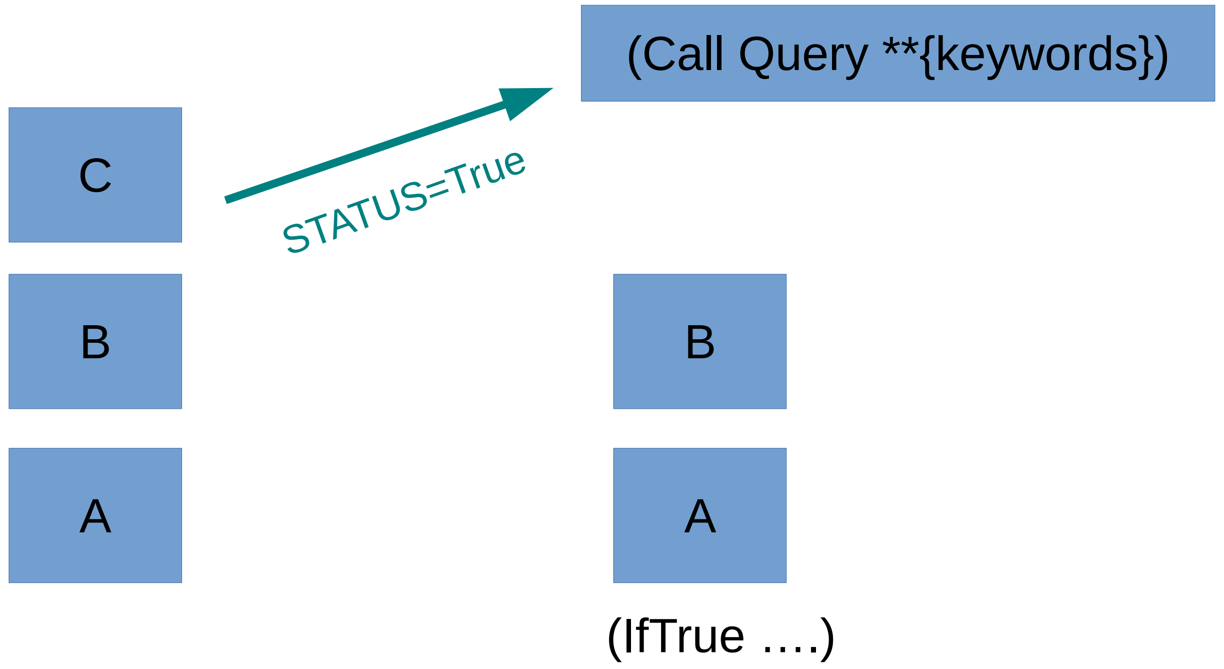


ZABBIX

Words of logic

(IfTrue *{query names} **{keywords})

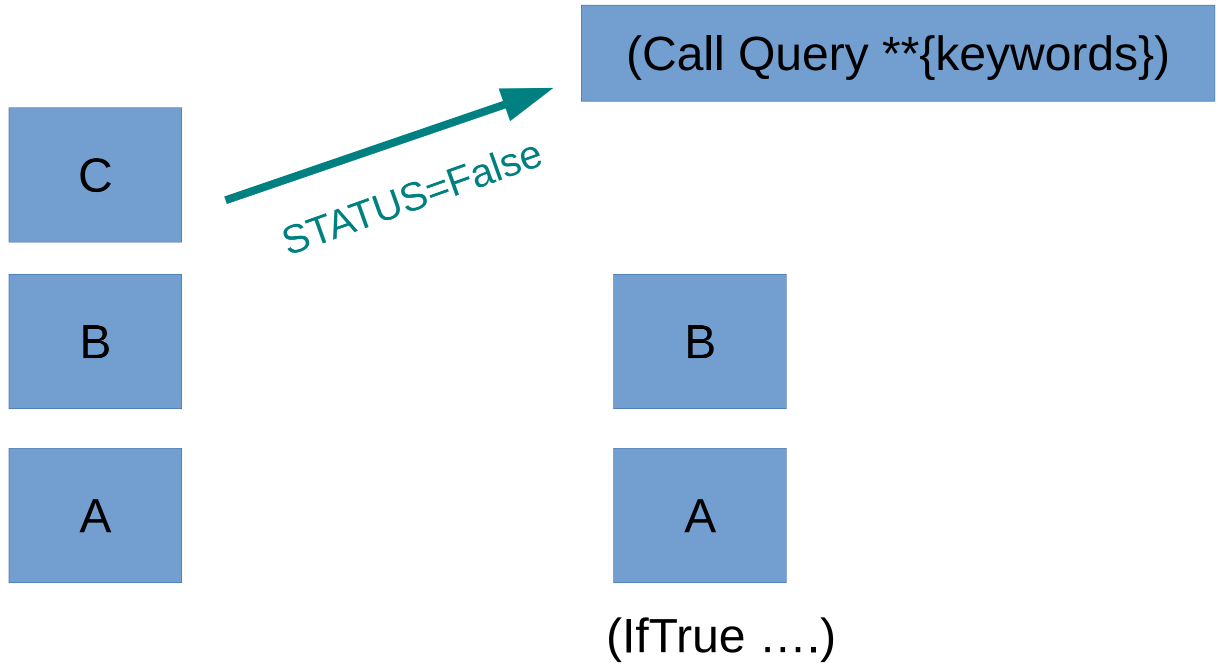
This word pulls the data from the stack, and if the data element is a “STATUS” and the value of the “STATUS” is True, this word will execute list of the queries using (Call …) word. The **{keywords} will be passed to the (Call ...)



Words of logic

(IfFalse *{query names} **{keywords})

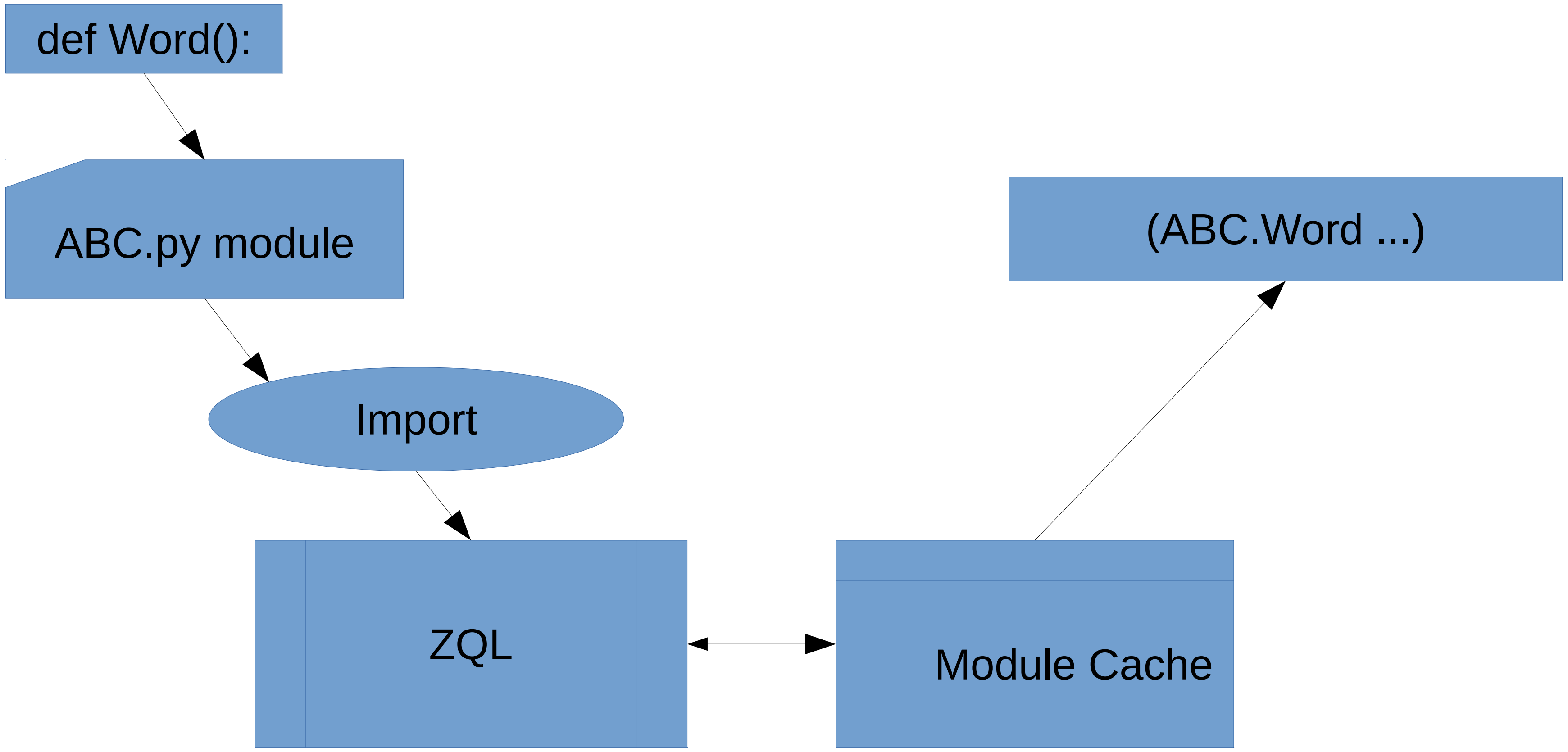
This word pulls the data from the stack, and if the data element is a “STATUS” and the value of the “STATUS” is False, this word will execute list of the queries using (Call …) word. The **{keywords} will be passed to the (Call ...)



ZABBIX

Extending ZQL

Beginning version 0.5, you can write your own extension modules for ZQL. Your first question is: How I can do it ? Simple, ZQL is built on top of the Python, so, you are writing a normal Python module with few ZQL specific variables, then you can load your module into a ZQL, and the functions in this module will become a part of the ZQL dictionary.



ZABBIX

Extending ZQL

The only word that you'll ever need for loading external modules into ZQL is:

```
(Import *{list of the names of the modules})
```

You do need to pass the list of the reference bases for the module loading. Those reference bases can be ether URL's or local directories. ZQL will cache them in-memory and into a disk-based module cache. This will save you some network traffic. Please refer to the “ZQL command-line tool” manual for the full list of the keyword parameters for working with the modules reference list and module cache.

Example:

```
(Import “Demo”)
```

Then you can call the functions declared in your module like:

```
(Demo.PrintArg “Hello” 3.14 :answer 42)
```

ZABBIX

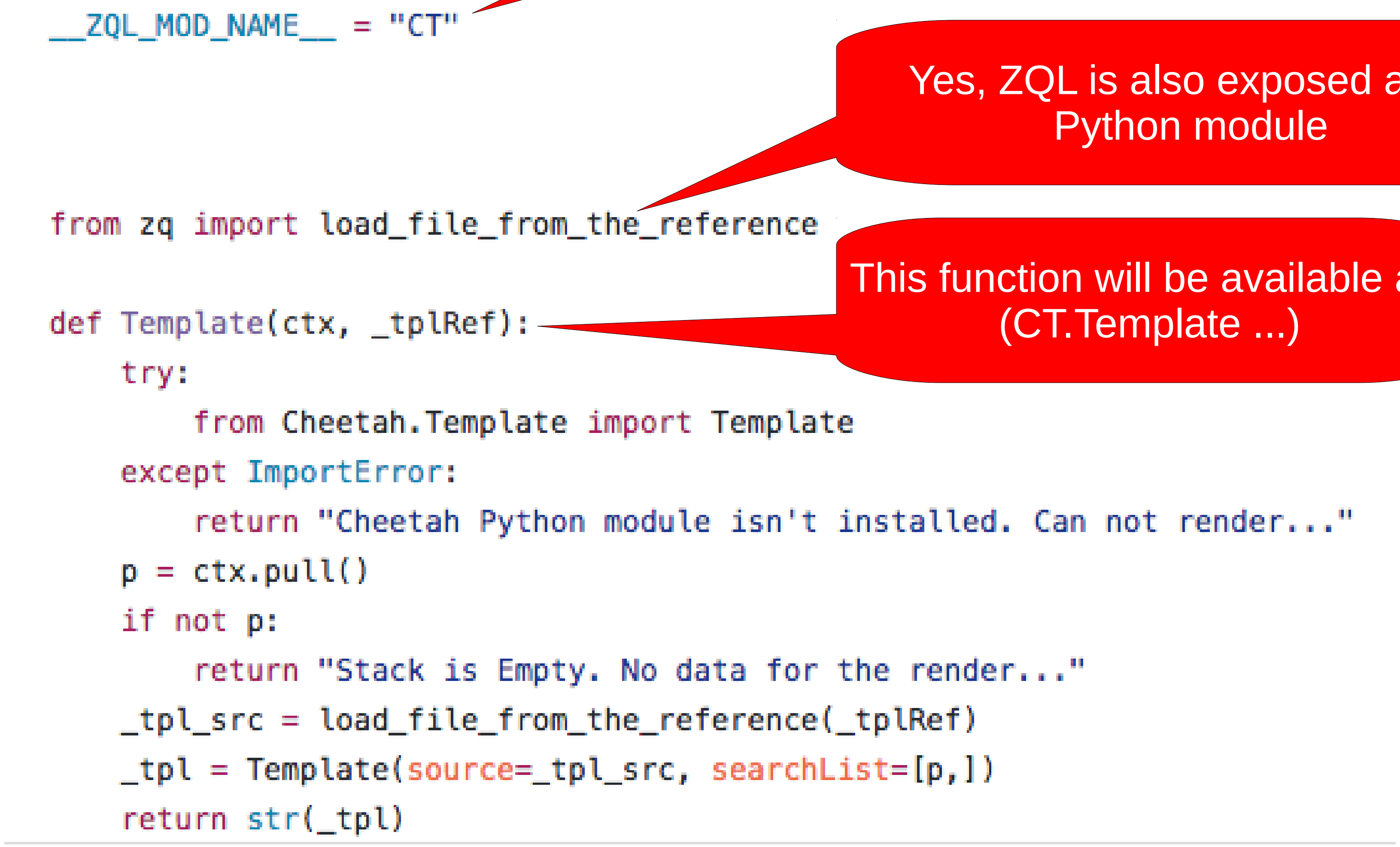
Extending ZQL

Anatomy of the module:

This is a special module variable.
This one is setting the ZQL function namespace.

Yes, ZQL is also exposed as a Python module

This function will be available as (CT.Template ...)



```
__ZQL_MOD_NAME__ = "CT"



from zq import load_file_from_the_reference

def Template(ctx, _tplRef):
    try:
        from Cheetah.Template import Template
    except ImportError:
        return "Cheetah Python module isn't installed. Can not render..."
    p = ctx.pull()
    if not p:
        return "Stack is Empty. No data for the render..."
    _tpl_src = load_file_from_the_reference(_tplRef)
    _tpl = Template(source=_tpl_src, searchList=[p,])
    return str(_tpl)
```

ZABBIX

Extending ZQL

Because the architecture of the Hy engine, once the sequence is executed, change of the globals is permitted, but it is not recognized by the parser. This means, that the query:

```
(ZBX) (Import “Version”) (Version.Version)
```

will fail. So, you have to do your imports during the bootstrap process or in different queries. Because the Environment in ZQL is state-full, those two separate queries will achieve desired result:

```
(ZBX) (Import “Version)

(ZBX) (Version.Version)
```

ZABBIX

Environment configuration

Certain aspects of ZQL is controlled through ZQL context configuration. There are several words which can help you to deal with the context configuration. You can get individual value, push the value to the stack, set the configuration value and so on.

```
(Cfg *{pattern})
```

The most basic word for reading the configuration, is (Cfg …). It accepts the list of patterns and selects the key→ value pairs from the configuration, where key is matched to at least one pattern. The discovered key→value pairs will be stored to the list and this list will be pushed to the Stack with the Key "CFG"

Example:

```
(ZBX) (Cfg "*") (Out) (Pretty_Json)
```

This query will push your configuration to the Stack and then push it out and pretty-print it.

ZABBIX

Environment configuration

Second configuration retrieval word is (Cfg! …). Unlike the previous one, this is a “border” word. Means it is returning not a context but the value. And therefore, it is accepting only one parameter: the key that you are looking in configuration.

```
(Cfg! {configuration parameter name})
```

Example:

```
(ZBX) (Cfg! “ZQ_MODULE_PATH”)
```

This query will return the value of the configuration parameter, referred by the specific name.

ZABBIX

Environment configuration

Also, you can change your configuration parameter, using the word (Cfg→ ...)

```
(Cfg-> {configuration parameter name} {value})
```

Example:

```
(ZBX) (Cfg-> “ZQ_MODULE_PATH” [“.”])
```

This query will change the single configuration parameter.

ZABBIX

Environment configuration

Configuration Parameter	Description
ZQ_MAX_PIPELINE	Maximum number of the elements in the Stack
ZQ_MAX_ENV_STACK	Maximum number of the elements in the environment stack
ZQ_REF_BASE	List of the reference bases, for the (Load ...)
ZQ_MODCACHEPATH	Location of the “on the disk” Modules Cache
ZQ_MODCACHEEXPIRE	For how long you are permitting to the module to stay in Cache.
ZQ_HOME	Home directory for the ZQL
ZQ_ENV_NAME	Name of the default environment

ZABBIX

Environment configuration

In case, if you will ever want to change multiple configuration parameters in the single call, you can use the word (Cfg* ...)

```
(Cfg* **{keyword parameters})
```

Using keyword parameters, you can pass the values to the multiple configuration keys.

Example:

```
(ZBX) (Cfg* :ZQ_MODULE_PATH [“.”] :ZQ_REFRESH “1d”)
```

This query will set multiple parameters in the one call.

From a Context to the Data

(Out)

(Out) is a one of the “border crossing” word in ZQL. All it’s doing, is pull the single element from the top of the Stack and return it as the value. In ZQL pipeline all words after (Out) will be manipulating with the reference to the data, returned by that “border word”, not with the context.

Example query:

(ZBX) (Hosts) (Out)

The word (Hosts) will send an Zabbix API call to the Zabbix server, which context is returned by (ZBX), word, then push the result of this query to the stack. Word (Out) will pull data from the stack and as it is no more further processing, the result of this query will be returned as the result of this query.

(Empty)

Another somewhat useful “border-crossing” word is (Empty). This word will pull all elements from the Stack and will return them as a List. I am still contemplating on how it could be useful in the production queries, but it is very useful for debugging complex queries.

Example query:

```
(ZBX) (Hosts) (Hostgroups) (Empty)
```

Here we will get two elements in the Stack with the data about Hosts and the Host Groups. Word (Empty) will pull all data from the stack and return them as a list. Since, this is a LIFO Stack, the first element of the List will be result of the (Hostgroups), and a second, will be result from the (Hosts)

(Pretty_Json)

Sometimes, we do need to do the pretty formatting of the JSON data returned by ZAPI (and ZQL). For that, we do have the “word” (Pretty_Json). This word in the the pipeline will take data on the input, provided by (Out) and if incoming data is a Dictionary, it will format it in a human-pleasing JSON format.

Example query:

```
(ZBX) (Hosts) (Out) (Pretty_Json)
```

The word (Hosts) will send an Zabbix API call to the Zabbix server, which context is returned by (ZBX), word, then push the result of this query to the Stack. Word (Out) will pull data from the Stack and pass it to (Pretty_Json). The word (Pretty_Json) will convert data into JSON and will reformat it will return it as result of that query.

ZABBIX

Final note

So, I am welcoming you to the world where you can query and change your Zabbix configuration data ether programmatically, from the ZQL Python module, or from your shell scripts. The world, in which you shall only have the “domain specific” knowledge – the knowledge of Zabbix.

ZQL will take care of the rest.

ZABBIX

質問

Q/A ?

Jautāt ?

Interroger ?

FRAGEN ?

ВОПРОСЫ ?

Vladimir Ulogov