3 自动发现□LLD□

概述

Low-level discovery provides a way to automatically create items, triggers, and graphs for different entities on a computer. For instance, Zabbix can automatically start monitoring file systems or network interfaces on your machine, without the need to create items for each file system or network interface manually. Additionally it is possible to configure Zabbix to remove unneeded entities automatically based on actual results of periodically performed discovery.

自动发现□LLD□提供了一种在计算机上为不同实体自动创建监控项，触发器和图形的方法。例如□Zabbix可以在你的机器上自动开始监控文件系统或网络接口，而无需为每个文件系统或网络接口手动创建监控项。此外，可以配置Zabbix根据定期执行发现后的得到实际结果，来移除不需要的监控项。

In Zabbix, six types of discovery items are supported out of the box:

在Zabbix中，支持六种类型的发现项目：

- discovery of file systems;
- 系统文件的发现；
- discovery of network interfaces;
- 网络接口的发现；
- discovery of CPUs and CPU cores;
- CPU和CPU内核的发现
- discovery of SNMP OIDs;
- SNMP OID的发现
- discovery using ODBC SQL queries;
- 使用ODBC SQL查询的发现
- discovery of Windows services.
- Windows服务的发现

A user can define their own types of discovery, provided they follow a particular JSON protocol.

用户可以自己定义发现类型，只要它们遵循特定的JSON协议。

The general architecture of the discovery process is as follows.

发现过程的一般架构如下。

First, a user creates a discovery rule in “Configuration” → “Templates” → “Discovery” column. A discovery rule consists of (1) an item that discovers the necessary entities (for instance, file systems or network interfaces) and (2) prototypes of items, triggers, and graphs that should be created based on the value of that item.

首先，用户在“配置”→“模板”→“发现”列中创建一个发现规则。发现规则包括（1）发现必要实体（例如，文件系统或网络接口）的项目和（2）应该根据该项目的值创建的监控项，触发器和图形的原型

An item that discovers the necessary entities is like a regular item seen elsewhere: the server asks a Zabbix agent (or whatever the type of the item is set to) for a value of that item, the agent responds with a textual value. The difference is that the value the agent responds with should contain a list of discovered entities in a specific JSON format. While the details of this format are only important for implementers of custom discovery checks, it is necessary to know that the returned value contains a

list of macro → value pairs. For instance, item “net.if.discovery” might return two pairs: “{#IFNAME}” → “lo” and “{#IFNAME}” → “eth0”.

发现必要实体的项目就像其他地方所看到的常规项目：服务器向该项目的值询问Zabbix agent或者该项目的任何类型的设置agent以文本值进行响应。区别在于agent响应的值应该包含特定JSON格式的发现实体的列表。这种格式的自定义检查者发现的细节才是最重要的，因为返回值必须包含宏→值对。例如，项目“net.if.discovery”可能会返回两对键值“{#IFNAME}”→“lo”和 “{#IFNAME}”→“eth0”

Low-level discovery items “vfs.fs.discovery” and “net.if.discovery” are supported since Zabbix agent version 2.0.
Discovery item “system.cpu.discovery” is supported since Zabbix agent version 2.4.
Discovery of SNMP OIDs is supported since Zabbix server and proxy version 2.0.
Discovery using ODBC SQL queries is supported since Zabbix server and proxy version 3.0.
Zabbix agent版本2.0支持自动发现项目“vfs.fs.discovery”和 “net.if.discovery”
从Zabbix agent版本2.4起支持发现项目“system.cpu.discovery”
从Zabbix server和proxy版本2.0起支持发现SNMP OID
从Zabbix server和proxy版本3.0起支持使用ODBC SQL查询的发现。
Return values of a low-level discovery rule are limited to 2048 bytes on a Zabbix proxy run with IBM DB2 database. This limit does not apply to Zabbix server as return values are processed without being stored in a database.
在使用IBM DB2数据库运行的Zabbix proxy上，自动发现规则的返回值限制为2048字节。此限制不适用于Zabbix server因为返回值不会被存储在数据库中。

These macros are used in names, keys and other prototype fields where they are then substituted with the received values for creating real items, triggers, graphs or even hosts for each discovered entity. See the full list of options for using LLD macros.

这些宏用于名称，键值和其他原型字段中，然后用接收到的值为每个发现的实体创建实际的监控项，触发器，图形甚至主机。请参阅使用LLD宏的选项的完整列表。

When the server receives a value for a discovery item, it looks at the macro → value pairs and for each pair generates real items, triggers, and graphs, based on their prototypes. In the example with “net.if.discovery” above, the server would generate one set of items, triggers, and graphs for the loopback interface “lo”, and another set for interface “eth0”.

当服务器接收到发现项目的值时，它会查看宏→值对，每对都根据原型生成实际监控项，触发器和图形。在上面的“net.if.discovery”示例中，服务器将生成环路接口“lo”的一组监控项，触发器和图表，另一组用于界面“eth0”

The following sections illustrate the process described above in detail and serve as a how-to for performing all types of discovery mentioned above. The last section describes the JSON format for discovery items and gives an example of how to implement your own file system discoverer as a Perl script.

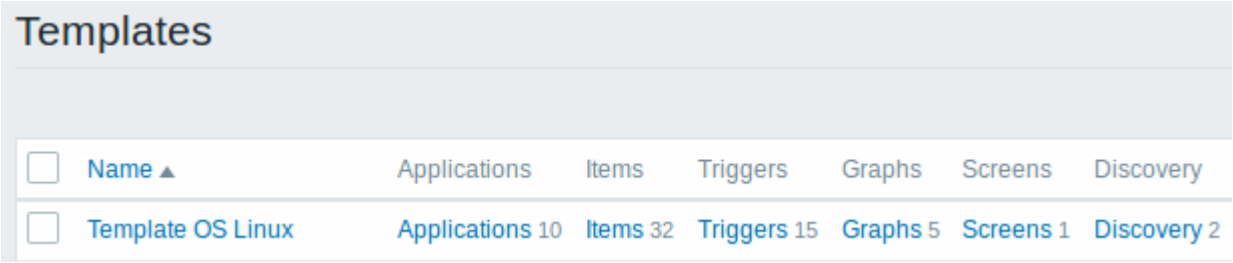
以下部分将详细说明上述过程，并作为一个指导上述类型的所有发现。最后一节描述了发现项目的JSON格式，并给出了文件系统发现实现的Perl脚本的示例。

3.1 文件系统的发现

To configure the discovery of file systems, do the following:

要配置文件系统的发现，请执行以下操作：

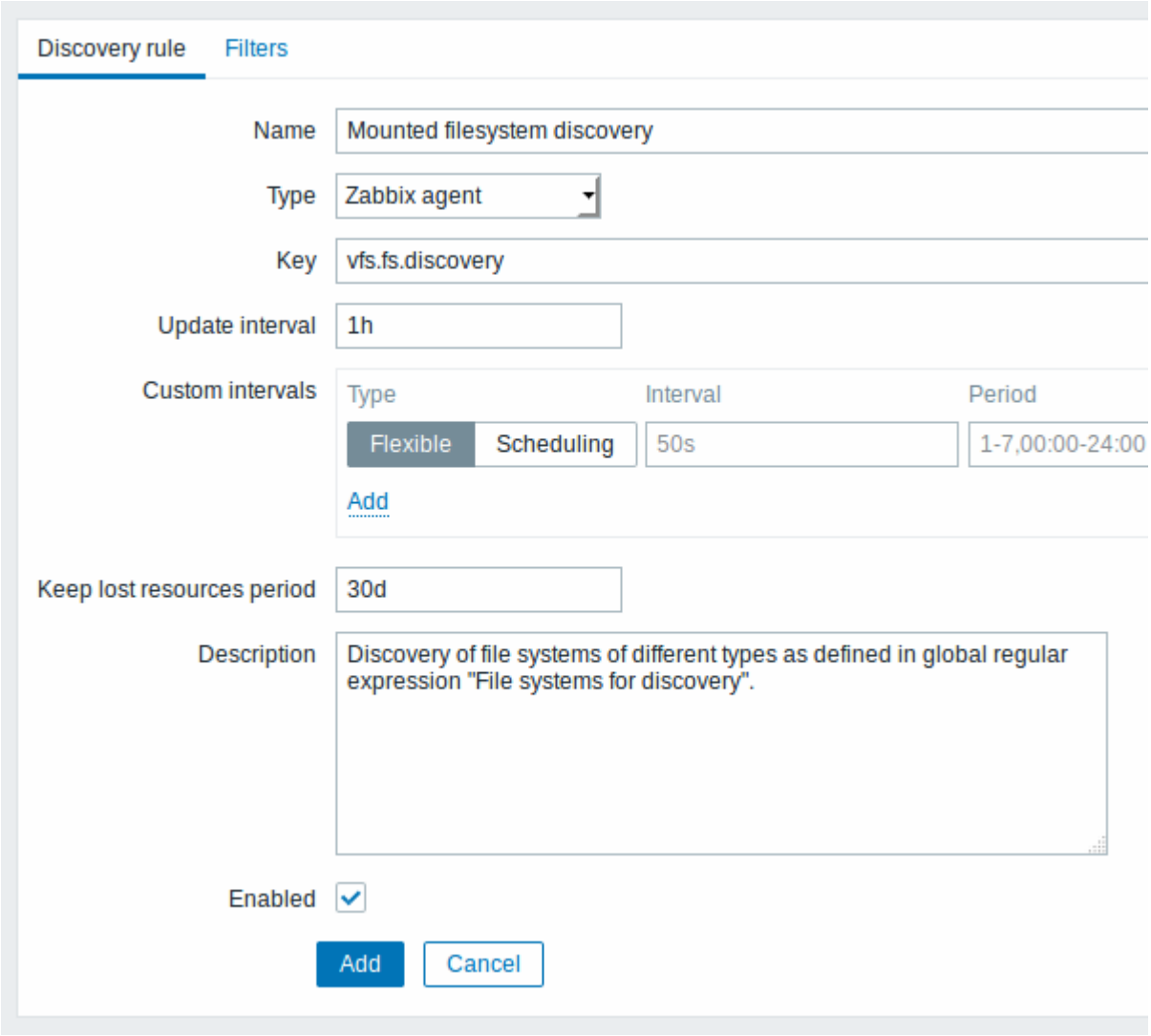
- Go to: *Configuration* → *Templates*
- 转到：*配置* → *模板*
- Click on *Discovery* in the row of an appropriate template
- 在一个合适的模板的行点击*发现*



- Click on *Create discovery rule* in the upper right corner of the screen
- 单击屏幕右上角的*创建发现规则*
- Fill in the form with the following details
- 填写以下详细信息。

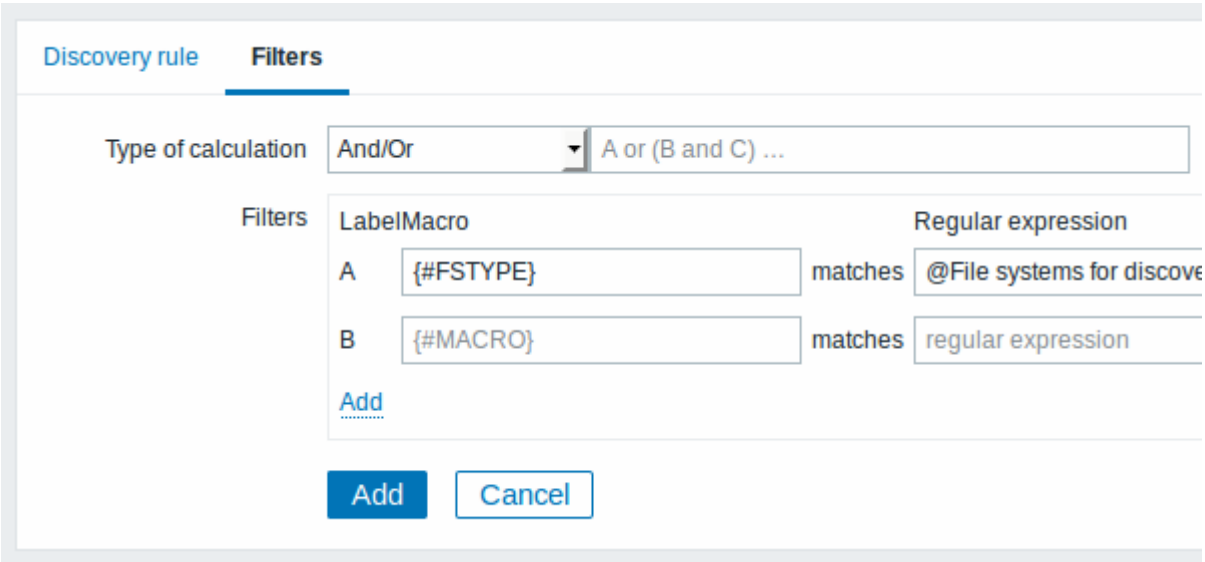
The **Discovery rule** tab contains general discovery rule attributes:

发现规则选项卡包含常规发现规则属性：



参数	描述
Name	Name of discovery rule.
Type	The type of check to perform discovery; should be *Zabbix agent* or *Zabbix agent (active)* for file system discovery.
Key	An item with “vfs.fs.discovery” key is built into the Zabbix agent on many platforms (see supported item key list for details), and will return a JSON with the list of file systems present on the computer and their types.
Update interval (in sec)	This field specifies how often Zabbix performs discovery. In the beginning, when you are just setting up file system discovery, you might wish to set it to a small interval, but once you know it works you can set it to 30 minutes or more, because file systems usually do not change very often. *Note*: If set to '0', the item will not be polled. However, if a flexible interval also exists with a non-zero value, the item will be polled during the flexible interval duration.
Custom intervals	You can create custom rules for checking the item: **Flexible** - create an exception to the *Update interval* (interval with different frequency) **Scheduling** - create a custom polling schedule. For detailed information see Custom intervals. Scheduling is supported since Zabix 3.0.0.
Keep lost resources period (in days)	This field allows you to specify for how many days the discovered entity will be retained (won't be deleted) once its discovery status becomes “Not discovered anymore” (max 3650 days). *Note:* If set to “0”, entities will be deleted immediately. Using “0” is not recommended, since just wrongly editing the filter may end up in the entity being deleted with all the historical data.
Description	Enter a description.
Enabled	If checked, the rule will be processed.

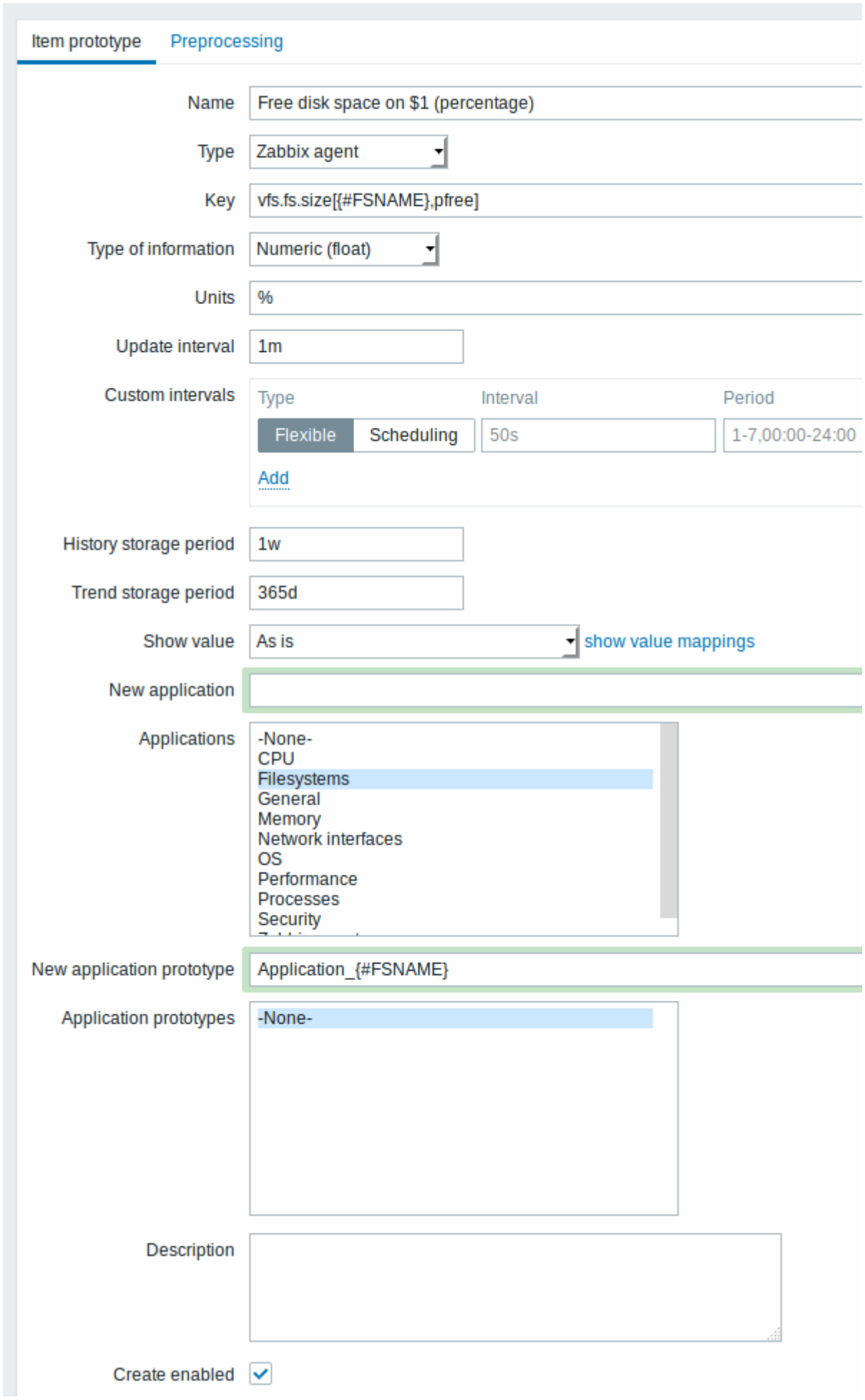
The **Filters** tab contains discovery rule filter definitions:



Parameter	Description
Type of calculation	The following options for calculating filters are available: **And** - all filters must be passed; **Or** - enough if one filter is passed; **And/Or** - uses *And* with different macro names and *Or* with the same macro name; **Custom expression** - offers the possibility to define a custom calculation of filters. The formula must include all filters in the list. Limited to 255 symbols.
Filters	A filter can be used to generate real items, triggers, and graphs only for certain file systems. It expects a POSIX Extended Regular Expression. For instance, if you are only interested in C:, D:, and E: file systems, you could put {#FSNAME} into “Macro” and "^C|^D|^E" regular expression into “Regular expression” text fields. Filtering is also possible by file system types using {#FSTYPE} macro (e.g. "^ext|^reiserfs") and by drive types (supported only by Windows agent) using {#FSDRIVETYPE} macro (e.g., "fixed"). You can enter a regular expression or reference a global regular expression in “Regular expression” field. In order to test a regular expression you can use “grep -E”, for example: `for f in ext2 nfs reiserfs smbfs; do echo $f | grep -E '^ext|^reiserfs' || echo "SKIP: $f"; done` {#FSDRIVETYPE} macro on Windows is supported since Zabbix **3.0.0**. Defining several filters is supported since Zabbix **2.4.0**. Note that if some macro from the filter is missing in the response, the found entity will be ignored.

Zabbix database in MySQL must be created as case-sensitive if file system names that differ only by case are to be discovered correctly.
Discovery rule history is not preserved.

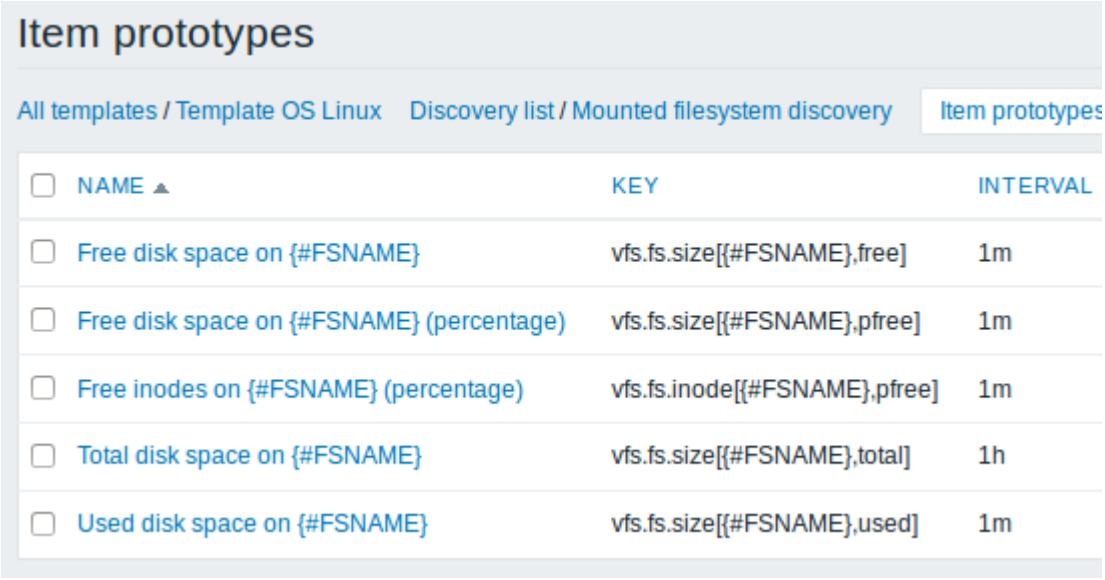
Once a rule is created, go to the items for that rule and press “Create prototype” to create an item prototype. Note how macro {#FSNAME} is used where a file system name is required. When the discovery rule is processed, this macro will be substituted with the discovered file system.


Item prototype
Preprocessing
Name
Free disk space on $1 (percentage)
Type
Zabbix agent
Key
vfs.fs.size[{#FSNAME},pfree]
Type of information
Numeric (float)
Units
%
Update interval
1m
Custom intervals
Type
Interval
Period
Flexible
Scheduling
50s
1-7,00:00-24:00
Add
History storage period
1w
Trend storage period
365d
Show value
As is
show value mappings
New application
Applications
-None-
CPU
Filesystems
General
Memory
Network interfaces
OS
Performance
Processes
Security
New application prototype
Application_{#FSNAME}
Application prototypes
-None-
Description
Create enabled

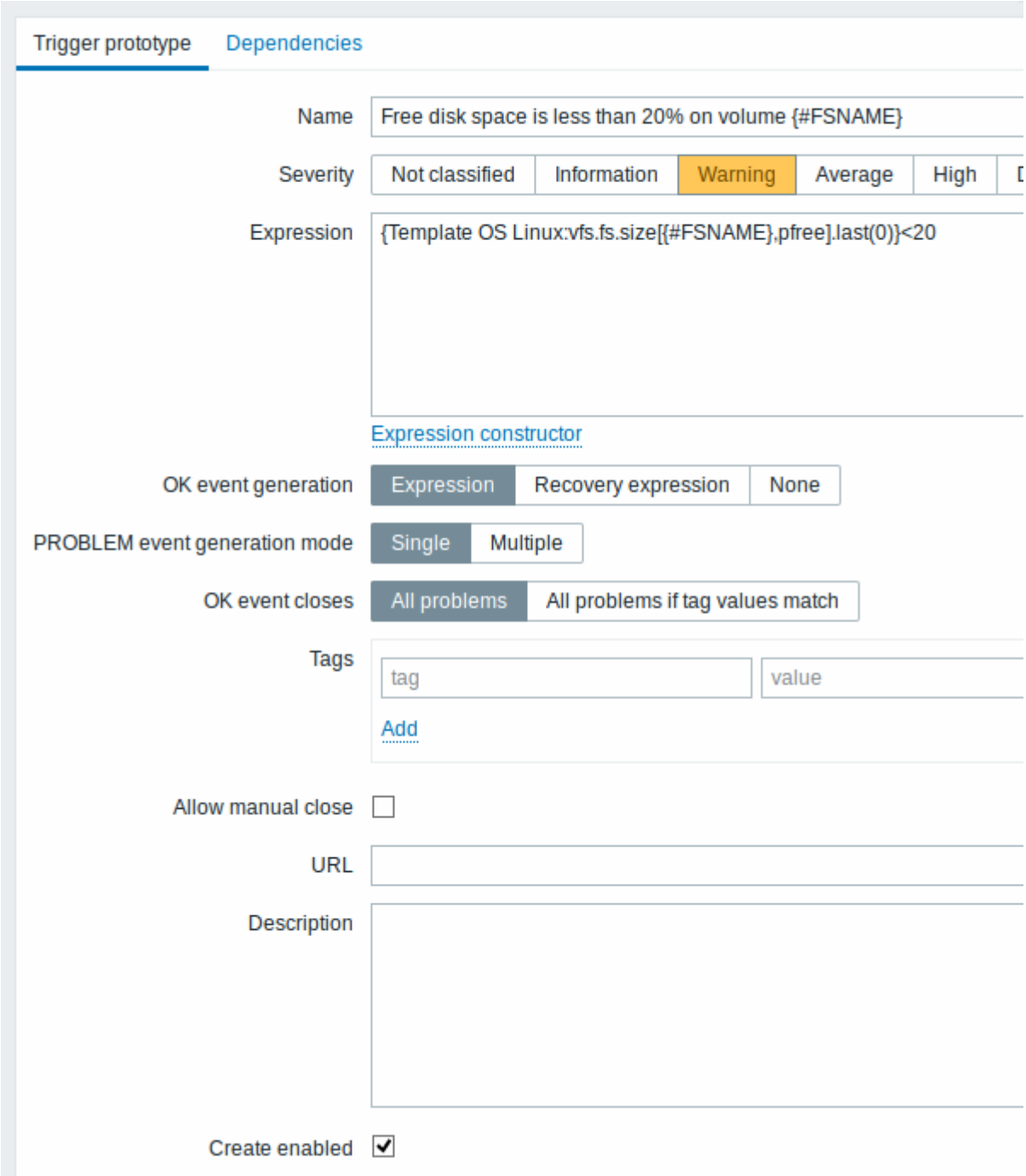
Attributes that are specific for item prototypes:

Parameter	**Description**
New application prototype	You may define a new application prototype. In application prototypes you can use low-level discovery macros that, after discovery, will be substituted with real values to create applications that are specific for the discovered entity. See also application discovery notes for more specific information.
Application prototypes	Select from the existing application prototypes.
Create enabled	If checked the item will be added in an enabled state. If unchecked, the item will be added to a discovered entity, but in a disabled state.

We can create several item prototypes for each file system metric we are interested in:



Then, we create trigger prototypes in a similar way:

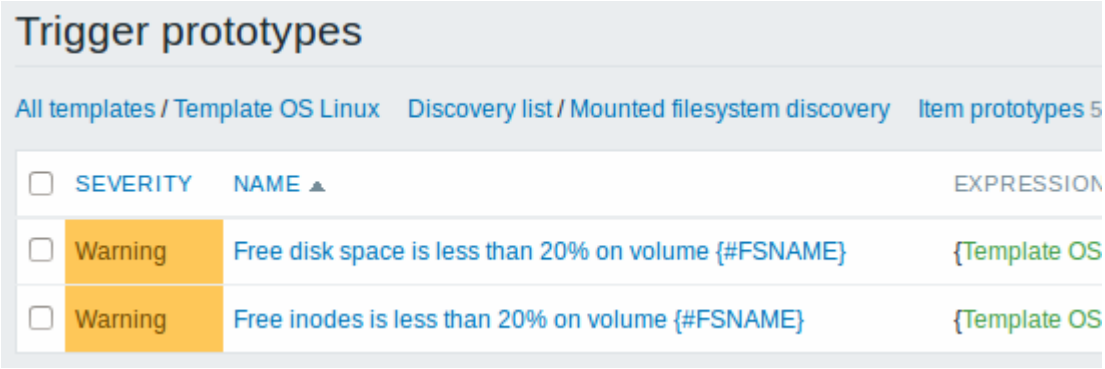


Attributes that are specific for trigger prototypes:

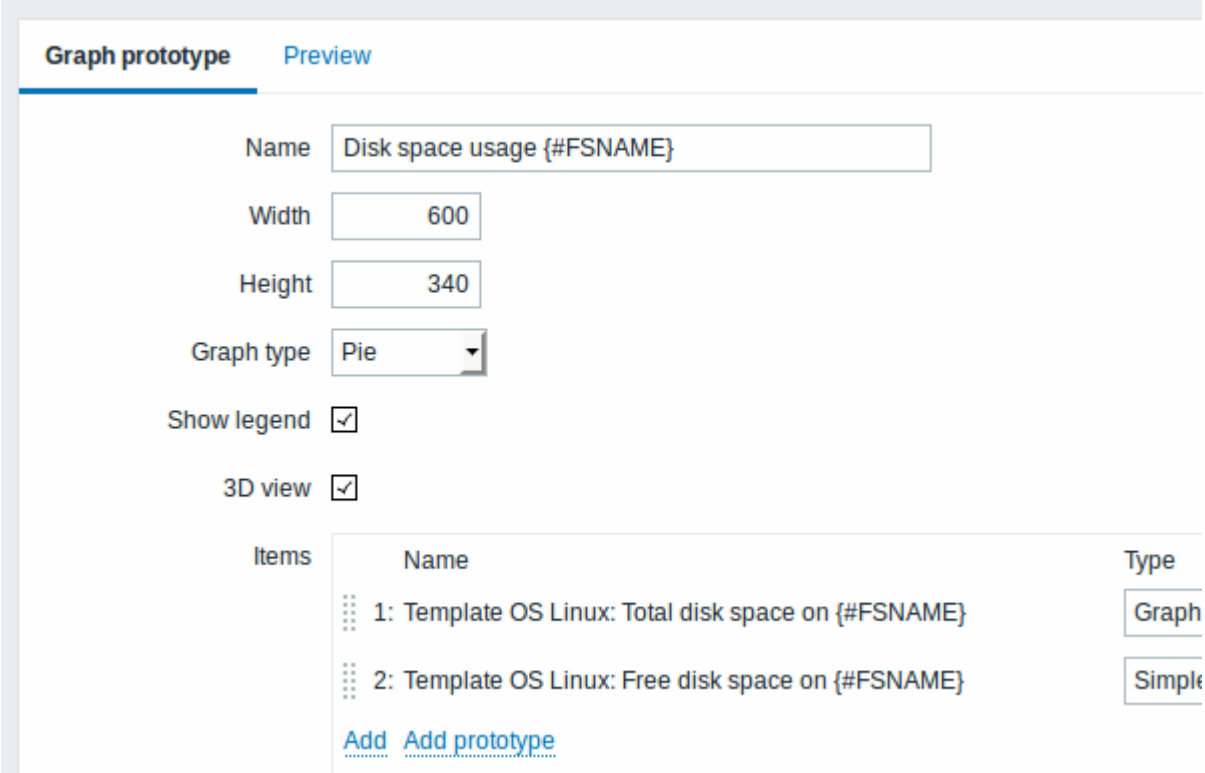
Parameter	Description
Create enabled	If checked the trigger will be added in an enabled state. If unchecked, the trigger will be added to a discovered entity, but in a disabled state.

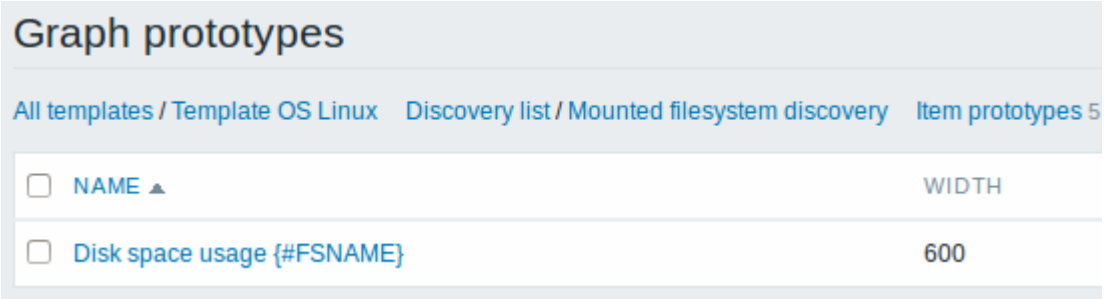
When real triggers are created from the prototypes, there may be a need to be flexible as to what constant ('20' in our example) is used for comparison in the expression. See how user macros with context can be useful to accomplish such flexibility.

You can define dependencies between trigger prototypes as well (supported since Zabbix 3.0). To do that, go to the *Dependencies* tab. A trigger prototype may depend on another trigger prototype from the same low-level discovery (LLD) rule or on a regular trigger. A trigger prototype may not depend on a trigger prototype from a different LLD rule or on a trigger created from trigger prototype. Host trigger prototype cannot depend on a trigger from a template.

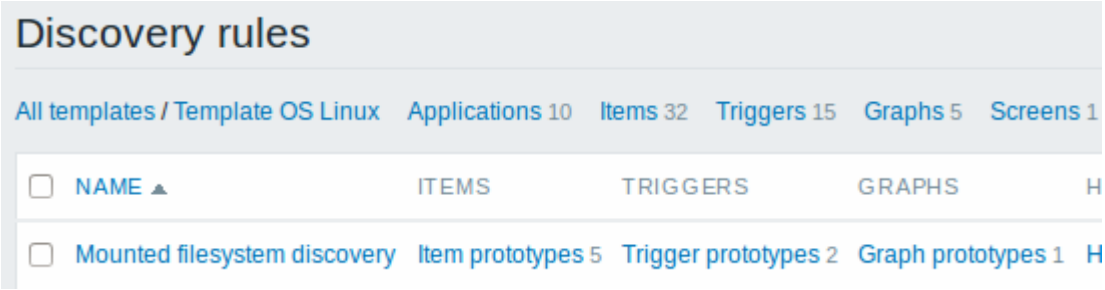


We can create graph prototypes, too:



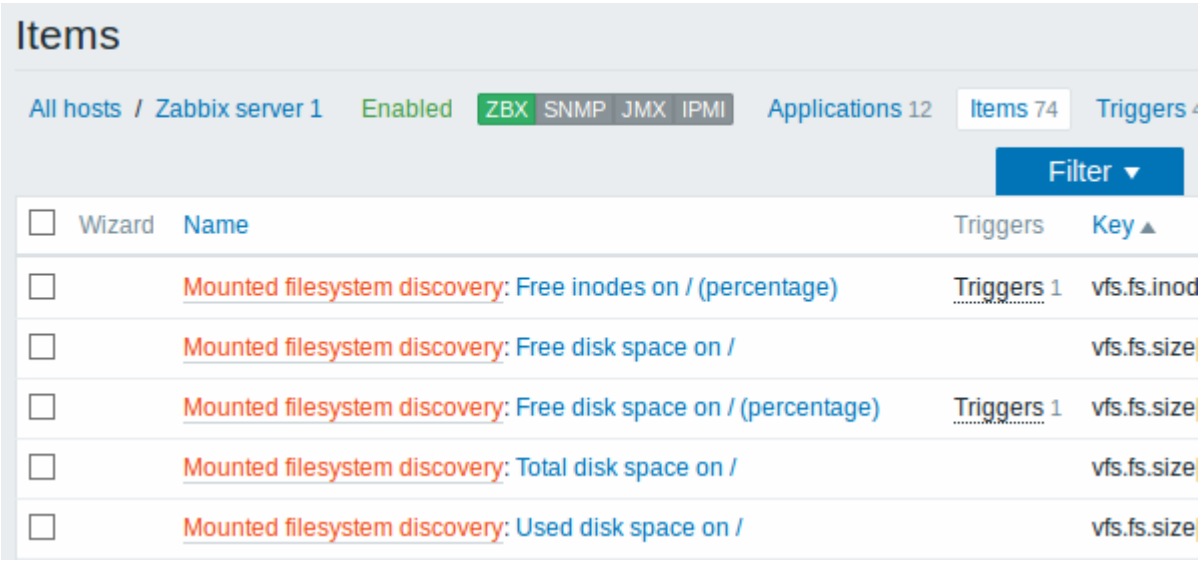


Finally, we have created a discovery rule that looks like shown below. It has five item prototypes, two trigger prototypes, and one graph prototype.



Note: For configuring host prototypes, see the section about host prototype configuration in virtual machine monitoring.

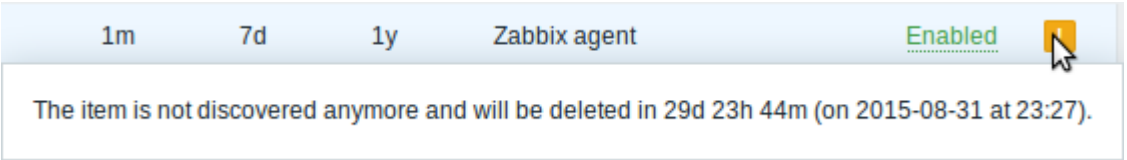
The screenshots below illustrate how discovered items, triggers, and graphs look like in the host's configuration. Discovered entities are prefixed with an orange link to a discovery rule they come from.



Note that discovered entities will not be created in case there are already existing entities with the same uniqueness criteria, for example, an item with the same key or graph with the same name.

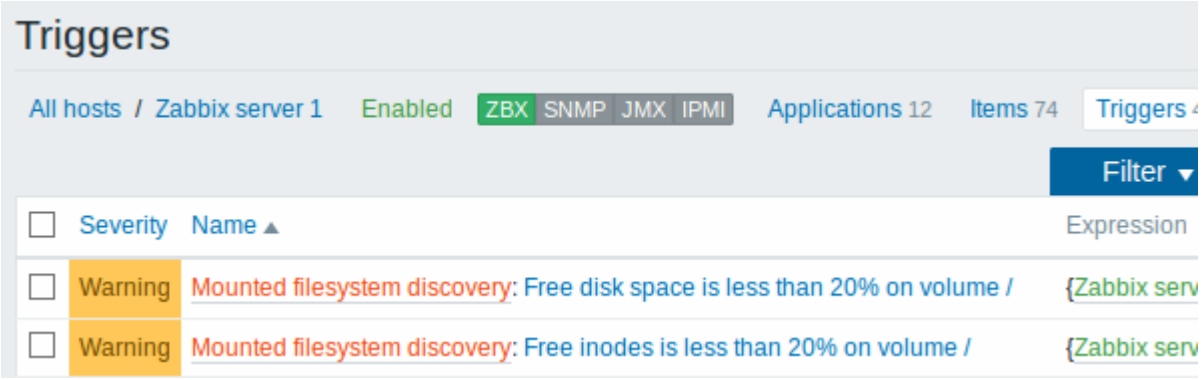
Items (similarly, triggers and graphs) created by a low-level discovery rule will be deleted automatically if a discovered entity (file system, interface, etc) stops being discovered (or does not pass the filter anymore). In this case the items, triggers and graphs will be deleted after the days defined in the *Keep lost resources period* field pass.

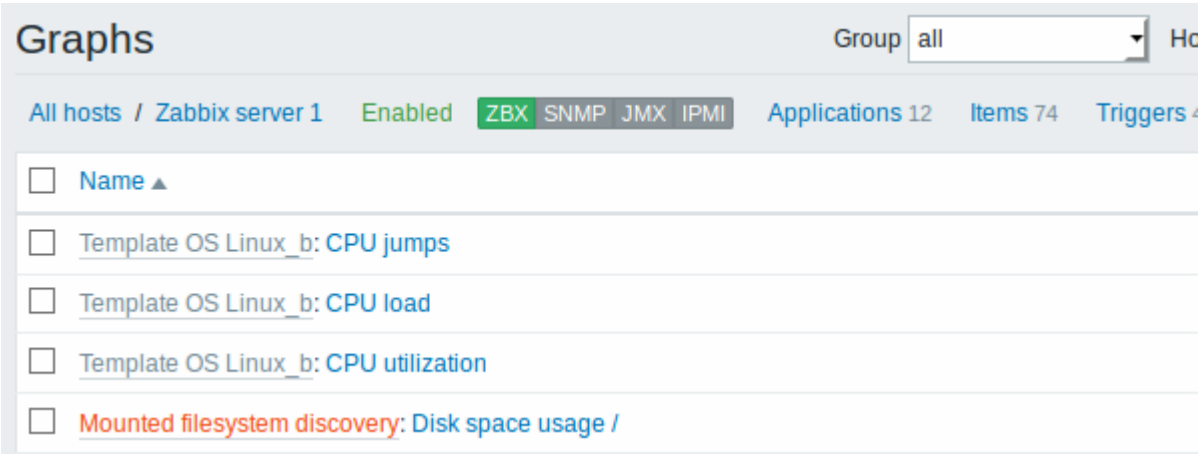
When discovered entities become 'Not discovered anymore', a lifetime indicator is displayed in the item list. Move your mouse pointer over it and a message will be displayed indicating how many days are left until the item is deleted.



If entities were marked for deletion, but were not deleted at the expected time (disabled discovery rule or item host), they will be deleted the next time the discovery rule is processed.

Entities containing other entities, which are marked for deletion, will not update if changed on the discovery rule level. For example, LLD-based triggers will not update if they contain items that are marked for deletion.





3.2 Discovery of network interfaces

Discovery of network interfaces is done in exactly the same way as discovery of file systems, except that you use the discovery rule key “net.if.discovery” instead of “vfs.fs.discovery” and use macro {#IFNAME} instead of {#FSNAME} in filter and item/trigger/graph prototypes.

Examples of item prototypes that you might wish to create based on “net.if.discovery”: “net.if.in[{#IFNAME},bytes]”, “net.if.out[{#IFNAME},bytes]”.

See above for more information about the filter.

3.3 Discovery of CPUs and CPU cores

Discovery of CPUs and CPU cores is done in a similar fashion as network interface discovery with the exception being that the discovery rule key is “system.cpu.discovery”. This discovery key returns two macros - {#CPU.NUMBER} and {#CPU.STATUS} identifying the CPU order number and status respectively. To note, a clear distinction cannot be made between actual, physical processors, cores and hyperthreads. {#CPU.STATUS} on Linux, UNIX and BSD systems returns the status of the processor, which can be either “online” or “offline”. On Windows systems, this same macro may represent a third value - “unknown” - which indicates that a processor has been detected, but no information has been collected for it yet.

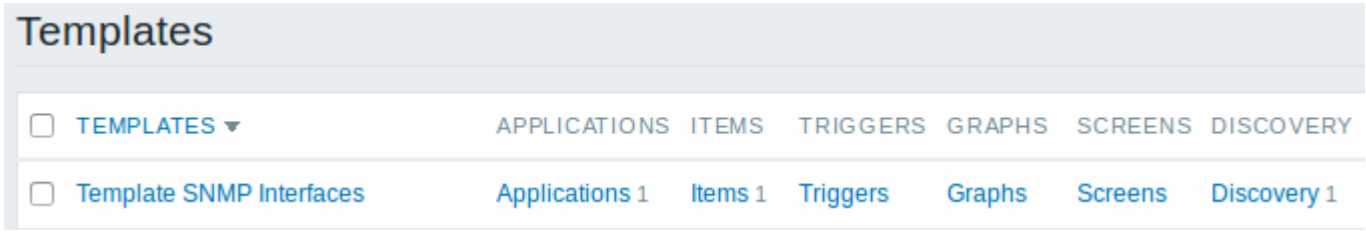
CPU discovery relies on the agent's collector process to remain consistent with the data provided by the collector and save resources on obtaining the data. This has the effect of this item key not

working with the test (-t) command line flag of the agent binary, which will return a NOT_SUPPORTED status and an accompanying message indicating that the collector process has not been started.

Item prototypes that can be created based on CPU discovery include, for example, “system.cpu.util[{#CPU.NUMBER}, <type>, <mode>]” or “system.hw.cpu[{#CPU.NUMBER}, <info>]”.

3.4 Discovery of SNMP OIDs

In this example, we will perform SNMP discovery on a switch. First, go to “Configuration” → “Templates”.



To edit discovery rules for a template, click on the link in the “Discovery” column.

Then, press “Create rule” and fill the form with the details in the screenshot below.

Unlike file system and network interface discovery, the item does not necessarily have to have “snmp.discovery” key - item type of SNMP agent is sufficient.

The OIDs to discover are defined in SNMP OID field in the following format: `discovery[{#MACRO1}, oid1, {#MACRO2}, oid2, …,]`

where *{#MACRO1}*, *{#MACRO2}* ... are valid lld macro names and *oid1*, *oid2*... are OIDs capable of generating meaningful values for these macros. A built-in macro *{#SNMPINDEX}* containing index of the discovered OID is applied to discovered entities. The discovered entities are grouped by *{#SNMPINDEX}* macro value.

To understand what we mean, let us perform few snmpwalks on our switch:

```
$ snmpwalk -v 2c -c public 192.168.1.1 IF-MIB::ifDescr
IF-MIB::ifDescr.1 = STRING: WAN
IF-MIB::ifDescr.2 = STRING: LAN1
IF-MIB::ifDescr.3 = STRING: LAN2

$ snmpwalk -v 2c -c public 192.168.1.1 IF-MIB::ifPhysAddress
IF-MIB::ifPhysAddress.1 = STRING: 8:0:27:90:7a:75
IF-MIB::ifPhysAddress.2 = STRING: 8:0:27:90:7a:76
IF-MIB::ifPhysAddress.3 = STRING: 8:0:27:2b:af:9e
```

And set SNMP OID to: `discovery[{#IFDESCR}, ifDescr, {#IFPHYSADDRESS}, ifPhysAddress]`

Now this rule will discover entities with {#IFDESCR} macros set to **WAN**, **LAN1** and **LAN2**, {#IFPHYSADDRESS} macros set to **8:0:27:90:7a:75**, **8:0:27:90:7a:76**, and **8:0:27:2b:af:9e**,

{#SNMPINDEX} macros set to the discovered OIDs indexes **1**, **2** and **3**:

```
{
    "data": [
        {
            "{#SNMPINDEX}": "1",
            "{#IFDESCR}": "WAN",
            "{#IFPHYSADDRESS}": "8:0:27:90:7a:75"
        },
        {
            "{#SNMPINDEX}": "2",
            "{#IFDESCR}": "LAN1",
            "{#IFPHYSADDRESS}": "8:0:27:90:7a:76"
        },
        {
            "{#SNMPINDEX}": "3",
            "{#IFDESCR}": "LAN2",
            "{#IFPHYSADDRESS}": "8:0:27:2b:af:9e"
        }
    ]
}
```

If an entity does not have the specified OID, then the corresponding macro will be omitted for this entity. For example if we have the following data:

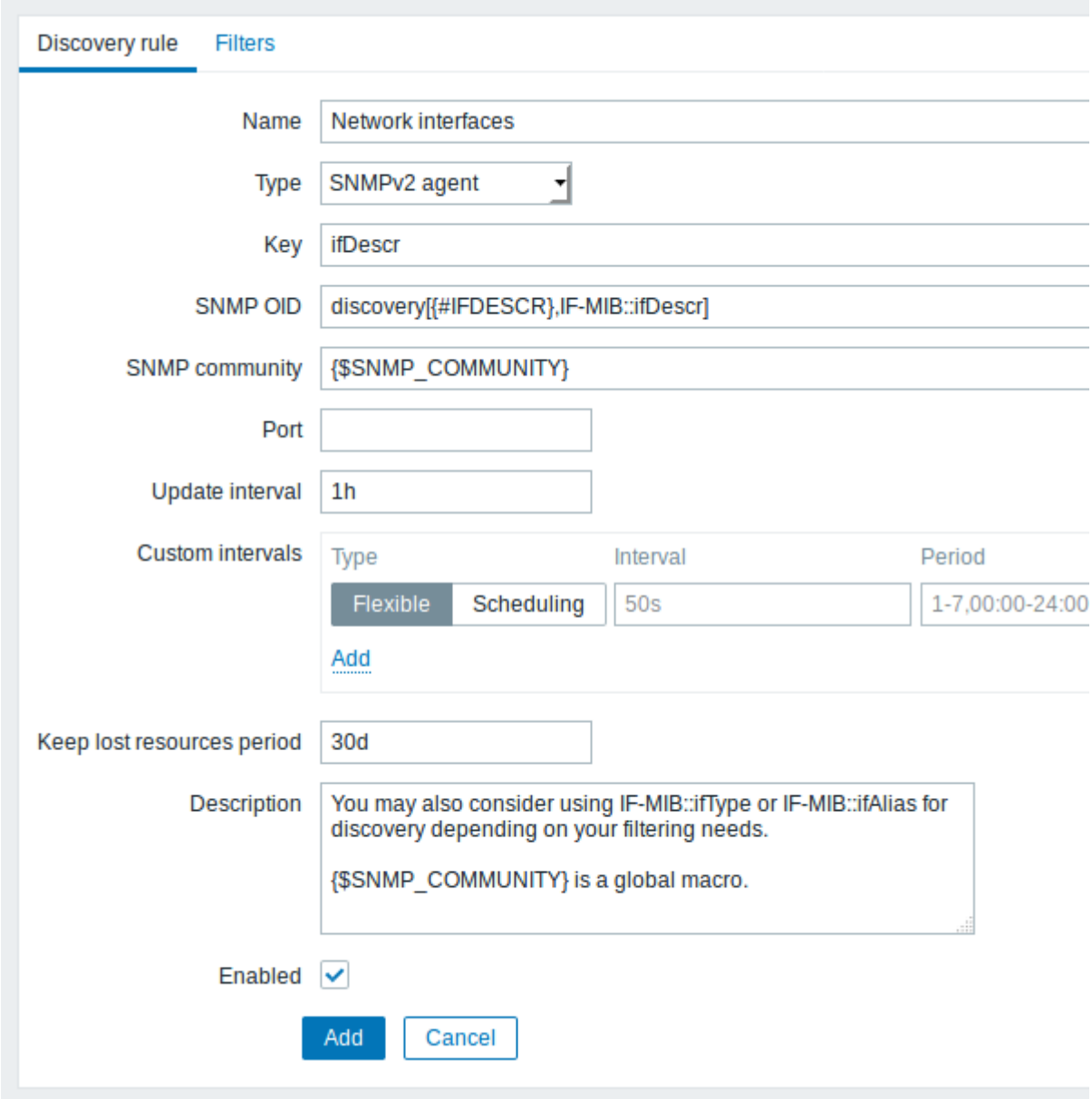
```
ifDescr.1 "Interface #1"
ifDescr.2 "Interface #2"
ifDescr.4 "Interface #4"

ifAlias.1 "eth0"
ifAlias.2 "eth1"
ifAlias.3 "eth2"
ifAlias.5 "eth4"
```

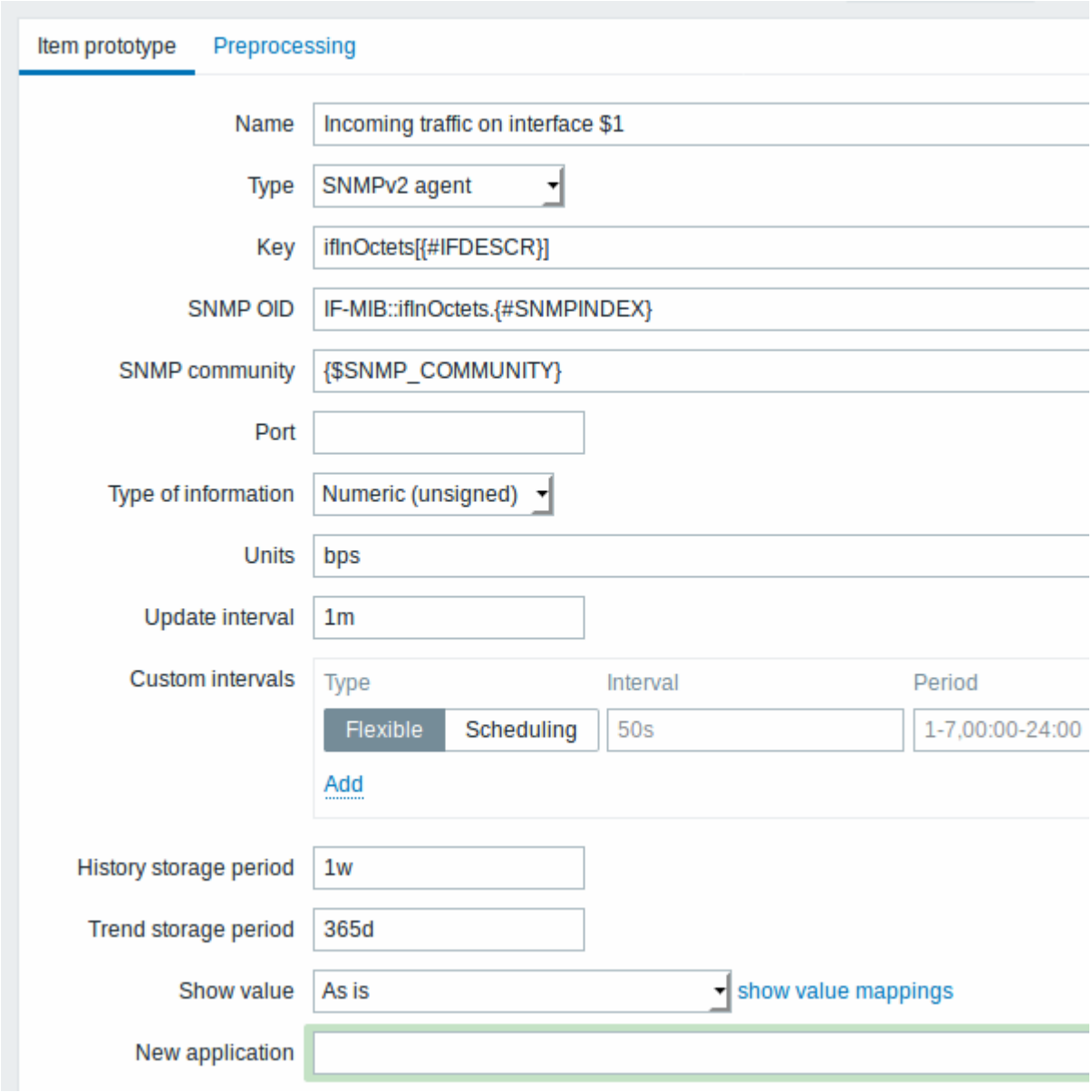
Then in this case SNMP discovery `discovery[{#IFDESCR}, ifDescr, {#IFALIAS}, ifAlias]` will return the following structure:

```
{
    "data": [
        {
            "{#SNMPINDEX}": 1,
            "{#IFDESCR}": "Interface #1",
            "{#IFALIAS}": "eth0"
        },
        {
            "{#SNMPINDEX}": 2,
            "{#IFDESCR}": "Interface #2",
            "{#IFALIAS}": "eth1"
        },
```

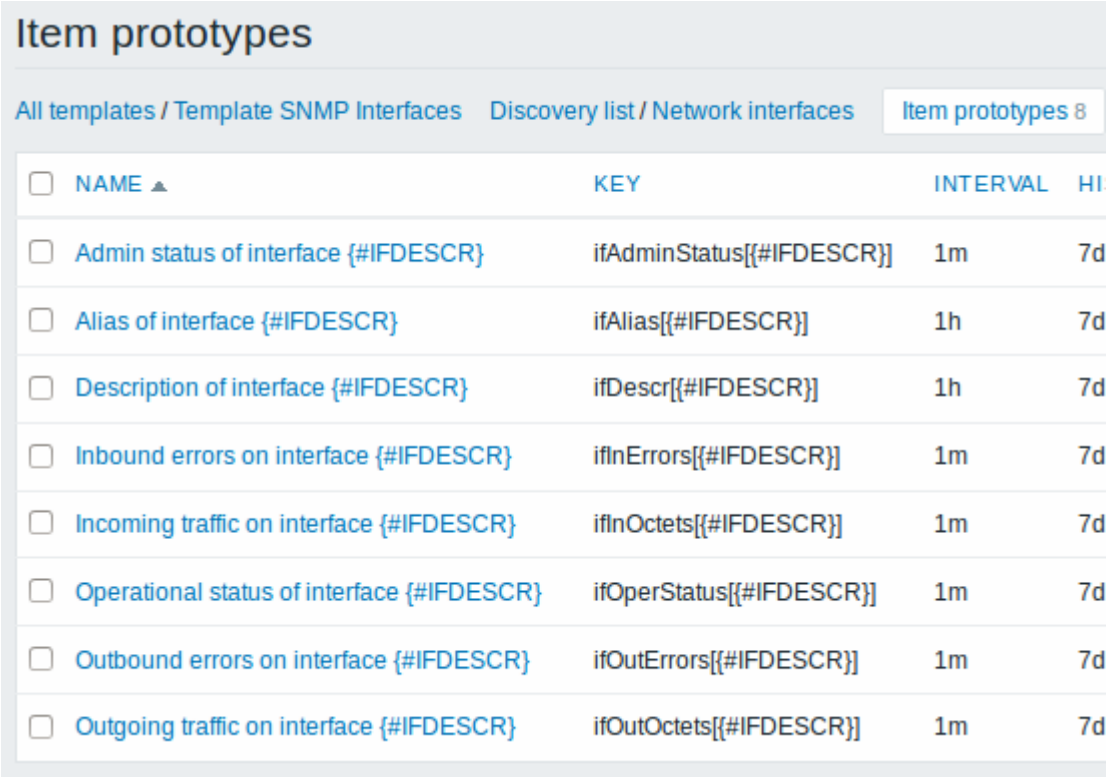
```
        {
            "{#SNMPINDEX}": 3,
            "{#IFALIAS}": "eth2"
        },
        {
            "{#SNMPINDEX}": 4,
            "{#IFDESCR}": "Interface #4"
        },
        {
            "{#SNMPINDEX}": 5,
            "{#IFALIAS}": "eth4"
        }
    ]
}
```



The following screenshot illustrates how we can use these macros in item prototypes:



Again, creating as many item prototypes as needed:

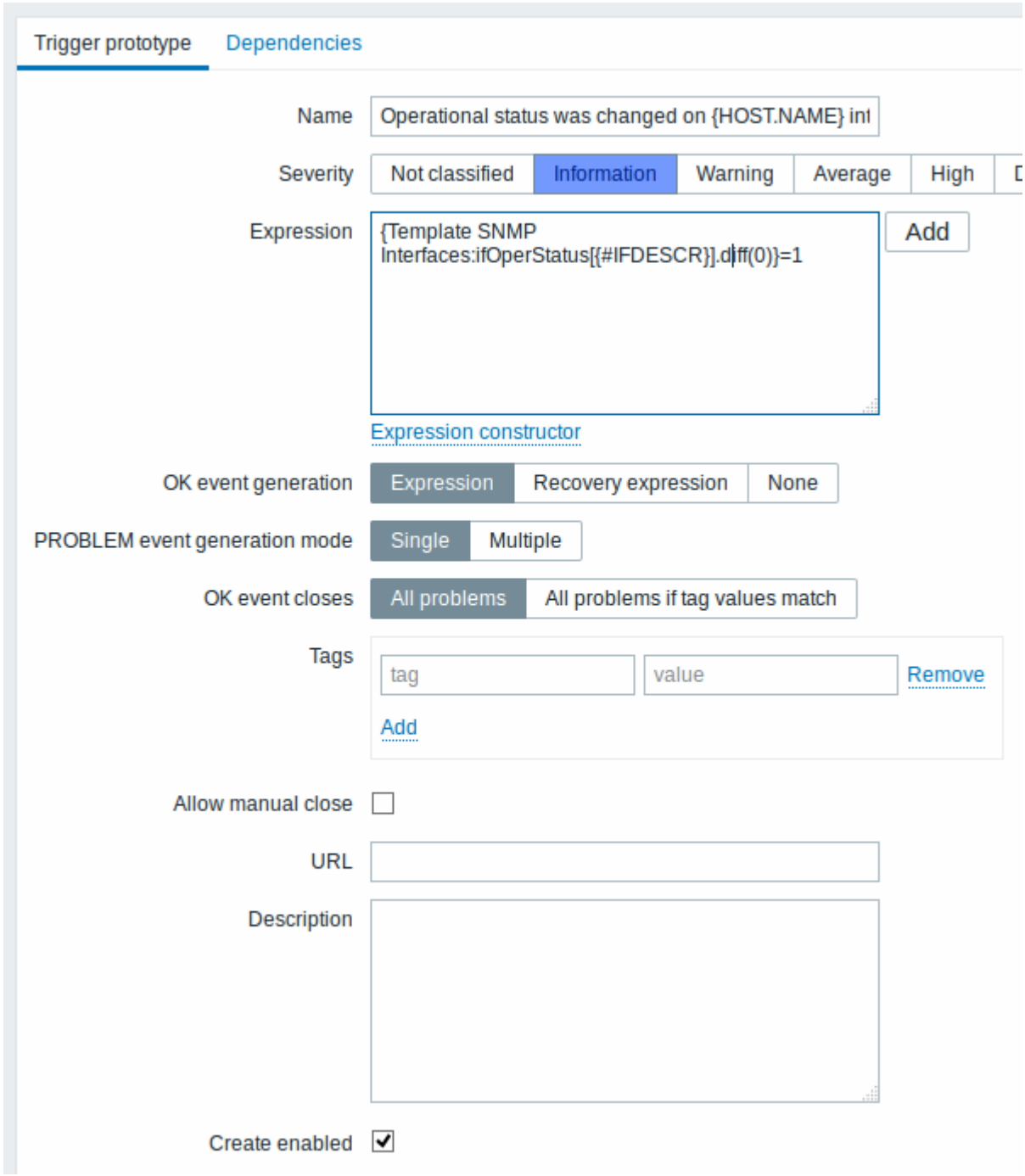


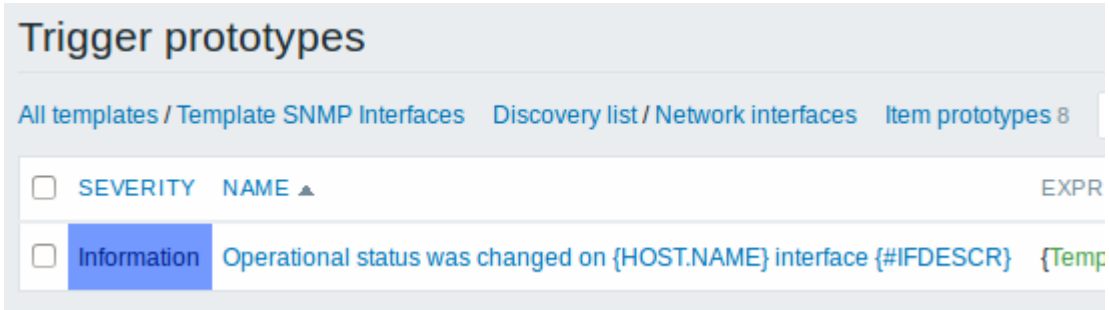
Item prototypes

All templates / Template SNMP Interfaces Discovery list / Network interfaces Item prototypes 8

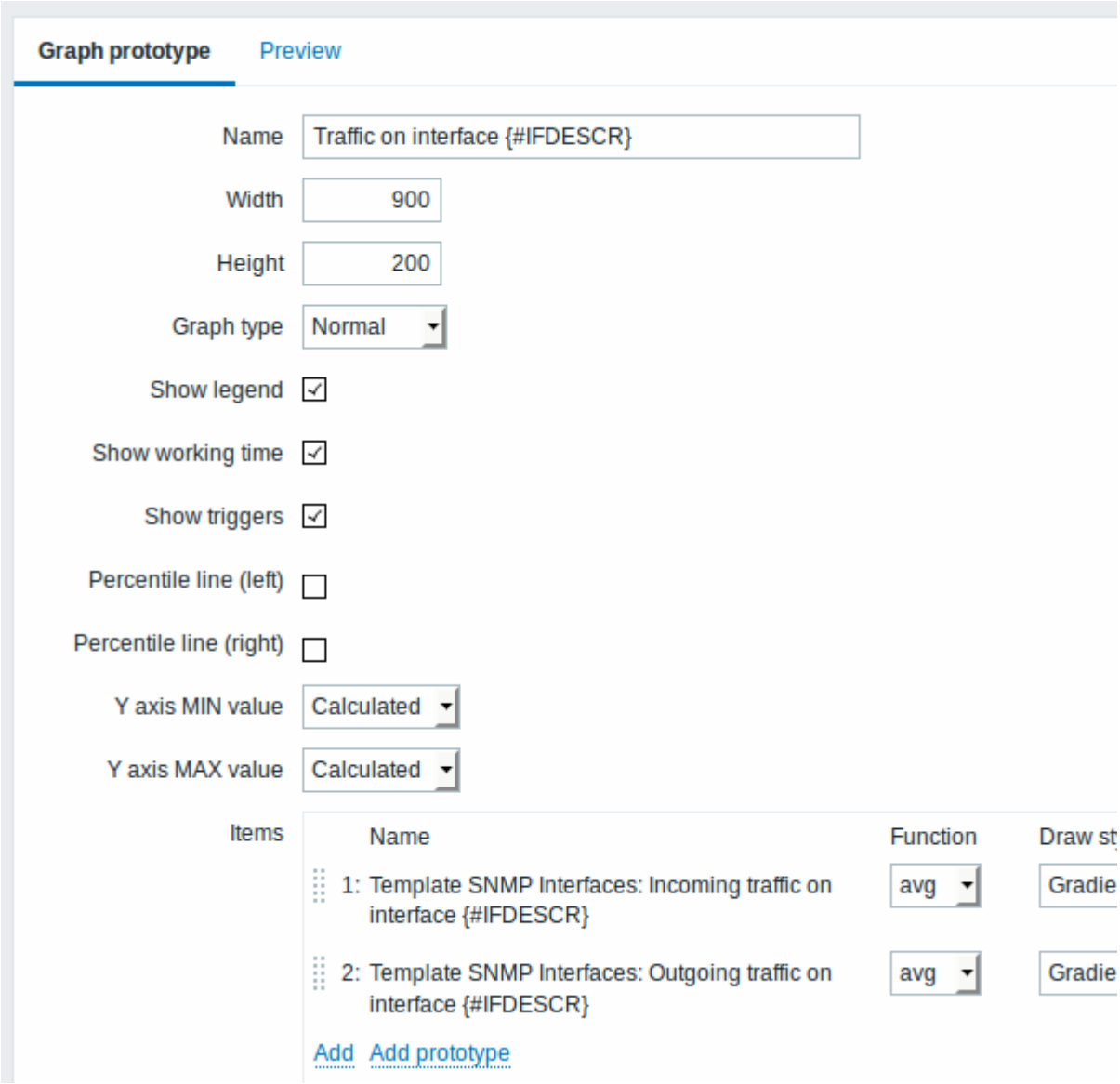
NAME ▲	KEY	INTERVAL	HI
Admin status of interface {#IFDESCR}	ifAdminStatus[{#IFDESCR}]	1m	7d
Alias of interface {#IFDESCR}	ifAlias[{#IFDESCR}]	1h	7d
Description of interface {#IFDESCR}	ifDescr[{#IFDESCR}]	1h	7d
Inbound errors on interface {#IFDESCR}	ifInErrors[{#IFDESCR}]	1m	7d
Incoming traffic on interface {#IFDESCR}	ifInOctets[{#IFDESCR}]	1m	7d
Operational status of interface {#IFDESCR}	ifOperStatus[{#IFDESCR}]	1m	7d
Outbound errors on interface {#IFDESCR}	ifOutErrors[{#IFDESCR}]	1m	7d
Outgoing traffic on interface {#IFDESCR}	ifOutOctets[{#IFDESCR}]	1m	7d

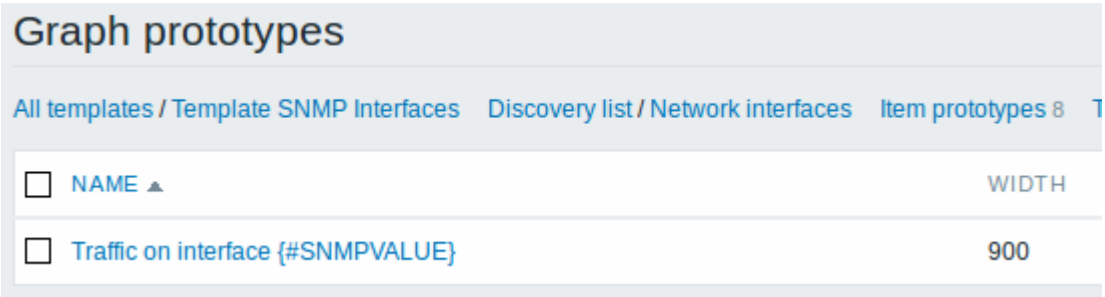
As well as trigger prototypes:



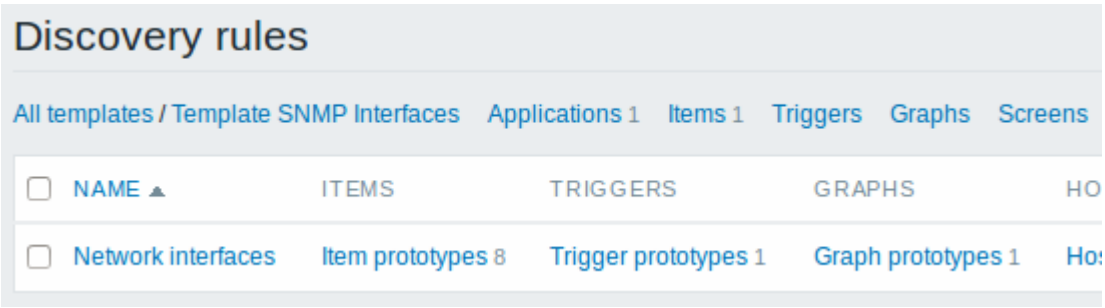


And graph prototypes:



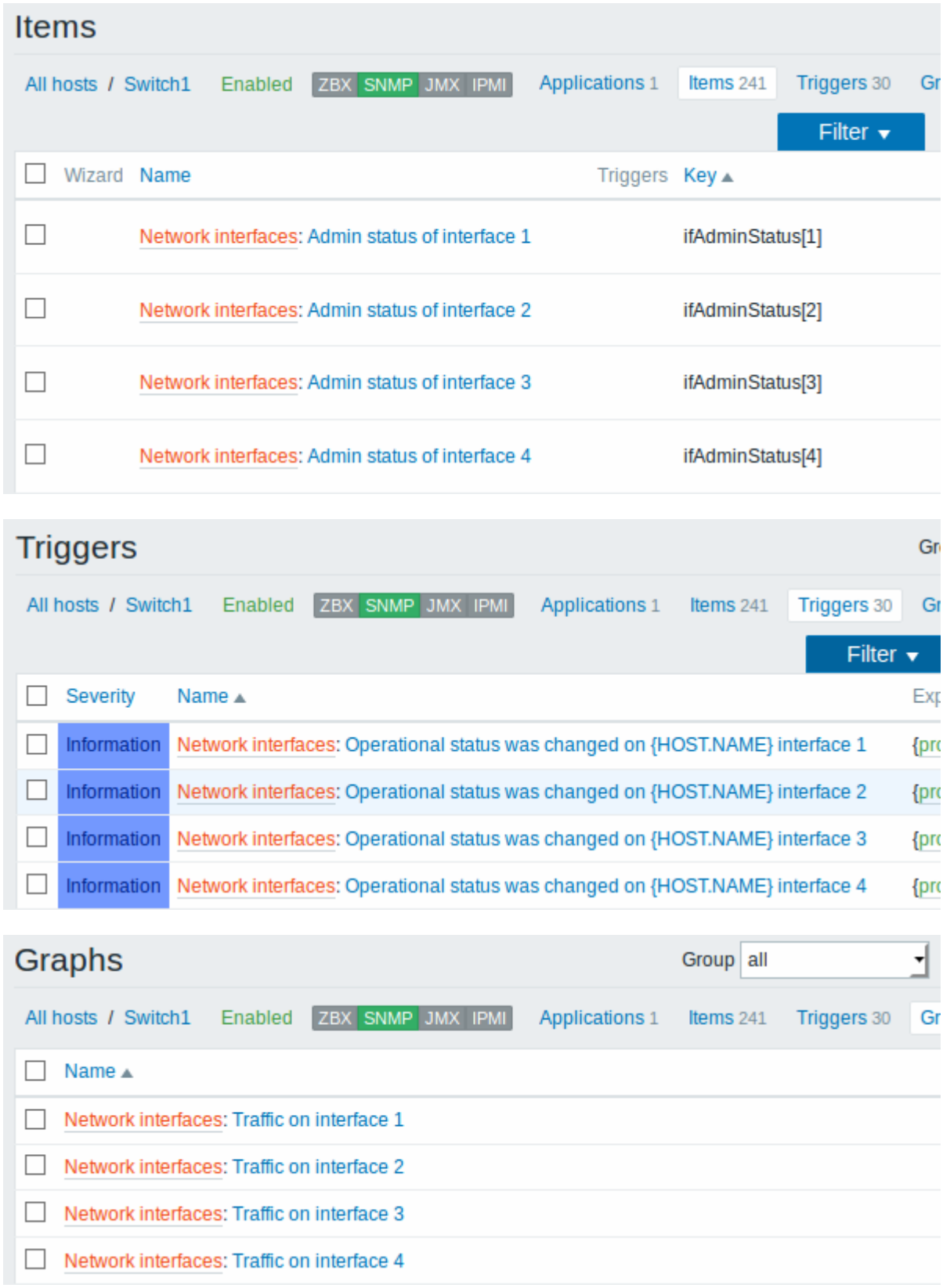


A summary of our discovery rule:



When server runs, it will create real items, triggers and graphs based on the values the SNMP

discovery rule returns. In the host configuration they are prefixed with an orange link to a discovery rule they come from.

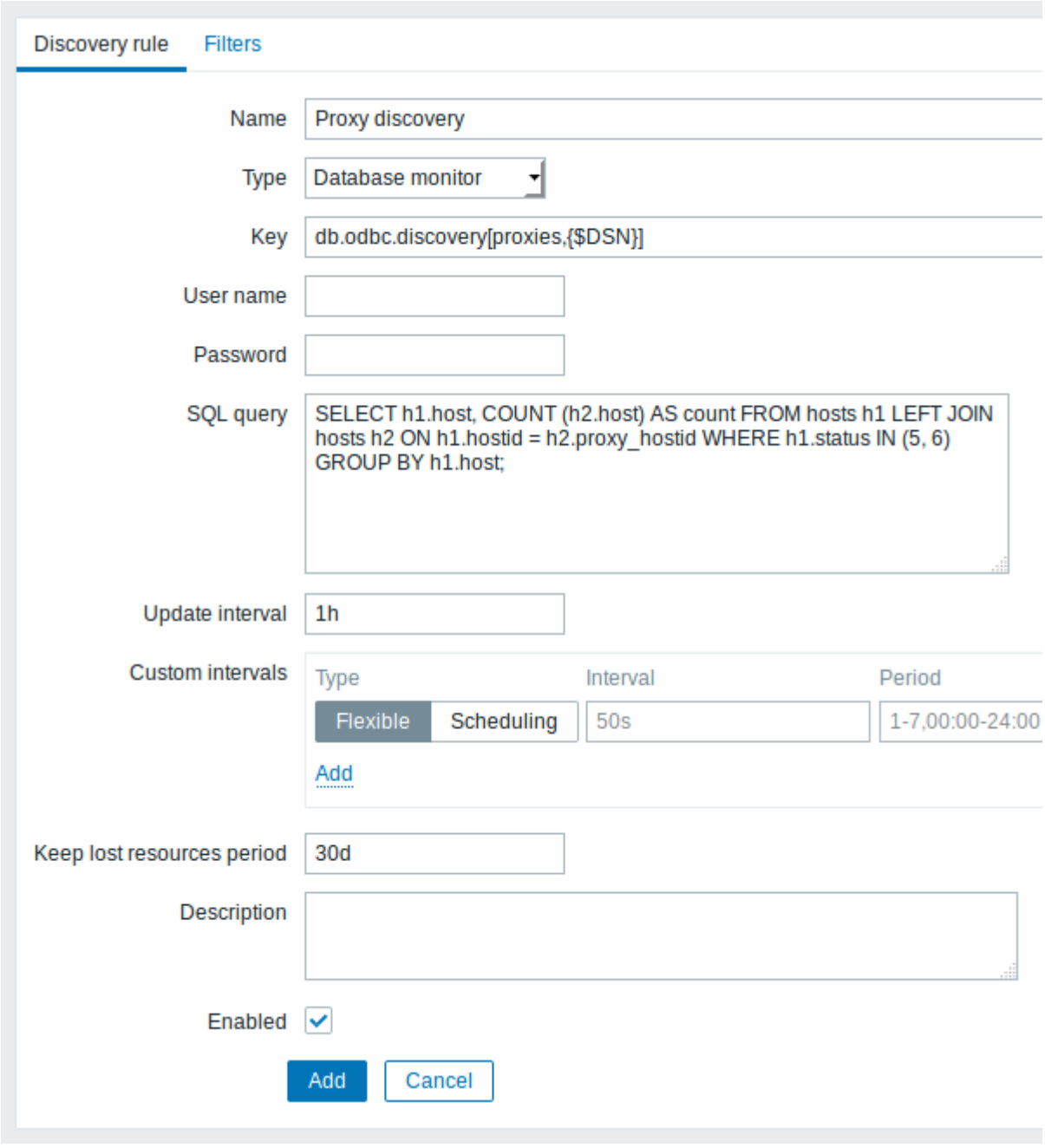


3.5 Discovery using ODBC SQL queries

This type of discovery is done using SQL queries, whose results get automatically transformed into a JSON object suitable for low-level discovery. SQL queries are performed using items of type “Database monitor”. Therefore, most of the instructions on ODBC monitoring page apply in order to get a working “Database monitor” discovery rule, the only difference being that “db.odbc.discovery[<description>,<dsn>]” key should be used instead of “db.odbc.select[<description>,<dsn>]”.

As a practical example to illustrate how the SQL query is transformed into JSON, let us consider low-level discovery of Zabbix proxies by performing an ODBC query on Zabbix database. This is useful for automatic creation of “zabbix[proxy,<name>,lastaccess]” internal items to monitor which proxies are alive.

Let us start with discovery rule configuration:



Here, the following direct query on Zabbix database is used to select all Zabbix proxies, together with the number of hosts they are monitoring. The number of hosts can be used, for instance, to filter out empty proxies:

```
mysql> SELECT h1.host, COUNT(h2.host) AS count FROM hosts h1 LEFT JOIN hosts
h2 ON h1.hostid = h2.proxy_hostid WHERE h1.status IN (5, 6) GROUP BY
h1.host;
+---------+-------+
| host    | count |
+---------+-------+
| Japan 1 |     5 |
| Japan 2 |    12 |
| Latvia  |     3 |
+---------+-------+
3 rows in set (0.01 sec)
```

By the internal workings of “db.odbc.discovery[]” item, the result of this query gets automatically transformed into the following JSON:

```
{
    "data": [
        {
            "{#HOST}": "Japan 1",
            "{#COUNT}": "5"
        },
        {
            "{#HOST}": "Japan 2",
            "{#COUNT}": "12"
        },
        {
            "{#HOST}": "Latvia",
            "{#COUNT}": "3"
        }
    ]
}
```

It can be seen that column names become macro names and selected rows become the values of these macros.

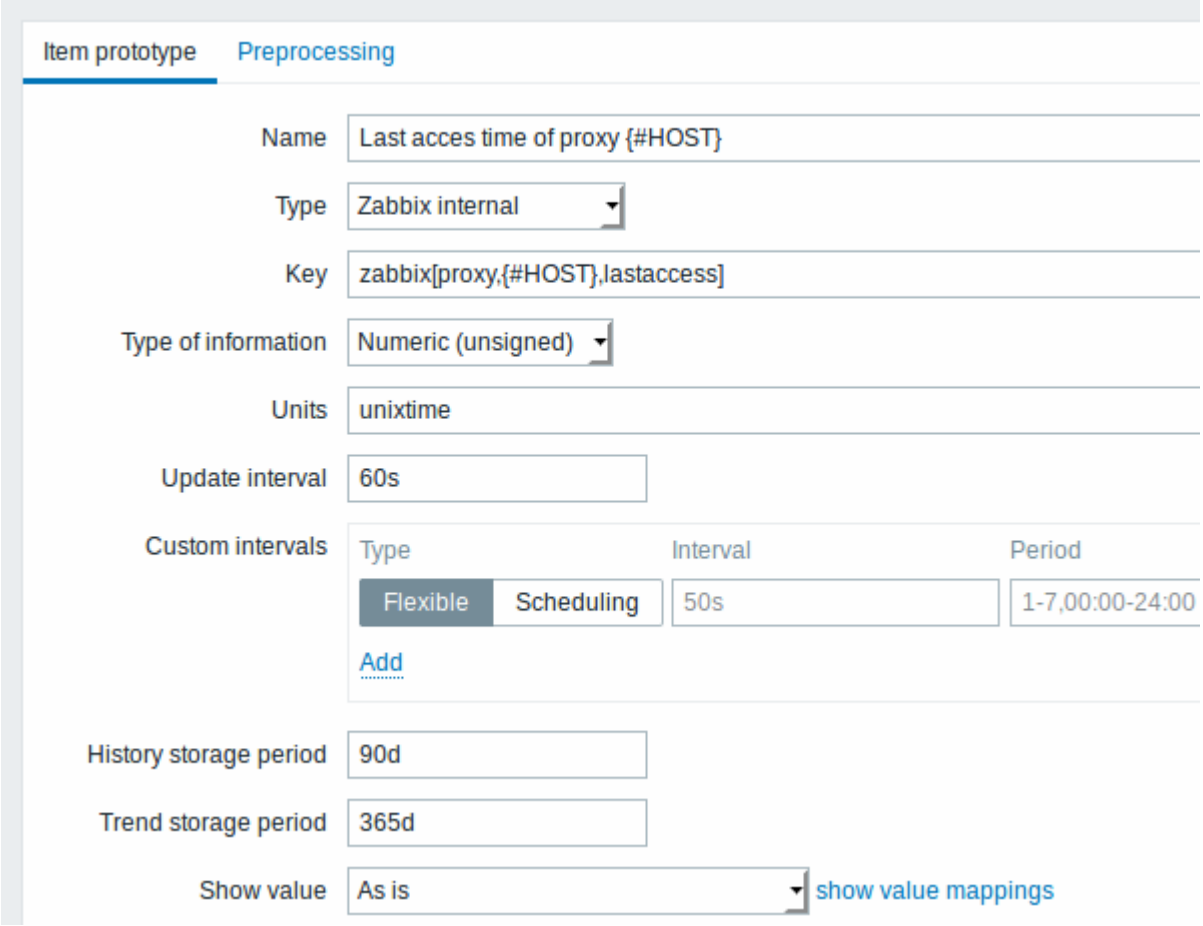
If it is not obvious how a column name would be transformed into a macro name, it is suggested to use column aliases like “COUNT(h2.host) AS count” in the example above.

In case a column name cannot be converted into a valid macro name, the discovery rule becomes not supported, with the error message detailing the offending column number. If additional help is desired, the obtained column names are provided under DebugLevel=4 in Zabbix server log file:

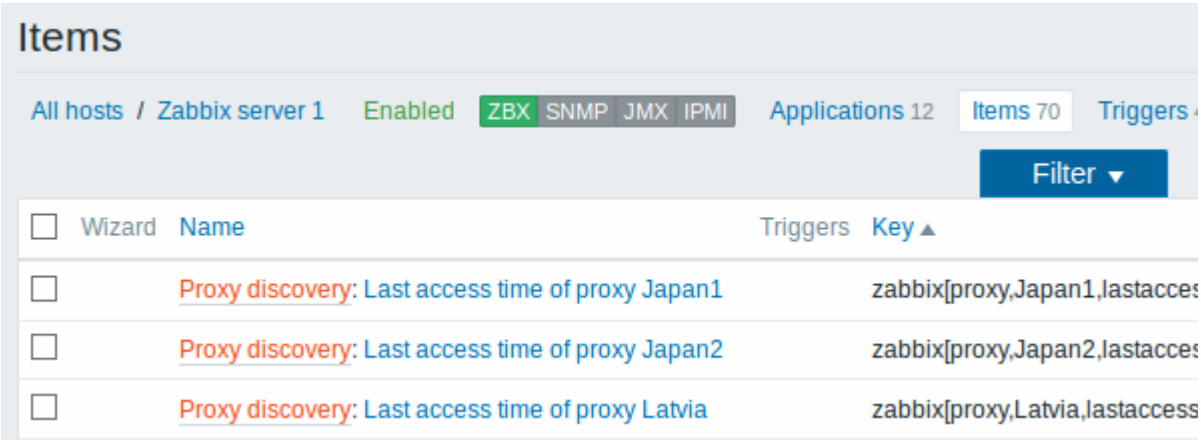
```
$ grep db.odbc.discovery /tmp/zabbix_server.log
 ...
 23876:20150114:153410.856 In db_odbc_discovery() query:'SELECT h1.host,
COUNT(h2.host) FROM hosts h1 LEFT JOIN hosts h2 ON h1.hostid =
```

```
h2.proxy_hostid WHERE h1.status IN (5, 6) GROUP BY h1.host;'
 23876:20150114:153410.860 db_odbc_discovery() column[1]:'host'
 23876:20150114:153410.860 db_odbc_discovery() column[2]:'COUNT(h2.host)'
 23876:20150114:153410.860 End of db_odbc_discovery():NOTSUPPORTED
 23876:20150114:153410.860 Item [Zabbix
server:db.odbc.discovery[proxies,{$DSN}]] error: Cannot convert column #2
name to macro.
```

Now that we understand how a SQL query is transformed into a JSON object, we can use {#HOST} macro in item prototypes:



Once discovery is performed, an item will be created for each proxy:



3.6 Discovery of Windows services

Windows service discovery is done in the same way as discovery of file systems. The key to use in the discovery rule is “service.discovery” and the following macros are supported for use in the filter and item/trigger/graph prototypes:

```
{#SERVICE.NAME}
{#SERVICE.DISPLAYNAME}
{#SERVICE.DESCRIPTION}
{#SERVICE.STATE}
{#SERVICE.STATENAME}
{#SERVICE.PATH}
{#SERVICE.USER}
{#SERVICE.STARTUP}
{#SERVICE.STARTUPNAME}
```

Based on Windows service discovery you may create an item prototype like “service.info[{#SERVICE.NAME},<param>]”, where *param* accepts the following values: *state*, *displayname*, *path*, *user*, *startup* or *description*. For example, to acquire the display name of a service you should use a “service.info[{#SERVICE.NAME},displayname]” item. If *param* value is not specified (“service.info[{#SERVICE.NAME}]”), the default parameter *state* is used.

{#SERVICE.STATE} and {#SERVICE.STATENAME} macros return the same content, however, {#SERVICE.STATE} returns a numerical value (0-7), while {#SERVICE.STATENAME} returns text (*running*, *paused*, *start pending*, *pause pending*, *continue pending*, *stop pending*, *stopped* or *unknown*). The same applies to {#SERVICE.STARTUP} and {#SERVICE.STARTUPNAME}, where one returns a numerical value (0-4) while the other - text (*automatic*, *automatic delayed*, *manual*, *disabled*, *unknown*).

3.7 Setting up multiple LLD rules for the same item

Since Zabbix agent version 3.2 it is possible to alter low-level discovery item keys using “Alias” parameter in zabbix_agentd.conf file to enable configuration of several LLD rules for the same item.

3.8 Creating custom LLD rules

It is also possible to create a completely custom LLD rule, discovering any type of entities - for example, databases on a database server.

To do so, a custom item should be created that returns JSON, specifying found objects and optionally - some properties of them. The amount of macros per entity is not limited - while the built-in discovery rules return either one or two macros (for example, two for filesystem discovery), it is possible to return more.

The required JSON format is best illustrated with an example. Suppose we are running an old Zabbix

1.8 agent (one that does not support “vfs.fs.discovery”), but we still need to discover file systems. Here is a simple Perl script for Linux that discovers mounted file systems and outputs JSON, which includes both file system name and type. One way to use it would be as a UserParameter with key “vfs.fs.discovery_perl”:

```
#!/usr/bin/perl

$first = 1;

print "{\n";
print "\t\"data\":[\n\n";

for (`cat /proc/mounts`)
{
    ($fsname, $fstype) = m/\S+ (\S+) (\S+)/;

    print "\t,\n" if not $first;
    $first = 0;

    print "\t{\n";
    print "\t\t\"{#FSNAME}\":\"$fsname\",\n";
    print "\t\t\"{#FSTYPE}\":\"$fstype\"\n";
    print "\t}\n";
}

print "\n\t]\n";
print "}\n";
```

Allowed symbols for LLD macro names are **0-9** , **A-Z** , **_** , **.**

Lowercase letters are not supported in the names.

An example of its output (reformatted for clarity) is shown below. JSON for custom discovery checks has to follow the same format.

```
{
  "data":[
  { "{#FSNAME}":"/",                          "{#FSTYPE}":"rootfs"   },
  { "{#FSNAME}":"/sys",                       "{#FSTYPE}":"sysfs"    },
  { "{#FSNAME}":"/proc",                      "{#FSTYPE}":"proc"     },
  { "{#FSNAME}":"/dev",                       "{#FSTYPE}":"devtmpfs" },
  { "{#FSNAME}":"/dev/pts",                   "{#FSTYPE}":"devpts"   },
  { "{#FSNAME}":"/lib/init/rw",               "{#FSTYPE}":"tmpfs"    },
  { "{#FSNAME}":"/dev/shm",                   "{#FSTYPE}":"tmpfs"    },
  { "{#FSNAME}":"/home",                      "{#FSTYPE}":"ext3"     },
  { "{#FSNAME}":"/tmp",                       "{#FSTYPE}":"ext3"     },
  { "{#FSNAME}":"/usr",                       "{#FSTYPE}":"ext3"     },
  { "{#FSNAME}":"/var",                       "{#FSTYPE}":"ext3"     },
  { "{#FSNAME}":"/sys/fs/fuse/connections",   "{#FSTYPE}":"fusectl"  }
  ]
```

```
}
```

Then, in the discovery rule's “Filter” field, we could specify “{#FSTYPE}” as a macro and “rootfs|ext3” as a regular expression.

You don't have to use macro names FSNAME/FSTYPE with custom LLD rules, you are free to use whatever names you like.

3.9 Using LLD macros in user macro contexts

User macros with context can be used to accomplish more flexible thresholds in trigger expressions. Different thresholds may be defined on user macro level and then used in trigger constants depending on the discovered context. Discovered context appears when the low-level discovery macros used in the macros are resolved to real values.

To illustrate we can use data from the example above and assume that the following file systems will be discovered: `/`, `/home`, `/tmp`, `/usr`, `/var`.

We may define a free-disk-space trigger prototype for a host, where the threshold is expressed by a user macro with context:

`{host:vfs.fs.size[{#FSNAME},pfree].last()}<`**`{$LOW_SPACE_LIMIT:"{#FSNAME}"}`**

Then add user macros:

- `{$LOW_SPACE_LIMIT}` **10**
- `{$LOW_SPACE_LIMIT:/home}` **20**
- `{$LOW_SPACE_LIMIT:/tmp}` **50**

Now, once the file systems are discovered, events will be generated if `/`, `/usr` and `/var` filesystems have less than **10**% of free disk space, the `/home` filesystem - less than **20**% of free disk space or the `/tmp` filesystem - less than **50**% of free disk space.

LLD macros are not supported inside of user macro contexts in trigger function parameters.