Zabbix template guidelines

Disclaimer

Current document should not be considered as a strict set of rules everybody must follow. Instead, this document only reflects our current approach to template building and any rule or best practice described here may evolve to something else or may be abandoned in the future.

Current document status: draft available for preview and feedback.

Introduction

Zabbix approach to monitoring. Resource vs service

Before we can start talking about what a good template is, let’s briefly discuss monitoring in general. As monitoring is such common and ambiguous term and there is no true RFC definition of what it is. Some may say that monitoring is when you are constantly looking for mentions about your company on social networks, while others may say it is when you collect sensor readings about your plant soil nutrition and moisture. As Zabbix is the universal monitoring solution, we provide a toolset for any kind of monitoring project. But in this document, we will focus on our views on monitoring services provided by modern distributed systems.

Everything can be seen as a service or as a resource.

A service is something that your organization provides to the outside world. Like an online retail store, or public email service. If your store is online and customers can successfully purchase something from it – your service is available. Or maybe service is something that your department provides to the rest of the company. Like computation resources that you provide to other departments on demand. Such internal service can be the dependency for your company online store service. As you can see, service issues clearly affect the real world. Service monitoring availability and performance is what you should always do in the first place. As Google suggests in SRE book, service unavailability should be considered a red-hot situation and the responsible person must be immediately paged.

A resource is a component that helps to provide services. It can be a server, a virtual machine, a container, a database, middleware app, microservice custom app, some hardware controller, network or anything else. You can breakdown resources to even smaller bits, like splitting server into CPU, RAM, IO subsystem or splitting network link to layer 3 and 2 connectivity and physical link present on both sides. The modern distributed system might be a complex set of different resources with dynamic nature where resources are added and removed on demand based on the service load, just like in Kubernetes cluster or AWS. Resources require your monitoring attention too, but differently. Because resource unavailability doesn’t necessarily automatically affect the real world, keep paging people on resource failures at a minimum. Create tickets instead that can be solved during working hours.

Service monitoring is considered a project level monitoring – it is not something you can get out of the box or get a template on share.zabbix.com - it is something you need to create yourself using Zabbix features. That is because all services are different, have different SLOs, have different architecture and so on, so it’s hard to prepare a common blueprint for service monitoring. But consider the
following approaches:

- Try synthetic monitoring: emulate user (person or another application that interact with your service) activity on a regular basis: check for symptoms with the black-box approach:
  - Use Web monitoring in Zabbix and go through common user scenarios, for example, try to log in and purchase test item from the store.
  - Simpler HTTP check will also work – just make sure your website URL returns HTTP 200 OK
  - If your service provides a REST API, write a script that will emulate some common service activity.
- Try real user monitoring approach: gather and read transactions from logs, network, or database about your real users:
  - Count the ratio of success/failure requests
  - Collect request rate per second/per minute to your service
  - Calculate min/max/average response times to your service or create a histogram.

And most likely situations such as zero rate (or sudden drop of rate), high errors ratio (HTTP 500 everywhere) are the situations that indicate serious service problems.

But why do you need to monitor resources if service monitoring is set up? There are multiple reasons but the most important one is this: once you know your service is down (symptom) you need to isolate the root cause of the problem.

Resources is something that is common and generic in many projects, different architectures. That’s where templates can thrive. Seriously, do you really need to waste time to create your own monitoring solution to control OS Linux? Or for MySQL database, Cisco router, or for docker host? Maybe you can spend your time more efficiently by preparing service level monitoring instead.

**What does it take to become a good resource template?**

Let’s try to define some properties of what a good resource template is, some key principles we follow in Zabbix when building templates:

1. **Flexible and reusable**

In Zabbix, the template is equal to the monitoring solution for some specific object. It’s a sort of container that should be used to transfer configuration, monitoring solution between Zabbix server instances. A good quality template is something that Zabbix users create, use for their own good, and then share it with the Zabbix community, so the next person can download this template and reuse it, update it with newer ideas and approaches, contributing to the common cause. So, the first thing that comes in mind when you try to answer a question what a good template in Zabbix is how flexible and reusable it is. If other Zabbix users can download it and use it without changing half of it – that’s really a good sign.

Here are a few rules of the thumb on how to achieve it:

- **Use low-level discovery as much as possible** to avoid unsupported items or triggers. If you have some metric in your situation, that doesn’t mean it will be available in someone else case – it could be different hardware, software version or configuration.
• **Use user macros** in triggers, items. For example, use \{NGINX.URL\} for nginx stub status URL. Or use \{TEMP.MAX.CRIT\} in temperature controlling trigger. This will allow users to configure and fine-tune templates and linked hosts, instead of changing it and breaking compatibility with future versions.

• **Avoid adding rare metrics/triggers required for your project/service** into the resource template. For project/service level metrics that you feel are very specific just move it to another template and link it to the generic template.

• **Avoid external dependencies** where possible. Use internal Zabbix data collection and processing possibilities to collect data first. Use HTTP agent, powerful preprocessing steps such as Javascript, JSONPath, JMX and so on. This would ensure that such template is easy to install, and all its configuration and processing are defined within the template. Resort to external scripts only if there is no alternative available.

2. **Knowledge and expertise**

We also think that a good template is not just a set of metrics (items), thresholds (triggers) and dashboards bundled together. The most important ingredient to a good template is how much expertise and knowledge about a monitored object is contained within. And by expertise and knowledge we mean, not the number of metrics someone knows how to collect – but knowing what metrics are useful and important, and which are just useless, or what thresholds should be used to be notified only about problems that matters without too much noise.

While very minimalistic template may not provide all the information you need, on the other hand, bloated, oversized templates are bad as well, as users lose focus on the most important metrics, as well as they get overwhelmed with problems noise. So:

• Avoid adding **too many metrics**. Keep it simple. Don’t try to do benchmarking, profiling, collect deep debugging level metrics. This will create unnecessary load on Zabbix, and object monitored. Let Zabbix isolate the problem – then do profiling and debugging using specialized tools on the hosts that really require it.

• Avoid creating **too much problem noise** with triggers in the template. Make sure that the problems created from the trigger require immediate (page) or postponed (ticket) action. Avoid ‘for your information’ and ‘this looks weird’ level triggers.

3. **Modularity and scope**

The last thing that is very important is the template scope. We already talked about services and resources, so, generally, the good template is the one that has a scope of the single resource:

• Vendor specific hardware server
• A temperature sensor
• Operation system templates like Linux OS or Windows OS
• Applications like Nginx, Apache, Tomcat, RabbitMQ
• DBs like MySQL, PostgreSQL, Oracle, DB2, Redis, Mongo and so on
• Cloud providers such as AWS, Azure or others
• Virtualization providers such as VMware clusters, Hyper-V
• Container orchestration systems such as Kubernetes
• A network device or network controller

Zabbix Documentation 4.4 - https://www.zabbix.com/documentation/4.4/
• Some custom applications

If you keep a template scope within a single resource, it will be much easier to share such templates and they will be useful to people who have the same building blocks in their architecture. Also, avoid merging resources of different layers – do not add metrics for Linux OS and PostgreSQL into the single template.

But what about ‘inner’, metrics scope? What metric types, classes should you be collecting? Surely you can do monitoring for various reasons, including collecting business indicators or looking for security breaches, but when creating a generic resource Zabbix template, try to adopt the following approach:

3.1 Always start with fault monitoring or availability monitoring. The most popular and very important answer people want to get from monitoring – is my system up and running? So, try to address that in your template first.

That is, prefer black-box monitoring approach here, simple or not so simple health checks are essential and the first thing you or any user want to know about. Add items and triggers to your template that can help you to be sure – the thing you are monitoring is accessible and is up and running. Use ICMP ping, check that TCP port is open, check that API returns HTTP 200 OK and so on.

The second problem that is addressed by fault monitoring is an imminent failure. For example,

• the hardware is overheated and is about to shutdown
• you are running very low on disk space and very soon your DB will refuse to write new data to the DB as there is no space.

Add items and triggers that will help you to intervene and prevent such a drastic outcome.

Also, if your monitoring object can detect faults on its own - use it! Many systems can report faults directly using logs or sending SNMP traps and so on. And that's the kind of expertise we talked earlier provided to you from the developers, vendors, authors of the system you want to control. And nobody knows it better than them. So just make sure you can retranslate faults detected by the system itself in your Zabbix template.

3.2 Once your template can check the health of your system - proceed with performance monitoring. This is where you will need to open the box wide open (white-boxing). There are really nice methods out there to help you choose what metrics to collect first: USE, Four golden signals from Google or RED for request driven services. Just make sure you extend the template with items and triggers to help solve the following use cases:

• My system is slow. It is up but response time is unsatisfactory. Performance has degraded.
• We just had a big outage. We need to investigate and do retrospective analysis to find out what happened to make sure it never happens again. To do such analysis we need helpful metrics collected beforehand.

3.3 Inventory and state control

While Zabbix is not the inventory system, it still can collect lots of information about the resource and most importantly, detect changes, such as system being restarted outside of maintenance window, the version was updated, or it is outdated and so on. So, make it part of your template checklist.
3.4 Security

If you know how to properly detect security issues with the resource, i.e.:

- Resource version used is the subject to CVE. Consider updating
- Misconfiguration causes the system to be publicly available without proper authentication when it should be.

Then consider adding such items and triggers to your template as well.

4. Follow guidelines

Finally, is the style of the template. How to name your items? Templates? Triggers? If we would all follow the same style when creating Zabbix templates - then it wouldn’t really matter who made this template - you, Zabbix or another community member from the other side of the globe - as template contents and layout will be very predictable and expected.

Conclusion

Following style guidelines and template core principles mean that we can reuse each other templates as building bricks for our monitoring projects, saving time and adding someone else knowledge on the monitored object.

That concludes the introduction to Zabbix template guidelines, a comprehensive set of rules how we build templates in Zabbix.

I recommend reading the guide if:

- You want to share your template with the rest of the world
- You want to avoid common mistakes when creating a template
- As a hardware or software vendor, you want to provide Zabbix template for your solution

Style guide

1.1 General

1.1.1 Avoid extensive template tuning

Try to keep everything default and simple in the template as long as possible. For example, item attributes as update interval, history, trends. Change them only if there is a good reason for this. Don’t waste time deciding whether you should make update interval 1 minute or 2 minutes, or maybe 2.5 minutes? Use the Pareto principle to get 80% template efficiency with 20% of your time effort. Don’t over-engineer it, unless there is a reason for it.
1.1.2 Template language

All template descriptions, names and so on, must be created in the English language first. If you need a template in another language – consider maintaining two copies – English and localized version.

1.1.3 Everything enabled

All items, triggers, LLD rules and other configuration entities should be enabled by default to make the template useful out of the box.

1.1.4 Avoid global macros

If user macros are used, define them in the template itself instead of using global macros - that way users get either the default values or an example of what the macro names are. If global macros are used, they are not exported along with the template.

1.1.5 Avoid global regexes

Avoid using global regexes in templates if possible, as they are not exported with the template. If global regex is used, document in the README what global regex with what values must be used with the template. (Note that since 4.0 you can use NOT to filter out negative results in LLD filters, see ZBXNEXT-2788)

1.1.6 Avoid trigger dependencies for triggers from different templates

Avoid trigger dependencies for triggers from different templates. Use global correlation and event tags instead.

1.1.7 Keep templates modular. Profile templates

Generally, to keep template reusable and modular, the single template should be capable to monitor single resource or inseparable set of resources only.

If you need to monitor multiple resources on the host (and you probably do) – consider creating so-called ‘Profile’ or ‘Meta’ empty template and then link multiple resource templates to it.
Good
“Template App Apache”,
“Template DB MySQL”,
“Template App PHP”,
“Template OS Linux” all linked to profile template named “Template App LAMP”.
Then, “Template App LAMP” is linked to hosts “lamp1” and “lamp2”.

Bad
“Template App Apache”,
“Template DB MySQL”,
“Template App PHP”,
“Template OS Linux” all linked directly to hosts “lamp1” and “lamp2”.

It is also a good place to redefine user macros on the profile template level if needed.

### 1.2 Templates

#### 1.2.1 Template name

Template name starts with “Template”, then comes the <Category short name>, then <Template name> itself (the specific part).

All parts are separated by spaces, but underscores can also be used.

All names (group, template, item, trigger, graph, application, dashboard, discovery) use normal case inside the specific part – for example, “Template App Zabbix server”.

To distinguish templates, most popular data collection method can be stated as an extra suffix at the end of the name, for example: “by SNMPv1”, “by SNMPv2”, “by SNMPv3”, “by Zabbix agent”, “by Zabbix agent active”, “by IPMI”, “by JMX”, “by ODBC” and so on.

<table>
<thead>
<tr>
<th>Good</th>
<th>Bad</th>
</tr>
</thead>
<tbody>
<tr>
<td>Template App Nginx by HTTP</td>
<td>Template NGINX</td>
</tr>
<tr>
<td>Template DB MySQL</td>
<td>Template MySQL</td>
</tr>
<tr>
<td>Template Net Brocade switch by SNMPv2</td>
<td>SNMP Brocade switch</td>
</tr>
<tr>
<td>Template Net Brocade switch SNMPv2</td>
<td>Brocade switch Template</td>
</tr>
<tr>
<td>Template_Net_Brocade_switch_by SNMPv2</td>
<td>Template Brocade Switch</td>
</tr>
</tbody>
</table>

#### 1.2.2 Template visible name

Currently, we suggest leaving the template visible name empty.

#### 1.2.3 Template description

Use this field to provide a short overview of the template, including:

- Short description
- Template homepage URL (at share.zabbix.com or github.com or else)
- Template author
- If documentation is quite short – documentation can be provided inline
- Current template version
- The simple changelog can be provided as well
1.2.4 Choosing a template group

All templates must be added into a template subgroup called **Templates/</Category Full Name>**.

<table>
<thead>
<tr>
<th>Good</th>
<th>Bad</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Template Net Cisco by SNMP&quot; added into &quot;Templates/Network Devices&quot;</td>
<td>&quot;Template Net Cisco by SNMP&quot; added into &quot;Datacenter/Network&quot; host group</td>
</tr>
</tbody>
</table>

1.2.5 Pick a template category

You can create your own template categories. But first, consider using one of the recommended categories:

<table>
<thead>
<tr>
<th>Category full name</th>
<th>Category short name</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modules</td>
<td>Module</td>
<td>For all templates not intended for direct host linkage but often used as a dependency for other templates</td>
<td>Template Module Generic SNMPv2&lt;br&gt;Template Module HOST-RESOURCES-MIB SNMPv2&lt;br&gt;Template Module Interfaces SNMPv2&lt;br&gt;Template Module Interfaces simple SNMPv2&lt;br&gt;Template Module ICMP ping</td>
</tr>
<tr>
<td>Network devices</td>
<td>Net</td>
<td>For all network devices (or software) which main role is networking including switches, routers, wireless, firewalls, etc</td>
<td>Template Net Generic device SNMPv2&lt;br&gt;Template Net Juniper SNMPv2&lt;br&gt;Template Net Mikrotik SNMPv2&lt;br&gt;Template Net Dell Force S-Series SNMPv1&lt;br&gt;Template Net Brocade FC SNMPv1</td>
</tr>
<tr>
<td>Storage devices</td>
<td>Storage</td>
<td>For FC and other storage devices</td>
<td>Template Storage IBM Storwize by SNMPv1&lt;br&gt;Template Storage EMC VNX</td>
</tr>
<tr>
<td>Server hardware</td>
<td>Server</td>
<td>For server hardware (iLO, IMM, blades and so on)</td>
<td>Template Server IBM IMM2 by SNMPv2&lt;br&gt;Template Server IBM IMM2 by IPMI&lt;br&gt;Template Server HP iLO by SNMPv2</td>
</tr>
<tr>
<td>Operating systems</td>
<td>OS</td>
<td>For server operating systems (Windows, Linux, OSX, ESXi by SNMP, Solaris and so on)</td>
<td>Template OS Linux&lt;br&gt;Template OS Linux by Zabbix agent active&lt;br&gt;Template OS Linux by SNMPv2&lt;br&gt;Template OS Linux VMware&lt;br&gt;Template OS ESXi SNMPv2&lt;br&gt;Template OS Solaris&lt;br&gt;Template OS Windows&lt;br&gt;Template OS Windows XP by SNMPv2</td>
</tr>
<tr>
<td>Databases</td>
<td>DB</td>
<td>For all SQL, NoSQL and key-value storages</td>
<td>Template DB MySQL&lt;br&gt;Template DB Redis&lt;br&gt;Template DB Oracle by ODBC</td>
</tr>
<tr>
<td>Category full name</td>
<td>Category short name</td>
<td>Description</td>
<td>Example</td>
</tr>
<tr>
<td>--------------------</td>
<td>---------------------</td>
<td>-------------</td>
<td>---------</td>
</tr>
<tr>
<td>Power</td>
<td>Power</td>
<td>For UPSes and other power category devices</td>
<td>Template Power Generic UPS by SNMPv2&lt;br&gt;Template Power APC by SNMPv2&lt;br&gt;Template Power Eaton SNMPv2</td>
</tr>
<tr>
<td>Telephony</td>
<td>Tel</td>
<td>For hardware and software telephony systems (Asterisk, Panasonic, Avaya, etc) including IP phones</td>
<td>Template Tel Asterisk by SNMPv3&lt;br&gt;Template Tel Avaya</td>
</tr>
<tr>
<td>Virtualization</td>
<td>VM</td>
<td>For VMs, Hyper-V, VMware, Xen, KVM...</td>
<td>Template VM VMWare&lt;br&gt;Template VM Hyper-V&lt;br&gt;Template VM Xen</td>
</tr>
<tr>
<td>Printers</td>
<td>Printer</td>
<td>For printers and MFPs</td>
<td>Template Printer Generic by SNMPv2&lt;br&gt;Template Printer HP Laserjet</td>
</tr>
<tr>
<td>Applications</td>
<td>App</td>
<td>For software that doesn't fit in any category above</td>
<td>Template App Generic Java JMX&lt;br&gt;Template App RabbitMQ&lt;br&gt;Template App Apache Tomcat JMX&lt;br&gt;Template App Apache ActiveMQ&lt;br&gt;Template App Docker&lt;br&gt;Template App Apache2&lt;br&gt;Template App Nginx by HTTP</td>
</tr>
<tr>
<td>Hardware</td>
<td>HW</td>
<td>For other hardware that doesn't fit in any category above</td>
<td>Template HW Netbotz by SNMPv2&lt;br&gt;Template HW Siemens PLC by Modbus&lt;br&gt;Template HW Skycontrol by SNMPv2&lt;br&gt;Template HW Skycontrol SNMPv1&lt;br&gt;Template HW Netping</td>
</tr>
</tbody>
</table>

### 1.3 Items

#### 1.3.1 Naming an item

Choose a simple, descriptive name for each item.

Prefix item names (metric) with object name (metric location):

*<metric location>*: <metric name>, for example:

- Interface eth0: Bits in
- Interface eth0: Bits out

You may use “#” if the metric location is just a number or index:

- #0: CPU utilization
- #1: CPU utilization

Consider adding suffixes like “per second”, “per hour” etc to describe the metric better.
No user macros or $1 macros must be used in item names, they are deprecated and will be removed in Zabbix 5.0.

Consider prefixing your item with “Get” if this a master item to highlight this item is the collector item, not the final metric.

### 1.3.2 Keys

Keys should use hierarchical dotted format.

```plaintext
namespace.component.metric_name[params]
```

**namespace:**

Required to split metrics of one template from another. In the simplest case, this may be a short product name.

e.g: nginx, pgsql, pgbouncer, docker

**component:**

Component or sub-resource of the monitored object. It could be hierarchical as well.

e.g: upstream, pool, db, db.table, db.client

**metric_name:**

For example: max_reached.

If possible, prefer to name metrics just as they are named in the monitored object itself with an exception if metric format there is completely different or metric name there is totally confusing and not human-friendly.

Every key must start with a letter and must use only Latin letters in lower case in the base part.

If you need a space, you could simply replace it with underscore “_”, e.g: response_time.

Remember that max key length is 255 chars (including users params).

```plaintext
pgsql.db.connection_time[#{PG_DB},${PG_CONN}]
```

e.g: request_time, request_count

Consider to append `.get` for collectors, items that are responsible for retrieving data to be used in
dependent items. (master items)

e.g: pgsql.db.get_connection […], nginx.get_stub, nginx.get_logs

Consider using `.rate` for per second metrics.

e.g: nginx.connections.accepted.rate

Consider using `.total` for accumulators.

**params:**

In params, first comes mandatory params the optional should follow.

Always **quote** all params that contain user or low-level discovery macros.

<table>
<thead>
<tr>
<th>Good</th>
<th>Bad</th>
</tr>
</thead>
<tbody>
<tr>
<td>abc[&quot;${MACRO}&quot;]</td>
<td>abc{${MACRO}}</td>
</tr>
<tr>
<td>abc[#{MACRO}&quot;]</td>
<td>abc{#{MACRO}}</td>
</tr>
</tbody>
</table>

**1.3.3 Item description field**

Use this field to describe:

- What metric is being collected
- Why it is important
- Provide a reference to the documentation if possible

**1.3.4 Units**

Don't forget to provide units wherever possible.

Add your units to blacklist to stop automatic conversion where conversion is silly.

For example:

Use “!requests/s” to prevent “Krequests/s” to appear.

Use preprocessing to transform GB, MB, KB to B (Bytes).

Use preprocessing to transform ms, minutes, hours to seconds.

**1.3.5 Value mapping**

Always use value mappings where applicable, for example, when collecting discrete states.

**1.3.6 Type of information**
Take type restrictions when choosing which one to use:

Type of data as stored in the database after performing conversions, if any Numeric (unsigned) - 64bit unsigned integer Numeric (float) - floating-point number Negative values can be stored. Allowed range: -999999999999.9999 to 999999999999.9999. Starting with Zabbix 2.2, receiving values in scientific notation is also supported. E.g. 1e+7, 1e-4. Character – short text data Log – long text data with optional log related properties (timestamp, source, severity, logeventid) Text – long text data Limits of text data are described in the table below. Read more here
https://www.zabbix.com/documentation/current/manual/config/items/item

If your item is rate (i.e., “Change per second” preprocessing is applied) – use Numeric(float).

Additionally, don’t forget to use Numeric(float) if you need to store negative integers.

**1.3.7 Using time suffixes in update intervals, calculated item formulas**

Always use time (1m, 5m, 1d...) suffixes in update intervals, history storage period, trends storage period, calculated item formulas to improve readability. Remember, that you can use them in user macros too.

**1.3.8 Choosing an update interval, history and trends storage period**

By default, use:

**Update interval: 1m History: 7d Trends: 365d**

Also, consider using preprocessing steps 'Discard unchanged (with heartbeat)' when collecting items that change rarely like statuses or configuration data (e.g. serial numbers or hostname):

If the item is a **health check**:

1m with the heartbeat of 1h

If the item is an **inventory item**:

15m with the heartbeat of 24h

If the item is **“Zabbix raw item”** (master items or items only needed for other calculated items, see below) - set history to 1h and trends to 0, as you don't need to keep such intermediate values.

Please also note: Never set update interval more than 1d, as you will not see such data in the ‘latest data” since Zabbix frontend considers values received more than 24h ago as not latest.

**1.3.9 Applications**

Use applications to logically group items together. Consider to group metrics by metric location (resource, subresource) first.
Do not split items into tiny, micro-groups with applications. Split applications into subgroups only if necessary, for template readability.

Consider using application prototypes where necessary. Especially, where a large number of discovered objects is expected. Again, think about template readability in Latest data and item filtering by Application.

### Single item - single application

Add item only to a single application.

While technically possible, do not add item into more than one application or application prototype to avoid duplicates in “Latest data” and confusing users.

### Zabbix raw items

Consider using an application called “Zabbix raw items” for all items which sole intention is to collect, or buffer values to then pass it further for dependent items or calculated items.

### 1.3.10 Calculated items

Use newlines and spaces to make long formulas human readable.

### 1.3.11 SNMP

SNMP OID field should not use any MIB objects, so templates would be working without MIBs imported. At the same time, provide metric name from MIB as an item key parameter and in item description.

Leading '.' in OID should not be used.
Leave item field ‘Port’ empty for SNMP items. If left empty, then port will be used from SNMP host interface.

1.4 Discovery rules (LLD)

1.4.1 Naming

Choose a simple, descriptive name for each discovery rule. Make sure it always ends with “discovery” word.

<table>
<thead>
<tr>
<th>Good</th>
<th>Bad</th>
</tr>
</thead>
<tbody>
<tr>
<td>Network interface discovery</td>
<td>Network Discovery of CPU cores</td>
</tr>
<tr>
<td>CPU core discovery</td>
<td></td>
</tr>
</tbody>
</table>

Items, triggers, graphs names generated from LLD should be prefixed with the discovery entity name they belong to. The only exception is the singleton discovery pattern.

1.4.2 Update interval

Use 1h. For advanced usage, see the best practices section.

1.4.3 Keep lost resources period

Keep it to default: 30d.

1.5 Triggers and problems

1.5.1 Naming

Trigger names must be prefixed with LLD object they belong to.

Trigger names should not use {HOST.NAME} macro to keep names shorter. Consider getting this data from the host column.

Avoid using {ITEM.LASTVALUE} in trigger name

Don’t use {ITEM.LASTVALUE1-9} macros right in trigger names. As of 4.0 they these macros are expanded to values when problem name is generated and stays.

Use it in an operational data field instead. (available in Zabbix 4.4)

Explain the threshold in name

Consider explaining why trigger fired (threshold) in parenthesis ().
1.5.2 Trigger description

Use this field to describe:

- What possible problem is being evaluated
- Why it is important to check this
- Provide a reference to the documentation if possible

1.5.3 Expressions

Trigger expressions should be reasonably flap-resistant - that is, not relying on the last value only but checking last 5 or 10 minutes instead. On the other hand, do not make the expressions overly complex - for example, do not use trigger hysteresis unless it really adds significant value.

Prefer to use user macros in trigger expressions to allow thresholds tuning.

<table>
<thead>
<tr>
<th>Good</th>
<th>Bad</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>{template:temperature.last()}&gt;{TEMP.MAX.WARN}</code></td>
<td><code>{template:temperature.last()}&gt;30</code></td>
</tr>
</tbody>
</table>

Use newlines and spaces to make long trigger expressions more human readable.

1.5.4 Using time and data suffixes in triggers

Always use time (1m, 5m, 1d...) and size suffixes (1K, 1B, 1G) in trigger expressions and problem names, trigger description, operational data to improve readability. Remember, that you can use them in user macros, too.

<table>
<thead>
<tr>
<th>Good</th>
<th>Bad</th>
</tr>
</thead>
</table>
| `{template:temperature.avg(10m)}>{TEMP.MAX.WARN}`
`{template:memory.free.avg(10m)<{MEM_FREE.WARN}
where {MEM_FREE.WARN} = 100M` | `{template:temperature.avg(600)}>{TEMP.MAX.WARN}`
`{template:memory.free.avg(10m)<{MEM_FREE.WARN}
where {MEM_FREE.WARN} = 104857600` |

1.5.5 Severity

Triggers created in the templates are mapped to standard Zabbix severity scale. Consider choosing severity assigned to the trigger with the following in mind:

<table>
<thead>
<tr>
<th>Severity</th>
<th>Description</th>
<th>Examples</th>
<th>Expected reaction type and time (not always true!), given as example only</th>
</tr>
</thead>
<tbody>
<tr>
<td>Not classified</td>
<td>Not used under normal circumstances</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Zabbix Documentation 4.4 - https://www.zabbix.com/documentation/4.4/
### Severity

<table>
<thead>
<tr>
<th>Severity</th>
<th>Description</th>
<th>Examples</th>
<th>Expected reaction type and time (not always true!), given as example only</th>
</tr>
</thead>
<tbody>
<tr>
<td>Info</td>
<td>The event happened that is not an alarm at all. This is the info that might be helpful in future for retrospective analysis or for auditing.</td>
<td>Examples: s/n changed, user logged in, etc</td>
<td>None</td>
</tr>
<tr>
<td>Warning</td>
<td>A minor alarm that could lead to some more serious problem if left without attention.</td>
<td>Examples: Disk usage is low but there is still some room</td>
<td>React during working hours, no notification is expected.</td>
</tr>
<tr>
<td>Average</td>
<td>Performance alarms: Average alarm that indicates serious performance problems or key service degradation. Fault alarms: partial resource failure or warnings that if left without attention might lead to complete device fault.</td>
<td>Examples: CPU utilization is high, Low memory, High device temperature, Disk health failure in the disk array, Website is slow.</td>
<td>React during working hours, create an issue ticket if the problem stays for hours.</td>
</tr>
<tr>
<td>High</td>
<td>Performance alarms: Key service is not available. Fault alarms: Device is not functioning or not available.</td>
<td>No ICMP PING, Website is down.</td>
<td>React off working hours if affects services with the page. React with a ticket during working hours otherwise.</td>
</tr>
<tr>
<td>Disaster</td>
<td>Reserved for alarms indicating blackouts, disasters, global business service faults. There should be no triggers with disaster level severity in resource templates.</td>
<td>Riga DC is down, Level core network is down, &gt;50% of users cannot purchase anything from our website.</td>
<td>Always react with by paging the responsible person.</td>
</tr>
</tbody>
</table>

### 1.6 Graphs and dashboards

#### 1.6.1 Graph names

Graph names must be prefixed with low-level discovery object they belong to.

Graph names can also be prefixed with resource.

#### 1.7 User macros

User macros and low-level macros accept only uppercase characters, that is [A-Z0-9._].

Consider using template specific prefix (namespace) to avoid potential conflicts with other templates.
Use macro context in objects from LLD. This way you can change and tune macros not only on host level but on LLD entity level.

<table>
<thead>
<tr>
<th>Good</th>
<th>Ok</th>
<th>Bad</th>
</tr>
</thead>
<tbody>
<tr>
<td>₹MYSQL./host</td>
<td>₹MYSQL_HOST</td>
<td>₹HOST</td>
</tr>
<tr>
<td>₹MYSQL.PORT</td>
<td>₹MYSQL_PORT</td>
<td>₹PORT</td>
</tr>
<tr>
<td>₹MYSQL.PARAM1</td>
<td>₹MYSQL_PARAM1</td>
<td>₹PARAM1</td>
</tr>
</tbody>
</table>

Use only widely accepted word shortenings in macro names, such as:

WARNING – WARN
CRITICAL – CRIT
PERCENTAGE – PCT
TEMPERATURE – TEMP
ERRORS – ERR
DISCARDS – DISC
HOSTNAME - HOST
DATABASE – DB
PASSWORD – PASS
USERNAME - USER
THRESHOLD - THRESH
CONNECTIONS - CONN
MAXIMUM – MAX
MINIMUM – MIN
AVERAGE – AVG
SECOND – SEC

If there is no good short-form - prefer to set macro name long but clearly understandable.

1.7.1 Trigger macros

For macros used in trigger expressions (thresholds) use form:

```
{[$<NAMESPACE>...]<METRIC_NAME>[.MAX|MIN][.OK | .WARN| .CRIT]}
```

Use MAX|MIN when you need to highlight whether it is the high or low threshold.

<table>
<thead>
<tr>
<th>Good</th>
<th>Bad</th>
</tr>
</thead>
<tbody>
<tr>
<td>₹MYSQL.REPLICATION_LAG.MAX.WARN</td>
<td>₹DISK_OK_STATUS</td>
</tr>
<tr>
<td>₹TEMP.MAX.WARN:”{#SENSOR}”</td>
<td>₹MEMORY_UTIL_MAX</td>
</tr>
<tr>
<td>₹SERVICE.STATUS.CRIT</td>
<td></td>
</tr>
<tr>
<td>₹IF.ERRORS.MAX.WARN</td>
<td></td>
</tr>
<tr>
<td>₹DISK.STATUS.OK</td>
<td></td>
</tr>
<tr>
<td>₹DISK.STATUS.WARN</td>
<td></td>
</tr>
<tr>
<td>₹DISK.STATUS.CRIT</td>
<td></td>
</tr>
<tr>
<td>₹MEM_UTIL.MAX.WARN</td>
<td></td>
</tr>
<tr>
<td>₹MEM_UTIL.MAX.CRIT</td>
<td></td>
</tr>
</tbody>
</table>
1.8 Files

Share your work as an XML file. Name the file just like your template, but all lowercase with spaces replaced by _

<table>
<thead>
<tr>
<th>Good</th>
<th>Bad</th>
</tr>
</thead>
<tbody>
<tr>
<td>template_app_nginx.xml</td>
<td>Template App Nginx.xml</td>
</tr>
<tr>
<td>template_db_mysql.xml</td>
<td>Template_DB_MySQL.xml</td>
</tr>
</tbody>
</table>

Store each template files in their own, separate directory. Create a README.md file or similar in this directory to describe what this template does and how to install it. Place user parameter files or any other files required to run this template into this directory as well.

1.8.1 Readme file structure

It is very important to provide a clear explanation of what your template does, how it can be installed, configured and tuned. Consider providing such documentation in the README file. Readme file should contain the following sections:

**Overview**

Describe what this template is about, what versions of monitored object it was tested on.

**Setup**

Provide clear step-by-step instructions on how to install the template.

**Zabbix configuration**

Provide info here how the template can be tuned using macros and so on.

**Template links**

List all template links if any.

**Discovery rules**

List discovery rules with filters applied.

**Items collected**

List all items being collected.

**Triggers**

List all triggers.

**Feedback**

Describe how to provide feedback.
Demo

Optional. Provide some screenshots from the template in action.

References

Optional. Provide any links to any templates that inspired you to create this one, or reference to the official documentation about the monitored object.

Best practices

2.1 Discovering items and tackling unsupported items

Use low-level discovery as much as possible. This helps to avoid unsupported items as well as to improve templates flexibility.

<table>
<thead>
<tr>
<th>Good</th>
<th>Bad</th>
</tr>
</thead>
<tbody>
<tr>
<td>Discovering temperature sensors using LLD</td>
<td>CPU core #1 utilization, Sensor 1 temperature value, network interface Fa0/0 directly by statically creating items without using LLD</td>
</tr>
<tr>
<td>Discovering network interface using LLD</td>
<td></td>
</tr>
<tr>
<td>Discovering CPU cores using LLD</td>
<td></td>
</tr>
</tbody>
</table>

2.1.1 Discovery frequency

Low-level discovery is considered a heavy operation in Zabbix, so its frequency should be low. Consider to always start at 1 per hour.

If discovery uses another frequent item as a source (Item type = dependent item) - apply “Discard unchanged with heartbeat” preprocessing for such discovery. You can also use such preprocessing for other discoveries too.

In such case, you can also use discovery preprocessing to filter out toggling parts of the low-level discovery data, for example, for data coming from master item:

```json
[
  {
    "volume_name": "my disk1",
    "volume_size": 1000000000000,
    "volume_used": 800000000000,
    "volume_updated_at": "2019-07-01 00:00:00"
  },
  {
    "volume_name": "my disk2",
    "volume_size": 1000000000000,
    "volume_used": 800000000000,
    "volume_updated_at": "2019-07-01 00:00:00"
  }
]
```
For such output consider transforming this array using JS or JSONPath preprocessing to:

```json
[ 
  { 
    "volume_name": "my disk1"
  },
  { 
    "volume_name": "my disk2"
  }
]
```

Before applying throttling discard rule.

### 2.1.2 Discovery with Zabbix trapper

When pushing items via Zabbix trapper protocol – consider pushing low-level discovery data as well since discovery items support it.

### 2.1.3 Use preprocessing to build low-level discovery on the fly

With JavaScript preprocessing and other powerful features, you can create low-level discovery data on the fly. Prefer this method over external discovery scripts:

- To keep discovery rules clearly observable by all future template users
- To keep discovery as a part of the monitoring solution – easily transferable as part of the template
- To avoid external dependencies such as external discovery scripts

**Example 1**

Get Nginx Plus zones stats using Zabbix HTTP agent from URL such as this:

```json
{"hg.nginx.org": {
  "processing": 0,
  "requests": 175276,
  "responses": {
    "1xx": 0,
    "2xx": 162948,
    "3xx": 10117,
    "4xx": 2125,
    "5xx": 8,
    "total": 175198
  },
  "discarded": 78,
  "received": 50484208,
}}
```
Feed this output to discovery rule via dependent item and apply Javascript preprocessing as this:

```javascript
//parsing NGINX plus output:
output = Object.keys(JSON.parse(value)).map(function(zone){
    return {"{#NGINX_ZONE}" : zone}
})
return JSON.stringify({"data" : output})
```

Making original JSON object a fully LLD compatible JSON Array that can be used for NGINX zones discovery.

**Example 2**

Get disks stats using Zabbix agent vfs.file.contents[/proc/diskstats] item:
Feed this output to regular item and then apply preprocessing as this:

**JAVASCRIPT**

```javascript
var parsed = value.split("\n").reduce(function(acc, x, i) {
    acc["values"] = x.split(/ +/)[3] = x.split(/ +/).slice(1)
    acc["lld"][i] = "{#DEVNAME}":x.split(/ +/)[3];
    return acc;
}, {}, {"values":{}, "lld":[]});

return JSON.stringify(parsed);
```

Create new discovery rule with the item above as the master item. Apply additional preprocessing to this discovery rule:

**JSONPATH**

`.lld`

### 2.1.4 Singleton discovery

While low-level discovery was designed to automate the creation of items, triggers and graphs for multiple similar entities such as network interfaces or disks, it can also be used as a simple filter for exclusive entities that either doesn't exist or exist in the single instance.

This approach allows keeping template clean, without users face unsupported items when template is applied to hosts with different configurations or versions of the monitored object.

To use singleton pattern, you need to do the following:

- Create discovery rule. Use regular items or dependent items to get some value that is not in LLD format. For brief example, lets it a regular item that returns text ‘found’ or ‘missing’.
- Use preprocessing in low-level discovery rule:
  - Check that received value matches your conditions and that items should be created
  - Using Javascript preprocessing, add an empty LLD macro named `{#SINGLETON}` inside LLD array of length 1

These two steps can be combined in the single line of JavaScript that would generate an LLD array.

```javascript
return JSON.stringify(value === 'found' ? [{'{#SINGLETON}': ''}] : []);
```

Use this macro `{#SINGLETON}` inside square brackets of all item prototypes keys.

Append this macro to any graph prototype name.

Empty macro is required, so Zabbix can differentiate item or graph from the prototype. When macro is expanded after discovery - only clean item name or graph name can be seen absolutely identical to
the one that you would statically define.

See MPM event discovery in Zabbix 4.4 “Template App Apache by HTTP” template as an example. We will also describe it in more detail in our blog.

<table>
<thead>
<tr>
<th>Good</th>
<th>Bad</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPM event singleton discovery in Template App Apache by HTTP (Zabbix 4.4)</td>
<td>Templates that monitor Apache HTTP server without such Singleton approach, thus leaving MPM event metrics as not supported when MPM event module is disabled</td>
</tr>
</tbody>
</table>

### 2.2 Getting items

#### 2.2.1 Minimize external libraries dependencies when writing external scripts/modules if possible

If you need to resort to external scripts – think about making them portable and easy to install as well.

#### 2.2.2 Preprocessing

Prefer to use Zabbix preprocessing in favour of complex data parsing with some scripts on the agent side:

- To keep Zabbix agent presence noticeable as less as possible
- To keep preprocessing rules clearly observable by all future template users
- To keep preprocessing rules as a part of the monitoring solution - easily transferable as part of the template
- To avoid maintaining two sets of preprocessing rules on Windows and Linux platforms

#### 2.2.3 Master item + dependent items/preprocessing

Prefer to use Zabbix master item + dependent items in favor of multiple separate calls:

- To keep Zabbix presence noticeable as less as possible - fewer calls to the monitored objects

Reuse master item contents to create Low-level discovery rules. Then reuse master item values again to be used in future items from prototypes.

**Master item history storage period**

Master items values may be of a very large size (ZBXNEXT-223), while these values are only needed for preprocessing in dependent items. So, minimize its history storage period to a minimum, non-zero value which is 1h.

#### 2.2.4 Security and authentication

While passing passwords as user macros may sound like a convenient idea – avoid as much as you can.
If you need to authenticate in order to gather metrics – prefer to create user named **zbx_monitor** with read-only access.

### 2.2.5 Getting data with user parameters/external check

Prefer using user parameters/external check or modules with dependent items/preprocessing over Zabbix trapper if you can, since when using Zabbix trapper you have less control over data collection.

### 2.2.6 Getting data with Zabbix trapper

Prefer using Zabbix trapper over user parameters/external check if one of the following statements are true:

- You need to send metrics from your own custom applications
- Data collection is irregular (backup job, alarm signal, etc)
- You need to send data with shifted timestamp
- Data collection script can take more than 30 seconds to complete

### 2.3 Healthchecks and discrete states

Always use value mappings for discrete states passed as integers.

Consider using “Boolean to decimal” preprocessing if item check result can only have two states such as YES/NO, TRUE/FALSE to preserve DB space and then apply simple value mapping.

Consider using “Discard unchanged with heartbeat” preprocessing for discrete states. This will improve state change reaction dramatically without putting additional load on Zabbix DB. Start with something like 10s/5m or 1m/30m. Note though that trigger functions such as count() or diff() may work differently.

#### 2.3.1 Healthchecks and discrete states triggers

For health check triggers consider using simple trigger expression:

```
{TEMPLATE_NAME:METRIC.count(#1,${SERVICE.STATUS.CRIT},eq)}=1
```

If your health check metric that returns only integer values and not text statuses, you may also use:

```
{TEMPLATE_NAME:METRIC.last()}=${SERVICE.STATUS.CRIT}
```

for simplicity.

If your health check can return multiple different values, try to map them to the following triggers of different severity (simplified scale):
### Zabbix template guidelines

<table>
<thead>
<tr>
<th>Level</th>
<th>Suggested Zabbix severity</th>
<th>Trigger name</th>
<th>Trigger dependencies</th>
<th>Sample expressions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Not OK</td>
<td>Information</td>
<td>Service X is not OK</td>
<td>depends on warning and critical level triggers</td>
<td><code>{TEMPLATE_NAME:METRIC.count(#1,{$SERVICE.STATUS.OK},ne)}=1</code></td>
</tr>
<tr>
<td>Warning</td>
<td>Warning</td>
<td>Service X is in warning state</td>
<td>depends on critical level trigger</td>
<td><code>{TEMPLATE_NAME:METRIC.count(#1,{$SERVICE.STATUS.WARN},eq)}=1</code></td>
</tr>
<tr>
<td>Critical</td>
<td>High or Average</td>
<td>Service X is in critical state</td>
<td></td>
<td><code>{TEMPLATE_NAME:METRIC.count(#1,{$SERVICE.STATUS.CRIT},eq)}=1</code></td>
</tr>
</tbody>
</table>

Use 'Not OK' level if there are too many bad statuses or not all of them known.

```
{TEMPLATE_NAME:METRIC.count(#1,{$SERVICE.STATUS.OK},ne)}=1
```

Note 'ne' in Not OK expression.

If there are multiple metric values all indicating critical level, put them together in the single expression:

```
{TEMPLATE_NAME:METRIC.count(#1,{$SERVICE.STATUS.CRIT:"not_responding"},eq)}=1 or {TEMPLATE_NAME:METRIC.count(#1,{$SERVICE.STATUS.CRIT:"timeout"},eq)}=1
```

Note that you may use macros context to label different statuses.

For noisy items, consider adding recovery expression:

```
{TEMPLATE_NAME:METRIC.count(5m,{$SERVICE.STATUS.CRIT},eq)}=0
```

### 2.4 Collecting inventory and text description states

Consider using “Discard unchanged with heartbeat” preprocessing for inventory and other textual data that rarely changes. This will improve inventory change reaction dramatically without putting additional load on Zabbix DB. Start with something like 15m/1d. Note though that trigger functions such as count() or diff() may work differently.

Always use this preprocessing step if rarely changing inventory field is collected from a general master item that is frequently polled.

### 2.5 Use trigger snippets

Check the following trigger snippets library and consider reusing configuration to avoid reinventing the wheel.

**Case: Something has just been restarted**
**Trigger:** `<resource>` has just been restarted (uptime < 10m)

### Applicable for
For uptime counters for device, host, or software/service running

<table>
<thead>
<tr>
<th>Name</th>
<th><code>&lt;resource&gt;</code> has been restarted (uptime &lt; 10m)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Description</td>
<td><code>&lt;resource&gt;</code> uptime is less than 10 minutes</td>
</tr>
<tr>
<td>Expression</td>
<td><code>{TEMPLATE_NAME:METRIC.last} &lt; 10m</code></td>
</tr>
<tr>
<td>Recovery expression</td>
<td>-</td>
</tr>
<tr>
<td>Recovery mode</td>
<td>-</td>
</tr>
<tr>
<td>Manual close</td>
<td>Yes</td>
</tr>
<tr>
<td>Severity</td>
<td>Warning for the host. Info for all others.</td>
</tr>
<tr>
<td>Depends on</td>
<td>-</td>
</tr>
</tbody>
</table>

### Case: Any master item + preprocessing in dependent items

**Trigger:** Master item is not responding

`<resource>`: Failed to get items (no data for 30m)

### Applicable for
Any type of items used for bulk data collection

<table>
<thead>
<tr>
<th>Expression</th>
<th><code>{TEMPLATE_NAME:METRIC.nodata(30m)} = 1</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>Recovery expression</td>
<td>-</td>
</tr>
<tr>
<td>Recovery mode</td>
<td>-</td>
</tr>
<tr>
<td>Manual close</td>
<td>Yes</td>
</tr>
<tr>
<td>Severity</td>
<td>Warning</td>
</tr>
<tr>
<td>Depends on</td>
<td>If present: <code>&lt;Proc&gt;</code> is not running</td>
</tr>
</tbody>
</table>

### Case: HTTP item + regex preprocessing in dependent items

**Trigger:** HTTP item is not responding

### Applicable for
HTTP items that provide output for future regex preprocessing. Use 'Headers and Body' mode in the item.

| Expression | `{TEMPLATE_NAME:METRIC.str("HTTP/1.1 200\")} = 0 or
{TEMPLATE_NAME:METRIC.nodata(30m)} = 1` |
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Recovery expression</td>
<td>-</td>
</tr>
<tr>
<td>Recovery mode</td>
<td>-</td>
</tr>
<tr>
<td>Manual close</td>
<td>Yes</td>
</tr>
<tr>
<td>Severity</td>
<td>Warning</td>
</tr>
<tr>
<td>Depends on</td>
<td>If present: <code>&lt;Proc&gt;</code> is not running</td>
</tr>
</tbody>
</table>

### Case: `<VALUE>` is too high (over X)/ is too low (under X) for slow to change values

For slow changing values (i.e. temperature, use max() for high, and min() for lows to get immediate response with delayed (confirmed) recovery.

**Trigger:** `<VALUE>` is too high (over X)

### Applicable for
High temperature (slow to change)

<table>
<thead>
<tr>
<th>Expression</th>
<th><code>{TEMPLATE_NAME:METRIC.max(5m)} &gt; X</code></th>
</tr>
</thead>
</table>
Trigger: `<VALUE>` is too low (under X)

<table>
<thead>
<tr>
<th>Applicable for</th>
<th>Low temperature (slow to change)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Expression</td>
<td><code>{TEMPLATE_NAME:METRIC.min(5m)} &lt; X</code></td>
</tr>
</tbody>
</table>

Case: `<VALUE>` is too high (over X for 5m)/ is too low (under X for 5m) for quick-to-change and jumpy values

For jumpy values, use min (for high) and max (for low) to make triggers more tolerable to spikes/noise.

Trigger: `<VALUE>` is too high (over X for 5m)

<table>
<thead>
<tr>
<th>Applicable for</th>
<th>CPU utilization (jumpy), signal strength (jumpy), network utilization</th>
</tr>
</thead>
<tbody>
<tr>
<td>Expression</td>
<td><code>{TEMPLATE_NAME:METRIC.min(5m)} &gt; X</code></td>
</tr>
</tbody>
</table>

Trigger: `<VALUE>` is too low (under X for 5m)

<table>
<thead>
<tr>
<th>Applicable for</th>
<th>CPU utilization (jumpy), signal strength (jumpy), network utilization</th>
</tr>
</thead>
<tbody>
<tr>
<td>Expression</td>
<td><code>{TEMPLATE_NAME:METRIC.max(5m)} &lt; X</code></td>
</tr>
</tbody>
</table>

Case: Serial number has changed on the device

Trigger: Serial numbers controls

<table>
<thead>
<tr>
<th>Applicable for</th>
<th>Serial numbers items</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td><code>&lt;resource&gt; has been replaced (new serial number received)</code></td>
</tr>
<tr>
<td>Description</td>
<td><code>&lt;resource&gt; serial number has changed. Ack to close</code></td>
</tr>
<tr>
<td>Expression</td>
<td><code>{TEMPLATE_NAME:METRIC.diff()}=1 and {TEMPLATE_NAME:METRIC.strlen()}&gt;0</code></td>
</tr>
<tr>
<td>Recovery expression</td>
<td>-</td>
</tr>
<tr>
<td>Recovery mode</td>
<td>None</td>
</tr>
<tr>
<td>Manual close</td>
<td>Yes</td>
</tr>
<tr>
<td>Severity</td>
<td>Info</td>
</tr>
<tr>
<td>Depends on</td>
<td>-</td>
</tr>
</tbody>
</table>

Case: Software version has changed on the device

Trigger: Version controls

<table>
<thead>
<tr>
<th>Applicable for</th>
<th>Software version items</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td><code>&lt;resource&gt; version has changed (new version: {ITEM.VALUE})</code></td>
</tr>
<tr>
<td>Description</td>
<td><code>&lt;resource&gt; version has changed. Ack to close</code></td>
</tr>
<tr>
<td>Expression</td>
<td><code>{TEMPLATE_NAME:METRIC.diff()}=1 and {TEMPLATE_NAME:METRIC.strlen()}&gt;0</code></td>
</tr>
<tr>
<td>Recovery expression</td>
<td>-</td>
</tr>
<tr>
<td>Recovery mode</td>
<td>None</td>
</tr>
<tr>
<td>Manual close</td>
<td>Yes</td>
</tr>
<tr>
<td>Severity</td>
<td>Info</td>
</tr>
<tr>
<td>Depends on</td>
<td>-</td>
</tr>
</tbody>
</table>
2.6 Visualization, graphs, and dashboards

Consider adding custom graphs for items that could be correlated.

**Good**: Graph containing all items of different CPU modes (user, system...)

Consider adding dashboards (screens) to provide monitored object summary or a quick overview.

**Good**: Template App Zabbix Server is a good example

2.7 Usage of event tags

This section will be filled in the next version of the document.