



Databbix

User Manual

Real-time data collector for Zabbix

Software version 2.5

September 2026

Databbix — User Manual

Copyright © 2014–2026 Trustserver SL. All rights reserved.

This document is distributed under the terms of the *Common Development and Distribution License* (CDDL), version 1.0. You may not use this document except in compliance with the terms of that license. You may obtain a copy of the license at <https://opensource.org/licenses/CDDL-1.0>.

This document is provided “as is”, without warranty of any kind, either expressed or implied, including, without limitation, warranties that the document is free of defects, merchantable, fit for a particular purpose or non-infringing. The entire risk as to the quality and accuracy of the document is with you. In no event shall Trustserver SL be liable for any direct or indirect damages arising from its use.

Databbix is free software, distributed under the GNU General Public License, version 2 or later.

Zabbix is a registered trademark of Zabbix LLC. Oracle and JD Edwards are registered trademarks of Oracle Corporation and its affiliates. Microsoft and SQL Server are registered trademarks of Microsoft Corporation. All other trademarks are the property of their respective owners.

Trustserver SL

<https://trustserver.com/en/>

Revision history

Version	Date	Description
1.0	2026-09-29	First version of the user manual, for Databbix 2.5.

Contents

I	Fundamentals	1
1	Introduction	3
1.1	What Databbix is	3
1.2	How this manual is organised	3
2	Architecture and operation	5
2.1	One process, one thread per monitor	5
2.2	Configuration files	6
2.2.1	Value inheritance	7
2.3	Monitor life cycle	7
3	Concepts	9
3.1	Host, item and key	9
3.2	Monitor	9
3.3	Query engine	10
3.4	Query collection	11
3.5	Query	11
3.6	Query item	11
3.7	Parameter	11
3.8	Module	12
3.9	Discovery	12
3.10	Context	13
3.11	Environment variable	13
3.12	Runtime data	13
3.13	Scripts and transformations	14
3.14	Caching and scheduling	14
3.15	Web service authentication	15
II	Common configuration	17
4	Installation and files	19
4.1	File locations	19
4.2	Bundled query files	19
4.3	Environment variables	20
5	Daemon configuration	21
5.1	Command line	21
5.2	The databbix.conf file	21
5.2.1	[General] section	21
5.2.2	[Log] section	22
5.2.3	Example	22

5.3	systemd service	22
5.4	Signals	23
6	Monitor configuration: common part	25
6.1	File structure	25
6.2	[General] section	26
6.3	[Zabbix] section	26
6.4	[Data] section	26
6.5	[Log] section	27
6.6	Special characters	27
7	XML query files: common part	29
7.1	What is common and what depends on the engine	29
7.2	General rules	30
7.3	The <Queries> element	30
7.4	Environment variables: <environment>	31
7.5	Scripts: <script>	31
7.6	The query: <query>	32
7.6.1	Common attributes	32
7.6.2	Query types per engine	32
7.7	The <item> element	32
7.8	The <parameter> element	33
7.9	Discovery queries	33
7.10	Constant queries	33
8	Zabbix configuration	35
8.1	Item keys	35
8.2	Internal items	35
III	Engines	37
9	Database engine	39
9.1	Configuration: the [Database] section	39
9.2	Drivers	39
9.2.1	Oracle driver	40
9.2.2	PGsql driver	40
9.2.3	ODBC driver	40
9.2.4	Runtime data	41
9.3	SQL queries: engine-specific elements	41
9.4	Query types	42
9.4.1	Basic query	42
9.4.2	Filter query (select="FILTER")	43
9.4.3	Bind variable query (select="BIND")	44
9.4.4	Discovery query	45
9.5	Contexts	46
9.5.1	Declaration	46
9.5.2	Use in queries	46
9.5.3	Example: a filter inside each database	47
9.6	Alternative SQL depending on the database	48
9.7	Configuration examples	48
9.7.1	Example: Oracle	48

9.7.2	Example: Oracle multitenant	49
9.7.3	Example: ASM	49
9.7.4	Example: PostgreSQL	49
9.7.5	Example: SQL Server through ODBC	50
10	Webservice engine	51
10.1	REST and SOAP	51
10.2	Configuration: the [Webservice] section	51
10.3	REST queries	52
10.3.1	JSON paths	53
10.3.2	Parameters in the URL	54
10.3.3	Filtering the response	54
10.3.4	REST discoveries	55
10.4	SOAP queries	56
10.4.1	Converting the SOAP response to JSON	57
10.4.2	Transforming SOAP responses	58
10.5	Runtime data	59
10.6	Authentication	60
10.6.1	Authentication engines and methods	60
10.6.2	How the authentication is chosen	60
10.6.3	Examples	60
11	MQTT engine	63
11.1	Configuration: the [MQTT] section	63
11.2	Query files	64
11.3	Reading values	64
11.4	Topic discovery	65
11.5	Runtime data	66
IV	Tools and usage	67
12	Script programming	69
12.1	Available languages	69
12.1.1	JavaScript	69
12.1.2	Python	69
12.2	Where scripts are written	69
12.3	Execution environment	70
12.3.1	Scope and state	70
12.3.2	Logging functions	71
12.3.3	Data types	71
12.3.4	Errors	71
12.4	Transforming values in SQL queries	72
12.4.1	Example: dates	72
12.4.2	Example: mapping states to numbers	72
12.4.3	Python example	73
12.5	Transforming discoveries	73
12.6	Transforming web service responses	74
12.6.1	Handling error codes	74
12.6.2	Restructuring the output	75
12.6.3	Filtering the output	75
12.6.4	SOAP responses	76

12.6.5	Combining with item transformations	76
12.7	Programming web service authentication methods	76
12.7.1	Credentials	76
12.7.2	Call verification	77
12.7.3	Methods based on the RESTToken engine	77
12.7.4	Methods based on the HTTPAuth engine	78
13	The queryexplorer utility	81
13.1	Invocation	81
13.2	Interactive mode	82
13.3	Non-interactive mode	82
13.4	Commands	83
13.4.1	Items and queries	83
13.4.2	Contexts	83
13.4.3	Variables and expressions	83
13.4.4	Other	84
13.5	Use cases	84
13.5.1	Validating a monitor file and its connection	84
13.5.2	Validating web service authentication	84
13.5.3	Testing a new query	84
13.5.4	Checking alternative SQL	84
13.5.5	Checking contexts	85
14	Practical guide: creating a new query	87
14.1	Step 1: write and test the SQL	87
14.2	Step 2: choose the query type	87
14.3	Step 3: write the XML	87
14.4	Step 4: add the file to the monitor	88
14.5	Step 5: test with queryexplorer	88
14.6	Step 6: create the items in Zabbix	89
15	Troubleshooting	91
15.1	Log levels	91
15.2	Common problems	91
V	Appendices	93
A	XML quick reference	95
A.1	<Queries> level (all engines)	95
A.2	Constant query (all engines)	95
A.3	Database engine	95
A.3.1	Basic query	95
A.3.2	Filter query	96
A.3.3	Bind variable query	96
A.3.4	Discovery query	96
A.4	Webservice engine	97
A.4.1	REST query	97
A.4.2	REST filter query	97
A.4.3	REST discovery query	97
A.4.4	SOAP query	98
A.5	MQTT engine	98

A.5.1	Value reading	98
A.5.2	Topic discovery	98
B	Default values	99

Part I

Fundamentals

Chapter 1

Introduction

1.1 What Databbix is

Databbix is a daemon that collects data from external sources and sends it to a Zabbix server. The sources can be SQL databases (Oracle, PostgreSQL, SQL Server and any other database reachable through ODBC), REST or SOAP web services and MQTT brokers. Towards Zabbix, Databbix behaves as an *active agent*: it asks the server for the list of items it has to collect for a host, computes them by running queries and sends the results.

Unlike the Zabbix agent's *user parameters*, Databbix:

- keeps the database connection open between executions;
- runs each query *only once* for all the items that use it, and shares the result rows among them;
- caches results for a configurable time;
- defines queries in XML files, with no programming required, and supports Zabbix low-level discovery (LLD).

1.2 How this manual is organised

Part I, Fundamentals. How Databbix works and the concepts used throughout the manual. Read it before configuring anything.

Part II, Common configuration. Installation, the daemon, the common part of monitor files and XML query files, and the Zabbix configuration. Everything described here is the same for any data source.

Part III, Engines. One chapter per engine (Database, Webservice and MQTT) with its configuration section, its XML elements and examples with data and results.

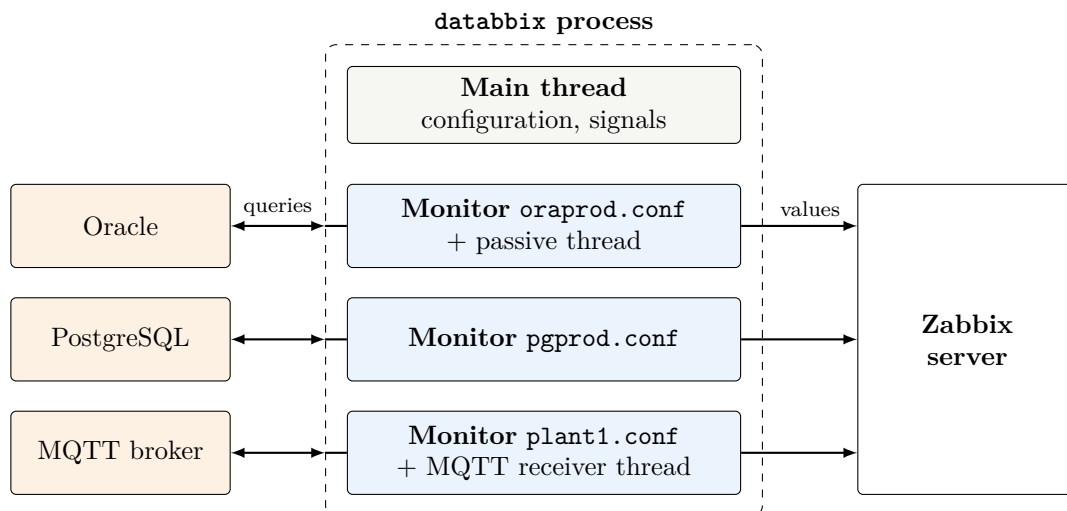
Part IV, Tools and usage. Script programming, the `queryexplorer` utility, a practical guide and troubleshooting.

Chapter 2

Architecture and operation

2.1 One process, one thread per monitor

Databbix runs as a single process, `databbix`, that serves all the data sources on the machine. Inside the process, each data source is served by its own thread, called a *monitor*:



Main thread. Reads the global configuration, walks the monitor directory and starts one thread per `.conf` file. It then waits: it reopens the log files when it receives `SIGHUP` and tells all monitors to stop when it receives `SIGTERM` or `SIGINT`.

Monitor thread. There is one per `.conf` file. Each monitor has its own connection to the data source, its own connection to Zabbix and its own log file.

Passive thread. If the monitor has a non-zero `PassivePort`, it starts an additional thread that listens on that port and answers Zabbix passive checks.

Engine threads. Some engines need threads of their own. The MQTT engine, for example, has a thread that continuously receives the messages from the broker.

Monitors are independent of each other:

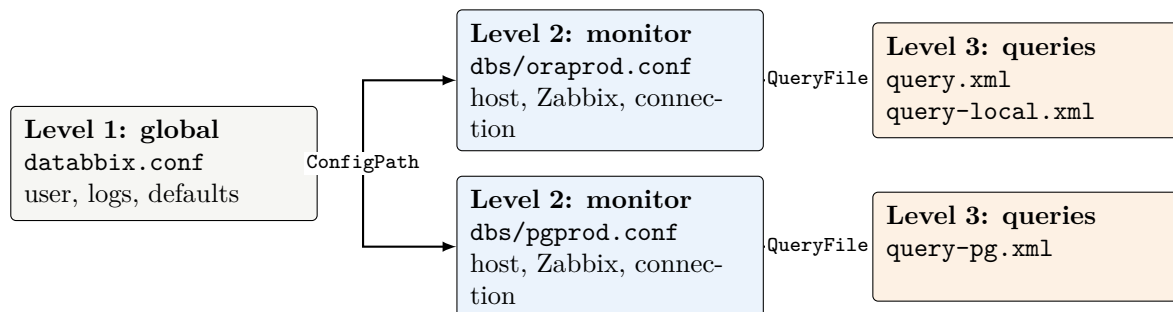
- If a data source fails or is down, only its monitor is affected. The rest keep working.
- If a monitor cannot start (for example, because of an error in its `.conf` or in an XML file), that monitor ends and logs the error, but the process keeps serving the others. The monitor is not restarted until the service is restarted.

- A monitor also ends if it exceeds `MaxErrors` consecutive failed executions.
- When all monitors have ended, the process ends.

Warning. Since a failing monitor does not stop the process, `systemctl status` may report the service as active even if some monitors are not running. To watch for this, use the passive item `agent.ping` or check in Zabbix that data arrives from every host.

2.2 Configuration files

The configuration is split into three levels:



File	Format	Contents
<code>databbix.conf</code> (one)	INI	Process configuration: daemon user and group, monitor directory, modules, and defaults for logs, query files and environment variables for all monitors.
<code>dbs/*.conf</code> (one per monitor)	INI	Everything specific to a data source: Zabbix host name, Zabbix server, engine, connection data (driver, connection string, user, URL, authentication...), query files and its own environment variables.
<code>query*.xml</code>	XML	Which data is obtained and how: SQL queries or service calls, the Zabbix items each one produces, contexts, scripts and authentications. They contain no connection data, so the same XML file serves every source of the same kind.
<code>tnsnames.ora</code> , <code>odbc.ini</code>	native	Definition of the Oracle and ODBC connections referred to by the monitor files.
<code>/etc/sysconfig/databbix</code>	shell variables	Service environment (<code>TNS_ADMIN</code> , <code>ODBCINI...</code>).
<code>AuthPersistFile</code> (optional)	JSON	State of a web service authentication (for example, the token), so that it does not have to authenticate again after a restart. Written by Databbix.

2.2.1 Value inheritance

Some parameters can be set at the first two levels. In that case, the value in the monitor file takes precedence over the one in `databbix.conf`:

- Parameters in the [Log] section are inherited one by one. A monitor can, for example, raise only its `LogLevel`.
- **QueryFile**: if the monitor defines any, its list completely *replaces* the global one.
- **Environment**: the monitor sees both the global variables and its own. If a variable is defined in both places, the monitor's value wins. Both take precedence over the XML defaults.

2.3 Monitor life cycle

On startup, each monitor does the following:

1. Reads its `.conf` file and checks that the mandatory parameters are present.
2. Loads the query collection from its XML files. If there is a syntax error, the monitor ends.
3. Connects to the data source. If it cannot, it retries later.
4. Asks the Zabbix server for the list of active items of the host (*active checks*).
5. If configured, starts the passive thread.

It then enters a loop. On each iteration it picks the first applicable task and sleeps until it is due:

1. If it does not have the item list yet, it requests it from Zabbix.
2. If there is no connection to the data source, it reconnects.
3. If some values could not be sent because of a Zabbix failure, it resends them.
4. If the item list is due for a refresh (every `Zabbix.Refresh` seconds), it requests it again.
5. Otherwise, it runs the queries of the items whose interval has elapsed and sends the results.

When the connection to the source or to Zabbix fails, the monitor waits for `General.RetryOnError` seconds and then retries. If the connection to Zabbix is lost, the collected values are kept and sent when the connection comes back.

Chapter 3

Concepts

This chapter defines the terms used throughout the manual. Each concept comes with an example of how it appears in the `.conf` and `.xml` files. The examples are fragments: the following chapters describe every parameter and attribute.

3.1 Host, item and key

These are Zabbix concepts:

Host. A monitored entity in Zabbix. Each Databbix monitor sends the data of a single host, identified by `General.Name`.

Item. A piece of data that Zabbix periodically collects from a host (number of sessions, free space in a tablespace...). The items served by Databbix are of type *Zabbix agent (active)*.

Key. The item identifier, of the form `name[parameter1,parameter2,...]`. The `name` says which item of which query must be computed, and the parameters say for which object (which tablespace, which database...).

Example. The Zabbix host `ORAPROD` has an active item with the key:

```
tez.ora.tspace.pctfree[USERS]
```

The monitor whose `.conf` has `Name = ORAPROD` looks in its XML files for the `<item>` named `tez.ora.tspace.pctfree` and computes it for the `USERS` tablespace.

3.2 Monitor

A monitor is the unit of work of Databbix: it links *one data source* with *one Zabbix host*. It is defined in a `.conf` file and runs in its own thread. For example, if a machine watches three databases, there are three `.conf` files, three monitors and three hosts in Zabbix.

A monitor is made of:

- an engine and a connection to the data source;
- a query collection;
- a connection to the Zabbix server and the list of the host's active items;
- a scheduler that decides when to run each query.

Example. File `/etc/databbix/dbs/oraprod.conf` with the minimum for an Oracle monitor:

```

[General]
Name = ORAPROD                # host in Zabbix

[Zabbix]
Server = zabbix.example.com   # where data is sent
Port = 10051

[Data]
Module = DB                   # query modules used

[Database]                    # where data comes from
DBDriver = Oracle
ConnectionString = oraprod
UserName = zabbix
Password = secret

```

3.3 Query engine

The engine is the component that knows how to talk to one kind of data source. Each monitor uses a single engine, chosen with **General.Engine**. The engine determines:

- which connection section the monitor file has ([Database], [Webservice] or [MQTT]);
- which elements and attributes the XML files accept (for example, <sql> in Database and <url> in Webservice);
- which query types exist and how values are obtained.

Engine	Data source	What a query is
Database	SQL databases through a <i>driver</i> : Oracle, PGsql or ODBC.	An SQL statement.
Webservice	REST (JSON) or SOAP web services.	An HTTP call to a URL; values are extracted from the response.
MQTT	An MQTT broker.	Reading the last value published on a <i>topic</i> .

All the XML files of a monitor must belong to the monitor's engine: the engine is declared in the **engine** attribute of <Queries>.

Example. A REST service monitor declares the engine in its **.conf**:

```

[General]
Name = JDEPROD
Engine = Webservice
QueryFile = /etc/databbix/query-jde.xml

[Webservice]
BaseURL = https://jde.example.com:9888/restservice

```

and the XML file it uses declares the same engine:

```

<Queries engine="Webservice">
  ...
</Queries>

```

3.4 Query collection

The set of queries resulting from loading the XML files of a monitor (**QueryFile**). Within a collection, query names and item names are unique: this is how Databbix knows which query to run for each key Zabbix asks for.

If several monitors use the same engine, the same files and the same environment variables, they share a single in-memory copy of the collection.

Example. A collection made of the distribution's query file plus a local one:

```
[General]
QueryFile = /etc/databbix/query.xml
QueryFile = /etc/databbix/query-local.xml
```

3.5 Query

A query (**<query>**) defines an operation on the data source (an SQL statement, a service call...) and the Zabbix items obtained from its result. A query can produce many items, and Databbix runs it only once for all of them.

Depending on how the result relates to the items, there are several types: basic, filter, bind, discovery and constant. They are described in chapter 7.

Example. A basic query that produces two items from a single statement:

```
<query name="PGWriter" module="PG">
  <sql><![CDATA[
    select checkpoints_timed, checkpoints_req
    from pg_stat_bgwriter
  ]]>
</sql>
<item name="tez.pg.checkpointttimed" column="checkpoints_timed"/>
<item name="tez.pg.checkpointreq" column="checkpoints_req"/>
</query>
```

3.6 Query item

Within a query, each **<item>** says where the value of a Zabbix item comes from: which column of the SQL result, which path of the JSON response, an internal figure (execution time)... Its **name** attribute is the key name in Zabbix.

Besides the items defined in the XML files, Databbix answers a few internal items, such as **agent.ping** or **agent.version**.

Example. Three ways of getting the value:

```
<!-- from an SQL column -->
<item name="tez.ora.sessions" column="n"/>
<!-- from a path inside a service's JSON response -->
<item name="ts.jde.instance.state" path="instanceState" type="INT"/>
<!-- from an internal figure: seconds the query took -->
<item name="tez.ora.tspace.exectime" built_in="exec_time"/>
```

3.7 Parameter

A parameter (**<parameter>**) links a parameter of the Zabbix key to the query. Depending on the query type, the parameter value is used to:

- choose a row of the result (*filter*);
- pass it to the SQL statement or the URL (*bind*);
- choose the context in which the query runs.

Example. With the key `tez.pg.sessions[sales]`, the first parameter (`sales`) selects the row whose `datname` column is `sales`:

```
<parameter column="datname" order="1" type="STRING"/>
```

With the key `tez.pg.param[max_connections]`, the first parameter is passed to the SQL statement in the `:name` variable:

```
<parameter bind=":name" order="1" type="STRING"/>
```

3.8 Module

A module is a name that groups related queries (DB, Instance, PG, MSSQL...). Each query and each context declares its module with the `module` attribute, and each monitor declares the modules it uses with `Data.Module`.

Example.

```
<query name="Sessions" module="Instance"> ... </query>
```

```
[Data]
Module = DB
Module = Instance
```

3.9 Discovery

Low-level discovery (LLD) is the Zabbix mechanism to create items automatically for the objects that exist (one item per tablespace, per database...). In Databbix, a discovery query returns the list of objects in the JSON format Zabbix expects. It is usually paired with a filter query that computes the values of each discovered object.

Example. Discovery of the databases of a PostgreSQL server:

```
<query name="PGDiscoverDB" module="PG" discovery="yes">
  <sql><![CDATA[
    select datname from pg_database where datname not like 'template%'
  ]]>
</sql>
<item name="tez.pg.db.discovery"/>
<field name="DBNAME" column="datname"/>
</query>
```

The item `tez.pg.db.discovery` returns to Zabbix:

```
{"data": [{"#DBNAME": "postgres"}, {"#DBNAME": "sales"}]}
```

and in Zabbix the prototype `tez.pg.sessions[{#DBNAME}]` is defined, which produces `tez.pg.sessions[postgres]` and `tez.pg.sessions[sales]`.

3.10 Context

A context is a connection state that can be changed between queries without reconnecting. Typical examples are the current database in SQL Server or PostgreSQL and the container (PDB) in Oracle multitenant. Thanks to contexts, a single monitor with a single connection can query every database on a server.

Contexts only exist in the Database engine. They are declared in the XML file with a default value. Each query runs in the default context, in a fixed one or in the one given by the item key. Databbix groups queries by context so as to switch as few times as possible.

Example. The context `Database` is declared, with default value `master`, along with a query that runs in the database given by the first key parameter:

```
<context name="Database" module="MSSQL" default="master"/>

<query name="MSFGSpace" module="MSSQL" select="FILTER">
  <sql> ... </sql>
  <parameter context="Database" order="1"/>
  <parameter column="filegroup_name" order="2" type="STRING"/>
  <item name="tez.mssql.fgspace.used" column="used"/>
</query>
```

The key `tez.mssql.fgspace.used[sales,PRIMARY]` runs the query in the `sales` database and returns the value for the `PRIMARY` filegroup.

3.11 Environment variable

An environment variable is a named value substituted in the XML files with the syntax `${Name}`. It is used to write generic XML files and adapt them to each installation from the configuration: a schema name, the administration database, etc.

They are unrelated to operating system environment variables. They are defined with `<environment>` in the XML file (default value) and with `Environment = Name=Value` in `databbix.conf` or in the monitor file, which take precedence.

Example. The XML file declares the variable and uses it in the SQL:

```
<environment name="SvmSchema" default="svm920"/>

<query name="JdeSpecJobs" module="JDEDB">
  <sql><![CDATA[
    select count(*) as n from ${SvmSchema}.f986110
  ]]>
  </sql>
  <item name="ts.jdedb.specjobs" column="n"/>
</query>
```

A monitor that says nothing queries `svm920.f986110`. Another monitor with this line queries `svm910.f986110`:

```
[General]
Environment = SvmSchema=svm910
```

3.12 Runtime data

Values that are only known once the connection is established, such as the database product the driver is connected to. They are used in the Database engine (section 9.2.4). They are read with the `runtime('NAME')` function in `<when>` conditions, to pick the right SQL for each database.

Example.

```

<sql>
  <multiple>
    <when condition="runtime('DB.DATABASE') = 'Oracle'"><![CDATA[
      select sysdate as now_ts from dual ]]></when>
    <when><![CDATA[
      select current_timestamp as now_ts ]]></when>
  </multiple>
</sql>

```

3.13 Scripts and transformations

XML files can include functions written in JavaScript or Python (<script>). These functions are used to:

- transform a value before sending it to Zabbix (for example, turn a date into elapsed seconds);
- process a web service response before extracting the values;
- generate credentials and validate responses in web service authentication.

Each monitor has an interpreter for its query collection. Scripts declared at <Queries> level are shared by all queries; those declared inside a <query> are only visible in that query. All scripts in a collection must be in the same language. Chapter 12 explains how to write them.

Example. A global function and an item that uses it:

```

<script language="Javascript"><![CDATA[
  function toUnix(date) {
    return Math.round(Date.parse(date) / 1000);
  }
]]>
</script>

<query name="LastBackup" module="DB">
  <sql><![CDATA[
    select max(completion_time) as end_time from v$backup_set ]]>
  </sql>
  <item name="local.ora.lastbackup" column="end_time" transform="Javascript:toUnix" />
</query>

```

3.14 Caching and scheduling

Zabbix decides how often it wants each item (the item's update interval). Databbix groups items by query and, when several items of the same query are due, runs the query only once.

The result of a query is cached for a maximum time (the monitor's **General.MaxCache** or the query's **maxCache** attribute, whichever is smaller). While the result is valid, the items of that query are served from the cache, without running it again.

Example. The monitor allows results to be reused for 60 seconds, but one particular query only for 30:

```

[General]
MaxCache = 60

```

```
<query name="TSSpace" module="DB" select="FILTER" maxCache="30"> ... </query>
```

3.15 Web service authentication

Most web services require authentication before answering. Three pieces are involved in Databbix:

Authentication engine. An authentication mechanism written in C++ inside Databbix: HTTPAuth, RESTToken, JDE and Oasis (table 3.1). Not to be confused with the monitor's query engine (Webservice).

Authentication method. A customisation of an authentication engine for a particular service, defined in a query file with the `<authentication>` element. It has a name (`name` attribute), states the engine it uses (`type` attribute) and provides JavaScript or Python functions that decide, for example, which *login* request is sent, where the token is in the response or when to authenticate again. Only HTTPAuth and RESTToken can be customised.

Credentials. User, password and other data (environment, role...) defined in the monitor file with `UserName`, `Password` and `Credentials = key=value`. This way, XML files contain no secrets.

The monitor's `Webservice.Authenticator` parameter gives a name, which Databbix resolves in this order:

1. It looks in the monitor's query files for an authentication method (`<authentication>`) with that `name`. If found, it uses its engine with the method's functions.
2. Otherwise, it looks for an authentication engine with that name and uses it directly, without customisation.
3. If neither exists, the monitor cannot connect and logs the error.

Engine	Mechanism
HTTPAuth	Standard HTTP authentication. The method is given by the <code>method</code> credential: BASIC, DIGEST, NTLM (with <code>username</code> and <code>password</code>) or HEADER (a fixed header, with <code>header</code> and <code>value</code>). Optionally, a script can generate the credentials.
RESTToken	Token authentication. A <i>login</i> request is made and the token obtained is sent in every call as a header or cookie. It is configured entirely in the XML file, with scripts that build the login request, validate the response and detect when to authenticate again.
JDE	Token for the JD Edwards REST services (<i>Server Manager</i>).
Oasis	<i>WS-Security UsernameToken</i> header for SOAP services.

Table 3.1: Authentication engines

The authentication session (for example, the current token) is kept while the monitor runs and renewed when the service rejects it. If `Webservice.AuthPersistFile` is set, it is also saved to disk to be reused after a restart.

Example 1. The HTTPAuth engine without customisation: since there is no method called HTTPAuth in the XML files, the engine is used directly, and the credentials go in the `.conf`.

```
[Webservice]
BaseUrl = https://api.example.com/
Authenticator = HTTPAuth
UserName = monitor
Password = secret
Credentials = method=BASIC
```

Example 2. The AISAuth authentication method, which customises the RESTToken engine, defined in the XML file and used from the `.conf`:

```
<authentication type="RESTToken" name="AISAuth">
  <script language="Javascript"> ... </script>
  <login type="JsonPost">
    <url>v2/tokenrequest</url>
    <request-function language="Javascript" function="authGenerate"/>
    <accept-function language="Javascript" function="authAccept"/>
  </login>
  <token type="Header" name="jde-AIS-Auth"/>
  <verify-call language="Javascript" function="verifyCall"/>
</authentication>
```

```
[Webservice]
BaseUrl = https://ais.example.com/jderest/
Authenticator = AISAuth
UserName = monitor
Password = secret
Credentials = environment=JPD920
Credentials = role=*ALL
AuthPersistFile = /var/lib/databbix/ais-token.json
```

Chapter 12 explains how to write the functions of an authentication method.

Part II

Common configuration

Chapter 4

Installation and files

4.1 File locations

A standard installation uses these paths:

Path	Contents
<code>/opt/trustserver/sbin/databbix</code>	Daemon executable.
<code>/opt/trustserver/lib64</code>	Trustserver libraries and Oracle client.
<code>/opt/trustserver/modules/databbix</code>	Modules (query engines, languages, authentication engines).
<code>/etc/databbix/databbix.conf</code>	Global daemon configuration.
<code>/etc/databbix/dbs/</code>	One <code>.conf</code> file per monitor.
<code>/etc/databbix/query*.xml</code>	Query definition files.
<code>/etc/databbix/tnsnames.ora</code>	Oracle connection aliases.
<code>/etc/databbix/odbc.ini</code>	ODBC data sources.
<code>/etc/sysconfig/databbix</code>	Environment variables for the service.
<code>/var/log/databbix/</code>	Log files.

4.2 Bundled query files

The distribution includes several ready-to-use sample XML files:

File	Use
query.xml	Oracle (default file).
query-mt.xml	Oracle multitenant (CDB/PDB).
query-ora-9.xml	Oracle 9.
query-pg.xml	PostgreSQL.
query-pg-9.xml, query-pg-10.xml	Queries specific to PostgreSQL versions.
query-pg-bareos.xml	Bareos catalogue on PostgreSQL. Always used together with query-pg.xml*.
query-mssql.xml	Microsoft SQL Server (through ODBC).
query-db-jde.xml	JD Edwards database (Oracle, DB2/400 or SQL Server). If JDEDB is set, it is used together with the file declaring the ContextName context*.
query-jde.xml, query-ais.xml	JD Edwards and AIS REST services.
query-soap.xml	SOAP service example.
query-mqtt.xml	MQTT broker.
query-emcunity.xml, query-sagex3.xml	Other web services.
zabbix_templates.xml	Related Zabbix templates.

***Files that depend on another one.** query-pg-bareos.xml does not declare the Database context it uses: query-pg.xml does. Likewise, query-db-jde.xml uses the context given by the ContextName variable (Container by default, declared in query-mt.xml; Database, in query-pg.xml or query-mssql.xml). Since a context must be declared before it is used, the file that declares it must come *first* in the QueryFile lines:

```
[General]
QueryFile = /etc/databbix/query-pg.xml          # declares the Database context
QueryFile = /etc/databbix/query-pg-bareos.xml   # uses it
```

4.3 Environment variables

The service reads /etc/sysconfig/databbix. These are the usual variables:

```
TNS_ADMIN=/etc/databbix          # tnsnames.ora directory
ODBCINI=/etc/databbix/odbc.ini    # ODBC data source file
```

The systemd unit also adds LD_LIBRARY_PATH=/opt/trustserver/lib64.

Chapter 5

Daemon configuration

5.1 Command line

```
databbix [options]
```

Option	Description
-h, --help	Shows help and exits.
-d, --daemon	Runs in the background (daemon mode). Without this option, the log is also printed to the console.
-n, --no-send-data	Does not send data to the Zabbix server. Useful for testing.
-c, --config-file <i>f</i>	Global configuration file. Default: <code>/etc/databbix/databbix.conf</code> .
-p, --config-path <i>d</i>	Directory with the monitors' <code>.conf</code> files. Default: <code>/etc/databbix/dbs</code> .
-m, --module-path <i>d</i>	Additional directories to search for modules.

To try a new configuration without installing the service, the easiest way is to start the daemon in the foreground, without sending data and with a high log level (see chapter 15):

```
/opt/trustserver/sbin/databbix -n -c /etc/databbix/databbix.conf
```

5.2 The databbix.conf file

An INI file with two sections: `[General]` and `[Log]`. The values of the `[Log]` section and those of `QueryFile` and `Environment` are the defaults for all monitors, and each monitor can override them in its own file.

5.2.1 [General] section

Parameter	Description
DaemonUser	User (name or UID) the process switches to when started with <code>-d</code> . The process also takes that user's supplementary groups.
DaemonGroup	Group (name or GID) the process switches to when started with <code>-d</code> . If not set, the primary group of <code>DaemonUser</code> is used.

Parameter	Description
ConfigPath	Directory of the monitor files. If present, it takes precedence over the <code>-p</code> option.
QueryFile	XML query file. The line can be repeated to load several files. Default: <code>/etc/databbix/query.xml</code> .
ModulePath	Additional module directories. Can be repeated.
Environment	Environment variable for the queries, of the form <code>Name=Value</code> . Can be repeated (see section 7.4).

5.2.2 [Log] section

Parameter	Default	Description
LogPath	<code>/var/log/databbix</code>	Log directory.
LogFile	<code>databbix.log</code>	Log of the main process.
LogLevel	3	Level of detail (see table 15.1).
LogMaxSize	0	Maximum size in bytes before rotating. 0 disables internal rotation.
LogMaxBackupCopies	4	Number of copies kept when rotating.
MaxQueryExecTimeLog	10	If a query takes longer than this many seconds, a warning is logged.

5.2.3 Example

```
[General]
DaemonUser = databbix
DaemonGroup = databbix
QueryFile = /etc/databbix/query.xml

[Log]
LogPath = /var/log/databbix
LogLevel = 4
MaxQueryExecTimeLog = 10
```

Warning. If `DemonUser` is a numeric UID with no entry in `/etc/passwd` and `DemonGroup` is not set, there is no primary group to use: the process keeps the group it was started with, which is `root` if it is launched as `root`. In that case always set `DemonGroup`.

5.3 systemd service

The bundled unit (`databbix.service`) starts the daemon in forking mode as the `databbix` user:

```
[Unit]
Description=Databbix Agent
After=network-online.target
Wants=network-online.target

[Service]
EnvironmentFile=/etc/sysconfig/databbix
Environment="LD_LIBRARY_PATH=/opt/trustserver/lib64"
```

```

PassEnvironment=TNS_ADMIN ODBCINI LD_LIBRARY_PATH
Type=forking
Restart=on-failure
KillMode=process
ExecStart=/opt/trustserver/sbin/databbix -c /etc/databbix/databbix.conf -d
RestartSec=10s
User=databbix
Group=databbix

[Install]
WantedBy=multi-user.target

```

Common operations:

```

systemctl enable --now databbix    # enable and start
systemctl restart databbix         # apply configuration or XML changes
systemctl status databbix

```

Warning. Databbix reads the `.conf` and XML files only at startup. To apply any change to them, restart the service.

5.4 Signals

Signal	Effect
SIGTERM, SIGINT	Orderly shutdown. Waits up to about 4 seconds for the monitors to finish.
SIGHUP	Reopens the log files. Used by logrotate.

The bundled `databbix.logrotate` file sends SIGHUP after rotating:

```

/var/log/databbix/*.log
{
    sharedscripts
    postrotate
        /usr/bin/killall -HUP databbix 2> /dev/null || true
    endscript
}

```

When using logrotate, leave `LogMaxSize = 0` so that logrotate and Databbix do not both rotate the logs.

Chapter 6

Monitor configuration: common part

Each file with the `.conf` extension in the monitor directory (`/etc/databbix/dbs` by default) defines a monitor. Files with any other extension are ignored, so to disable a monitor it is enough to rename its file (for example, to `mydb.conf.off`).

6.1 File structure

A monitor file has two kinds of sections:

- **Common sections**, the same for all engines: `[General]`, `[Zabbix]`, `[Data]` and `[Log]`. They are described in this chapter.
- **One engine section**, with the connection data for the data source. Its name matches the engine chosen in `General.Engine`. It is described in each engine's chapter.

Section	Scope	Contents
<code>[General]</code>	common	Host name, engine, query files, environment variables, retries and cache.
<code>[Zabbix]</code>	common	Zabbix server and passive port.
<code>[Data]</code>	common	Query modules and error tolerance.
<code>[Log]</code>	common	The monitor's own log.
<code>[Database]</code>	Database	Driver, connection string and credentials (chapter 9).
<code>[Webservice]</code>	Webservice	Base URL, authentication and TLS (chapter 10).
<code>[MQTT]</code>	MQTT	Broker, credentials and subscription (chapter 11).

Skeleton of a monitor file:

```
# ---- Common sections -----
[General]
Name = MYHOST
Engine = Database           # Database / Webservice / MQTT

[Zabbix]
Server = zabbix.example.com
Port = 10051

[Data]
Module = ...

[Log]
LogLevel = 4
```

```
# ---- Engine section (only one, the one for General.Engine) -----
[Database]
...
```

6.2 [General] section

Parameter	Default	Description
Name	<i>mandatory</i>	Host name in Zabbix. It must match the host's <i>Host name</i> field in Zabbix <i>exactly</i> .
Engine	Database	Engine: <code>Database</code> , <code>Webservice</code> or <code>MQTT</code> . Determines which connection section is read and which XML files are accepted.
QueryFile	the global one	XML query file(s). If present, it completely replaces the global list. Can be repeated.
Environment	—	<code>Name=Value</code> variables. They take precedence over those in <code>databbix.conf</code> and over the XML defaults. Can be repeated.
RetryOnError	60	Seconds between retries after a connection error with the source or with Zabbix.
MaxCache	10	Maximum lifetime in seconds of a cached result.

6.3 [Zabbix] section

Parameter	Default	Description
Server	<i>mandatory</i>	DNS name or IP of the Zabbix server (or proxy).
Port	<i>mandatory</i>	Server port, usually 10051.
Refresh	300	Seconds between requests for the active item list.
MaxItemsPerPacket	100	Maximum number of values per send.
PassivePort	0	TCP port for passive checks. With 0 no port is opened. If used, it must be different for each monitor.

The passive port answers the items `agent.ping`, `agent.version`, `agent.hostname` and `tez.ora.connstatus` (1 if the connection to the source is up, 0 otherwise). For any other item it returns `ZBX_NOTSUPPORTED`. The passive port is used to check from Zabbix that the monitor is alive.

6.4 [Data] section

Parameter	Default	Description
Module	<i>mandatory</i>	Query module used by this monitor (<code>DB</code> , <code>Instance</code> , <code>PG</code> , <code>JDE</code> ...). Can be repeated.

Parameter	Default	Description
<code>MaxErrors</code>	5	Maximum number of consecutive failed executions. When exceeded, the monitor stops. 0 disables the limit.
<code>ConnLostOnDefault CtxError</code>	yes	Database engine only: if switching to the default context fails, the connection is considered lost and re-established (section 9.5).

Note. In old Database engine configurations, `Module` and `MaxErrors` appear in the `[Database]` section. That form is still valid, but if they exist in `[Data]`, those in `[Data]` win.

6.5 [Log] section

It accepts the same parameters as the `[Log]` section of `databbix.conf` (section 5.2.2). If not given, the global values are inherited. The default log file is `<Name>.log`, inside `LogPath`.

6.6 Special characters

The `#` character starts a comment. If a value needs to contain `#` (for example, an Oracle multitenant common user or the wildcard of an MQTT subscription), write it as the sequence `\043`:

```
# User c##zabbix
UserName = c\043\043zabbix
```

Security. Monitor files contain clear-text passwords. They must belong to `root:databbix` with permissions 0640, as the RPM installs them: the daemon can read them but not modify them. The `databbix` user must not be able to write to its own configuration (`.conf` files, query XML files or their directories), because the XML files may contain code that the daemon runs.

Chapter 7

XML query files: common part

This chapter describes the part of the query file format that is the same for all engines. What depends on each engine (how the operation on the source is written and where each value comes from) is described in chapters 9, 10 and 11.

7.1 What is common and what depends on the engine

An XML query file always has the same structure: a root element `<Queries>` with global declarations and a list of queries. Each query combines common elements with engine-specific ones:

```
<Queries engine="...">                                <!-- common -->

  <environment name="..." default="..." />           <!-- common -->
  <script language="...">...</script>                 <!-- common -->
  <context .../>                                         <!-- Database only -->
  <authentication ...>...</authentication>             <!-- Webservice only -->

  <query name="..." module="..." ...>               <!-- common + engine attributes -->
    <sql>...</sql>                                       <!-- Database only -->
    <url>...</url>                                       <!-- Webservice only -->
    <parameter order="..." .../>                     <!-- common + engine attributes -->
    <item name="..." .../>                             <!-- common + engine attributes -->
    <field name="..." .../>                           <!-- common + engine attributes -->
    <script>...</script>                                <!-- common -->
  </query>

</Queries>
```

Table 7.1 summarises which elements and attributes exist in each engine.

Table 7.1: Elements and attributes per engine

Element / attribute	Database	Webserv.	MQTT	Section
<i><Queries> level</i>				
<Queries engine query databbix>	yes	yes	yes	7.3
<environment>	yes	yes	yes	7.4
<script>	yes	yes	yes	7.5
<context>	yes	—	—	9.5
<authentication>	—	yes	—	10.6

Element / attribute	Database	Webserv.	MQTT	Section
<i><query></i>				
name, module, maxCache, maxExecTime	yes	yes	yes	7.6
discovery="yes"	yes	yes	yes	7.9
type="Constant"	yes	yes	yes	7.10
select="FILTER BIND"	yes	—	—	9.4
type="REST SOAP", transport, encode	—	yes	—	10.3
<sql>, <default>	yes	—	—	9.3
<url>, <base>	—	yes	—	10.3
<transform>				
<method>, <argument>	—	yes (SOAP)	—	10.4
<i><parameter></i>				
order, value, type	yes	yes	yes	7.8
context	yes	—	—	9.5
column, bind	yes	—	—	9.4
path, urlparameter	—	yes	—	10.3
<i><item></i>				
name, transform, default, built_in	yes	yes	yes*	7.7
value	constant queries only			7.10
column	yes	—	—	9.4
path, type	—	yes	—	10.3
<i><field></i> (discoveries)				
name	yes	yes	yes	7.9
column, context	yes	—	—	9.4.4
path	—	yes	—	10.3.4

* In MQTT, item names are fixed (chapter 11).

7.2 General rules

- Element and attribute names are case-insensitive (<Queries> and <queries> are equivalent).
- An unknown attribute or element, or one that does not belong to the engine, is an error: the monitor does not start and the error appears in its log.
- It is advisable to write query and script text inside a <![CDATA[...]]> section so that <, > and & need not be escaped.
- If a monitor loads several files (several QueryFile lines), all of them must belong to the same engine. Query and item names must be unique across all the files.

7.3 The <Queries> element

It is the root element and accepts these attributes:

Attribute	Description
engine	Engine: Database (default), Webservice or MQTT. Must match the monitor's <code>General.Engine</code> .
query	Minimum query format version required by the file. The current version is 2.5.0.
databbix	Minimum Databbix version required by the file.

If the installed version is lower than the required one, the monitor does not start and says so in the log.

7.4 Environment variables: <environment>

Environment variables make it possible to parametrise an XML file and reuse it in several installations: schema names, databases, contexts...

```
<environment name="SvmSchema" default="svm920"/>
```

They are used with the syntax `${Name}`. Where they are substituted depends on the engine: in Database, in the SQL text, in the attributes of contexts, `<default>` and `<parameter>`, and in `<when>` conditions with `environment('Name')`; in Webservice, in the text of the `<header>` elements of HTTPAuth authentication methods.

```
<sql><![CDATA[
  select count(*) as n from ${SvmSchema}.f986110
]]>
</sql>
```

The value of a variable is taken from the first of these sources that defines it:

1. Environment in the monitor's `.conf` file.
2. Environment in `databbix.conf`.
3. The `default` attribute of the `<environment>` in the XML file.

For example, to make a monitor use the `svm910` schema:

```
[General]
Environment = SvmSchema=svm910
```

An undefined variable is replaced by an empty string.

7.5 Scripts: <script>

A `<script>` contains JavaScript or Python functions used to transform values or responses. It can be placed at `<Queries>` level, and then it is available to all queries, or inside a `<query>`, and then it is only available to that query.

```
<script language="Javascript"><![CDATA[
  function toUnix(date) {
    return Math.round(Date.parse(date) / 1000);
  }
]]>
</script>
```

All scripts in a query collection must use the same language. Chapter 12 explains how to write them.

7.6 The query: <query>

7.6.1 Common attributes

Attribute	Description
<code>name</code>	<i>Mandatory.</i> Unique query name. It appears in the log and in <code>queryexplorer</code> .
<code>module</code>	<i>Mandatory.</i> Module the query belongs to.
<code>discovery</code>	<code>yes</code> for a discovery (LLD) query.
<code>type</code>	Constant for a constant-value query. In the Webservice engine, also REST or SOAP.
<code>maxCache</code>	Maximum cache lifetime of this query's results, in seconds. The effective lifetime is the smaller of this value and the monitor's <code>MaxCache</code> .
<code>maxExecTime</code>	Threshold in seconds above which a slow execution warning is logged. If not given, <code>MaxQueryExecTimeLog</code> is used.

7.6.2 Query types per engine

Engine	Type	What it does
All	Constant	Returns fixed values from the XML file.
	Discovery	Produces the JSON of a Zabbix LLD rule.
Database	Basic	An SQL statement that returns one row.
	Filter	A statement that returns many rows; the key selects the row.
	Bind	The key is passed to the statement as a variable.
Webservice	REST	HTTP call with a JSON response; the key can be sent in the URL or used as a filter.
	SOAP	SOAP call with fixed arguments.
MQTT	Value	Last value published on a <i>topic</i> .

7.7 The <item> element

Each <item> declares a Zabbix item produced by the query. These are its common attributes; each engine adds the attributes that say where the value comes from (`column` in Database, `path` and `type` in Webservice).

Attribute	Description
<code>name</code>	<i>Mandatory.</i> Item key in Zabbix, without parameters. It must be unique across all loaded queries.
<code>transform</code>	Function that transforms the value before sending it, of the form <code>Language:function</code> (for example, <code>Javascript:toUnix</code>). See chapter 12.
<code>default</code>	Value sent if the requested data does not exist (the row of a filter query or the path of a JSON response).
<code>built_in</code>	Internal item of the query: <code>exec_time</code> (execution time in seconds) or <code>exec_success</code> (1 if the last execution succeeded, 0 otherwise).

The same piece of data can feed several items with different names, for example to keep an old key and a new one at the same time:

```
<item name="tez.ora.queryversion" column="version"/>
<item name="ts.db.queryversion" column="version"/>
```

7.8 The <parameter> element

Links each parameter of the Zabbix key to the query. These are the common attributes; the attribute that says *what for* the parameter is used depends on the engine (`column`, `bind` or `context` in Database; `path` or `urlparameter` in Webservice).

Attribute	Description
<code>order</code>	Position of the parameter in the item key, starting at 1.
<code>value</code>	Fixed value, instead of taking it from the key. Incompatible with <code>order</code> .
<code>type</code>	STRING (default) or INT. With INT, the value is compared or sent as a number.

Key parameters are separated by commas only and leading and trailing spaces are removed. Quotes are not interpreted, so a parameter cannot contain commas.

7.9 Discovery queries

In every engine, a query with `discovery="yes"` produces the JSON of a Zabbix low-level discovery rule:

```
{"data": [ {"#MACRO1": "value", "#MACRO2": "value", ... } ]}
```

- Each discovered entity (a row, an element of a JSON array, a *topic*) is an element of **data**.
- Each `<field>` defines a macro: the **name** attribute is the macro name without `{#}` and `}`, and the engine-specific attribute says where its value comes from.
- The `<item>` elements of a discovery only have **name** and, optionally, **transform**; in that case the function receives the whole JSON (section 12.5).

7.10 Constant queries

With `type="Constant"`, the query does not access the data source: each `<item>` returns the value of its `value` attribute. They work with any engine and are typically used to publish the version of the query file:

```
<query name="JDEVersion" module="JDE" type="Constant">
  <item name="ts.jde.queryversion" value="$Id: manual-databbix-en.tex 2674
    2026-09-30 20:30:56Z bruno $" />
</query>
```


Chapter 8

Zabbix configuration

For Databbix to collect data for a host, do the following in Zabbix:

1. Create the host with its *Host name* equal to the monitor's `General.Name`. Case matters.
2. Create the items (or import a template, for example those in `zabbix_templates.xml`) of type **Zabbix agent (active)**. The key of each item must match the `name` attribute of an `<item>` in the XML files.
3. If `PassivePort` is configured, add an agent interface with that port to be able to use passive items such as `agent.ping`.
4. Allow the Zabbix server or proxy to accept connections from the Databbix machine on port 10051.

8.1 Item keys

The key entered in Zabbix has the form:

```
item.name[parameter1,parameter2,...]
```

- `item.name` selects the `<item>` in the XML file and, with it, the query that is run.
- Parameters are assigned by position to the query's `<parameter>` elements according to their `order` attribute (the first one is `order="1"`).
- Parameters are separated by commas only and leading and trailing spaces are removed. **Quotes are not interpreted**, so a parameter cannot contain commas.

Examples:

<code>tez.ora.sessions</code>	query without parameters
<code>tez.ora.tspace.pctfree[USERS]</code>	FILTER query by tablespace
<code>tez.pg.param[max_connections]</code>	BIND query
<code>tez.mssql.fgspace.used[sales,PRIMARY]</code>	context (database) + filter
<code>tez.ora.tspace.discovery</code>	discovery rule

8.2 Internal items

The following keys need no query in the XML files:

Key	Value
<code>agent.ping</code>	Always 1.
<code>agent.version</code>	Databbix version, for example <code>databbix-2.5.0-2639</code> .
<code>agent.hostname</code>	Host name (<code>General.Name</code>).
<code>tez.ora.ping</code>	Status code of the connection to the data source (0 if there is no connection).

Any other key not defined in the XML files is returned to Zabbix as *not supported*.

Part III

Engines

Chapter 9

Database engine

The **Database** engine obtains data by running SQL statements against a database. It is the default engine: a monitor without **General.Engine** uses it.

9.1 Configuration: the [Database] section

Parameter	Description
DBDriver	<i>Mandatory.</i> Driver: Oracle , PGsql or ODBC .
ConnectionString	<i>Mandatory.</i> Connection string. Its format depends on the driver (section 9.2).
UserName	<i>Mandatory.</i> Database user.
Password	<i>Mandatory.</i> Password.
ConnectOptions	Special options. The only one supported is SYSDBA (required for ASM).

The database user only needs read permission on the views queried by the XML files.

9.2 Drivers

The **DBDriver** parameter selects the driver the monitor connects with. Each driver interprets the connection string in its own way, supports specific contexts and offers its own runtime data.

DBDriver	Databases	Contexts	Connection string
Oracle	Oracle (OCI)	Container	tnsnames.ora alias, <i>Easy Connect</i> or full descriptor.
PGsql	PostgreSQL (libpq)	Database	libpq key=value pairs.
ODBC	Any with an ODBC driver (SQL Server, DB2/400...)	Database	SQLDriverConnect string: DSN=name or DRIVER={...};...

Note. PostgreSQL and Oracle can also be accessed through the ODBC driver, using an ODBC driver from the vendor or a third party. Even so, the native driver (**PGsql** or **Oracle**) is recommended whenever possible: it is the one that knows each database's contexts (**Database** in PostgreSQL, **Container** in Oracle) and provides its specific runtime data, such as **ORACLE.CONTAINER**.

In all drivers, the bind variables of BIND queries are always written as `:name`. The driver converts them to each database's syntax (`$1`, `$2...` in PostgreSQL, `?` in ODBC).

9.2.1 Oracle driver

Connection string. Passed as is to the Oracle client, so it accepts any form the client accepts:

```
ConnectionString = oraprod                                # tnsnames.ora alias
ConnectionString = dbhost.example.com:1521/ORAPROD      # Easy Connect
ConnectionString = (DESCRIPTION=(ADDRESS=(PROTOCOL=TCP)(HOST=dbhost)(PORT=1521)
    )(CONNECT_DATA=(SERVICE_NAME=ORAPROD)))
```

To use aliases, the service's `TNS_ADMIN` variable must point to the `tnsnames.ora` directory.

ConnectOptions. `SYSDBA` connects with administrator privileges (required for ASM). Empty for a normal connection.

Contexts. **Container:** the current container (PDB), changed with `ALTER SESSION SET CONTAINER`. It is only available on Oracle 12c or later, on a multitenant database and when connected to the root container (`CDB$ROOT`). In any other case the driver supports no contexts and `ORACLE.CONTAINER` is 0.

Bind variables. Native Oracle (`:name`).

9.2.2 PGsql driver

Connection string. `libpq` `key=value` pairs separated by spaces (`host`, `port`, `dbname`, `sslmode`, `connect_timeout...`). The user and password are added from `UserName` and `Password`; do not put them in the string.

```
ConnectionString = host=pg01.example.com port=5432 dbname=postgres sslmode=
    require
```

ConnectOptions. Not used.

Contexts. **Database:** the current database. Since PostgreSQL cannot switch databases within a connection, the driver opens one connection per database used (replacing the string's `dbname`) and keeps them open.

Bind variables. Converted to `$1`, `$2...`

Warning. Always include `dbname` in the connection string: the driver uses it as the default database and replaces it when switching context. The password is appended to the string without quotes, so it cannot contain spaces.

9.2.3 ODBC driver

Connection string. Passed to `SQLDriverConnect`, appending `;UID=...;PWD=...` with the user and password. It can refer to a data source in `odbc.ini` or describe the full connection:

```
ConnectionString = DSN=MSSQL_PROD
ConnectionString = DRIVER={FreeTDS};SERVER=mssql.example.com;PORT=1433;DATABASE
    =master
```

To use DSN, the service's `ODBCINI` variable must point to the `odbc.ini` file.

ConnectOptions. Not used.

Contexts. Database: the current database (catalogue), changed within the same connection with the ODBC attribute `SQL_ATTR_CURRENT_CATALOG`. It works with drivers that support it, such as the SQL Server ones.

Bind variables. Converted to `?`.

Note. The password is appended to the string without quotes, so it cannot contain the `;` character.

9.2.4 Runtime data

Runtime data is read in `<when>` conditions with `runtime('NAME')` (section 9.6) and in `queryexplorer` with `getRuntimeValue`.

Name	Driver	Value
<code>DATABBIX.PROGRAM_NAME</code>	all	databbix, or <code>queryexplorer</code> in that utility.
<code>CONN.TYPE</code>	all	Database.
<code>DB.DRIVER</code>	Oracle	Oracle.
	PGsql	PostgreSQL.
	ODBC	ODBC.
<code>DB.DATABASE</code>	Oracle	Oracle.
	PGsql	PostgreSQL.
	ODBC	Product name reported by the ODBC driver (<code>SQL_DBMS_NAME</code>): for example, Microsoft SQL Server or DB2/400 SQL.
<code>DB.VERSION</code>	Oracle	Server version, in the form <code>major.minor.update</code> .
	PGsql	The server's <code>server_version</code> setting, for example 15.4.
	ODBC	Version reported by the ODBC driver (<code>SQL_DBMS_VER</code>).
<code>ORACLE.CONTAINER</code>	Oracle	1 if the <code>Container</code> context is available; 0 otherwise.

This way, the same XML file can choose the SQL by product (`runtime('DB.DATABASE') = 'Microsoft SQL Server'`), by version (`major(runtime('DB.VERSION')) >= 12`) or by whether the database is multitenant (`runtime('ORACLE.CONTAINER') = '1'`).

9.3 SQL queries: engine-specific elements

Besides the common elements (chapter 7), Database engine queries use:

Element / attribute	Description
<code><sql></code>	Statement text. Mandatory except in constant queries. Accepts <code>\${...}</code> variables and alternatives with <code><multiple></code> (section 9.6).
<code><sql context="C"></code>	Discoveries only: statement returning the values of context <code>C</code> to be iterated (section 9.4.4).

Element / attribute	Description
<code>select</code>	<code><query></code> attribute: <code>FILTER</code> or <code>BIND</code> . Without it, the query is basic.
<code><item column="..."></code>	Result column the value comes from. Case-insensitive.
<code><parameter column></code>	In <code>FILTER</code> queries: column compared with the parameter.
<code><parameter bind></code>	In <code>BIND</code> queries: SQL bind variable.
<code><parameter context></code>	Context the query runs in.
<code><default></code>	Fixed context of the query (section 9.5).
<code><field column context></code>	In discoveries: where the macro value comes from.
<code><context></code>	At <code><Queries></code> level: declaration of a context.

Note. Always give computed columns an explicit alias (`as name`) and use that alias in `column`.

9.4 Query types

The examples in this section show, for each type, the result of the SQL statement at a given moment and the value each Zabbix key obtains from it.

9.4.1 Basic query

The statement returns **exactly one row**. Each `<item>` takes the value of one column.

```
<query name="PGWriter" module="PG">
  <sql><![CDATA[
    select checkpoints_timed, checkpoints_req from pg_stat_bgwriter
  ]]>
</sql>
<item name="tez.pg.checkpointtimed" column="checkpoints_timed"/>
<item name="tez.pg.checkpointreq" column="checkpoints_req"/>
</query>
```

Statement result

checkpoints_timed	checkpoints_req
1842	37

Values obtained

Key	Value
tez.pg.checkpointtimed	1842
tez.pg.checkpointreq	37

If the statement returns zero rows or more than one, all the query's items become *not supported*.

9.4.2 Filter query (select="FILTER")

The statement returns **many rows**. Each `<parameter>` with `column` links a key parameter with a column, and each item takes the row whose columns match its key parameters. The statement runs *only once* for all items, so this is the most efficient way to watch collections of objects (tablespaces, databases, jobs...).

```
<query name="PGSessions" module="PG" select="FILTER">
  <sql><![CDATA[
    select datname,
           count(st.pid) as n,
           sum(case st.state when 'active' then 1 else 0 end) as active
    from pg_database db
         left outer join pg_stat_activity st using (datname)
    where datname not like 'template%'
    group by datname
  ]]>
</sql>
<parameter column="datname" order="1" type="STRING"/>
<item name="tez.pg.sessions"      column="n"/>
<item name="tez.pg.sessionsactive" column="active" default="0"/>
</query>
```

Statement result

datname	n	active
postgres	3	1
sales	12	4
hr	0	0

Values obtained

Key	Selected row	Value
tez.pg.sessions[sales]	sales	12
tez.pg.sessionsactive[sales]	sales	4
tez.pg.sessions[hr]	hr	0
tez.pg.sessions[purchasing]	none	<i>not supported</i>
tez.pg.sessionsactive[purchasing]	none	0 (default attribute)
tez.pg.sessions[Sales]	none	<i>not supported</i>

The first four keys are resolved with a single execution of the statement. The comparison is exact: it is case-sensitive (**S**ales does not match **s**ales) and only trims spaces at the ends of the parameter. If a column is space-padded (char type), trim it in the SQL with `trim()`.

Several filter columns

With several filter parameters, the key has one parameter for each, in the order given by `order`, and the row must match all of them:

```
<query name="MSStateRepl" module="MSSQL" select="FILTER">
  <sql><![CDATA[ exec sp_replmonitorhelppublication ]]></sql>
  <default context="Database" value="distribution"/>
  <parameter column="publisher_db" order="1" type="STRING"/>
  <parameter column="publication" order="2" type="STRING"/>
  <item name="tez.mssql.repl.status" column="status"/>
</query>
```

Statement result

publisher_db	publication	status
sales	pub_orders	3
sales	pub_customers	2
purchasing	pub_orders	6

Values obtained

Key	Value
tez.mssql.repl.status[sales, pub_orders]	3
tez.mssql.repl.status[purchasing, pub_orders]	6
tez.mssql.repl.status[sales]	<i>not supported</i> (missing parameter)

Numeric parameters

With `type="INT"`, the parameter and the column are compared as numbers. This way, the key does not have to reproduce the exact format of the column:

```
<parameter column="job_id" order="1" type="INT"/>
<item name="local.job.status" column="status"/>
```

Statement result and values obtained

job_id	status	Key	Value
7	OK	local.job.status[7]	OK
42	ERROR	local.job.status[042]	ERROR

With `type="STRING"`, the key `local.job.status[042]` would not find the row, because the text 042 differs from 42.

9.4.3 Bind variable query (`select="BIND"`)

The key parameter is passed to the statement as a bind variable. The statement runs *once for each distinct key* and must return exactly one row.

```
<query name="PGParameter" module="PG" select="BIND">
  <sql><![CDATA[
    select setting from pg_settings where name = :name
  ]]>
</sql>
<parameter bind=":name" order="1" type="STRING"/>
<item name="tez.pg.param" column="setting"/>
</query>
```

Executions and values obtained

Key	Value of :name	Value
tez.pg.param[max_connections]	'max_connections'	200
tez.pg.param[work_mem]	'work_mem'	4096
tez.pg.param[no_such]	'no_such'	<i>not supported</i> (0 rows)

The variable is always written as `:name`, whatever the driver (section 9.2). With `type="INT"`, the value is bound as a number.

Warning. Since it runs once per key, a BIND query is much more expensive than a FILTER query when there are many keys. Use it only when the set of possible values is very large and only a few are queried.

9.4.4 Discovery query

Each result row is a discovered entity, and each `<field>` takes the value of a column (`column`) or of a context (`context`).

```
<query name="TSDiscover" module="DB" discovery="yes">
  <sql><![CDATA[
    select tablespace_name, contents
    from dba_tablespaces where status = 'ONLINE'
  ]]>
</sql>
<item name="tez.ora.tspace.discovery"/>
<field name="TSNAME" column="tablespace_name"/>
<field name="CONTENTS" column="contents"/>
</query>
```

Statement result

tablespace_name	contents
SYSTEM	PERMANENT
USERS	PERMANENT
TEMP	TEMPORARY

Value of the item `tez.ora.tspace.discovery`:

```
{"data": [
  {"{#TSNAME}": "SYSTEM", "{#CONTENTS}": "PERMANENT"},
  {"{#TSNAME}": "USERS", "{#CONTENTS}": "PERMANENT"},
  {"{#TSNAME}": "TEMP", "{#CONTENTS}": "TEMPORARY"}
]}
```

In Zabbix, the prototype `tez.ora.tspace.pctfree[{#TSNAME}]` creates the items `tez.ora.tspace.pctfree[SYSTEM]`, `[USERS]` and `[TEMP]`, resolved by a FILTER query with parameter `column="tablespace_name"`.

Discoveries only accept context parameters.

Discovery across contexts

A discovery can iterate over all the values of a context: for example, list the filegroups of *all* the databases of an SQL Server. To do so, add an `<sql>` with a `context` attribute that returns, in its first column, the values to iterate. The main statement runs once in each of them:

```
<query name="MSDiscoverFG" module="MSSQL" discovery="yes">
  <sql context="Database"><![CDATA[
    select name from sys.databases where name not in ('model')
  ]]>
</sql>
<sql><![CDATA[
  select name from sys.filegroups
```

```

]]>
</sql>
<item name="tez.mssql.fg.discovery"/>
<field name="DBNAME" context="Database"/>
<field name="FGNAME" column="name"/>
</query>

```

Statement results

Context statement name	Main statement in each database	
	database	name
master	master	PRIMARY
sales	sales	PRIMARY
	sales	HISTORY

Value of the item `tez.mssql.fg.discovery`:

```

{"data": [
  {"#DBNAME": "master", "#FGNAME": "PRIMARY"},
  {"#DBNAME": "sales", "#FGNAME": "PRIMARY"},
  {"#DBNAME": "sales", "#FGNAME": "HISTORY"}
]}

```

Context values that cannot be entered (for example, an offline database) are skipped and logged.

9.5 Contexts

A context is a connection state that can be changed between queries, such as the current database in SQL Server or PostgreSQL or the container (PDB) in Oracle multitenant. This way, a single monitor with a single connection can query several databases on the same server. Contexts only exist in the Database engine.

9.5.1 Declaration

Contexts are declared at `<Queries>` level:

```

<environment name="PGAdminDB" default="postgres"/>
<context name="Database" module="PG" default="${PGAdminDB}"/>

```

Attribute	Description
name	<i>Mandatory.</i> Context name. The driver must know how to change it; the usual names are <code>Database</code> (SQL Server, PostgreSQL) and <code>Container</code> (Oracle multitenant).
module	<i>Mandatory.</i> Module it belongs to.
default	<i>Mandatory.</i> Default value. Accepts <code>\${...}</code> variables.

9.5.2 Use in queries

There are three ways to choose the context of a query:

1. **Default.** If the query says nothing, it runs with the context's `default` value.

2. Fixed for the query, with <default>:

```
<default context="Database" value="msdb"/>
```

If value is `__IGNORE__` (for example, through an unset environment variable), the element is ignored and the default value is kept.

3. Taken from the item key, with a context parameter:

```
<parameter context="Database" order="1"/>
```

9.5.3 Example: a filter inside each database

With a context parameter, the first key parameter selects the database and the following ones select the row:

```
<context name="Database" module="MSSQL" default="master"/>

<query name="MSFGSpace" module="MSSQL" select="FILTER">
  <sql><![CDATA[
    select g.name as filegroup_name,
           sum(cast(fileproperty(f.name, 'SPACEUSED') as bigint) * 8192) as used
    from sys.database_files f
           inner join sys.filegroups g on g.data_space_id = f.data_space_id
    group by g.name
  ]]>
</sql>
<parameter context="Database" order="1"/>
<parameter column="filegroup_name" order="2" type="STRING"/>
<item name="tez.mssql.fgspace.used" column="used"/>
</query>
```

Statement result in each database

database	filegroup_name	used
master	PRIMARY	5242880
sales	PRIMARY	73400320
sales	HISTORY	524288000

Values obtained

Key	Context	Value
tez.mssql.fgspace.used[sales,HISTORY]	sales	524288000
tez.mssql.fgspace.used[sales,PRIMARY]	sales	73400320
tez.mssql.fgspace.used[master,PRIMARY]	master	5242880

Databbix groups executions by context: it enters `sales` only once, runs the statement and resolves the first two keys with the same result. If it cannot enter a context, it marks the items of that context as *not supported*. If the failing context is the *default* one and `ConnLostOnDefaultCtxError = yes`, the connection is considered lost and re-established.

9.6 Alternative SQL depending on the database

The same XML file can serve different database products or versions if the SQL text is chosen at run time, with `<multiple>`:

```
<sql>
  <multiple>
    <when condition="runtime('DB.DATABASE') = 'Oracle'"><![CDATA[
      select count(*) as n from ${SvmSchema}.f986110
      where trim(jcfndfuf2) = :jobname
    ]]>
    </when>
    <when condition="runtime('DB.DATABASE') = 'Microsoft SQL Server'"><![CDATA[
      select count(*) as n from ${SvmSchema}.f986110
      where rtrim(jcfndfuf2) = :jobname
    ]]>
    </when>
    <when><![CDATA[
      -- no condition: used if none of the above applies
      select count(*) as n from ${SvmSchema}.f986110
      where jcfndfuf2 = :jobname
    ]]>
    </when>
  </multiple>
</sql>
```

The first `<when>` whose condition holds is used. A `<when>` without a condition always holds, so it must be the last one. If no condition holds, the query fails.

Conditions are expressions that can use these functions:

Function	Description
<code>runtime('X')</code>	Runtime data of the active connection, for example <code>DB.DATABASE</code> or <code>DB.VERSION</code> (section 9.2.4).
<code>environment('X')</code>	Value of an environment variable.
<code>major('X.Y.Z')</code>	First number of a version, as a number.
<code>toscalar('X')</code>	Converts a text into a number.

They accept comparison operators (`=`, `!=`, `<`, `>`...) and logical operators (`and`, `or`, `not`). The runtime data available with each driver can be listed with the `getRuntimeNames` command of `queryexplorer`.

9.7 Configuration examples

9.7.1 Example: Oracle

```
[General]
Name = ORAPROD

[Zabbix]
Server = zabbix.example.com
Port = 10051
PassivePort = 10153

[Data]
Module = DB
```

```
Module = Instance

[Database]
DBDriver = Oracle
ConnectionString = oraprod          # tnsnames.ora alias
UserName = zabbix
Password = secret
```

9.7.2 Example: Oracle multitenant

```
[General]
Name = ORACDB
QueryFile = /etc/databbix/query-mt.xml

[Zabbix]
Server = zabbix.example.com
Port = 10051

[Data]
Module = MT

[Database]
DBDriver = Oracle
ConnectionString = oracdb
UserName = c\043\043zabbix        # c##zabbix
Password = secret
```

9.7.3 Example: ASM

```
[General]
Name = asm

[Zabbix]
Server = zabbix.example.com
Port = 10051

[Data]
Module = ASM

[Database]
DBDriver = Oracle
ConnectionString = asm
ConnectOptions = SYSDBA
UserName = sys
Password = secret
```

9.7.4 Example: PostgreSQL

```
[General]
Name = PGPROD
QueryFile = /etc/databbix/query-pg.xml
QueryFile = /etc/databbix/query-pg-10.xml
Environment = PGAdminDB=postgres
```

```
[Zabbix]
Server = zabbix.example.com
Port = 10051

[Data]
Module = PG

[Database]
DBDriver = PGsql
ConnectionString = host=pg01.example.com port=5432 dbname=postgres
UserName = zabbix
Password = secret
```

9.7.5 Example: SQL Server through ODBC

First, define the data source in `/etc/databbix/odbc.ini`:

```
[MSSQL_PROD]
Driver=FreeTDS
Description=Production SQL Server
Server=mssql.example.com
Port=1433
Database=master
```

Then the monitor refers to it by name:

```
[General]
Name = MSSQLPROD
QueryFile = /etc/databbix/query-mssql.xml

[Zabbix]
Server = zabbix.example.com
Port = 10051

[Data]
Module = MSSQL

[Database]
DBDriver = ODBC
ConnectionString = DSN=MSSQL_PROD
UserName = zabbix
Password = secret
```

Chapter 10

Webservice engine

The **Webservice** engine obtains data by calling web services over HTTP. It supports two kinds of service, REST and SOAP, which can coexist in the same query file and the same monitor.

10.1 REST and SOAP

Both kinds share the connection (base URL, TLS, authentication) and the way values are extracted: in both cases the response is turned into a JSON document and each item gives a *path* inside it. They differ in how the request is built and in the original format of the response:

	REST (type="REST")	SOAP (type="SOAP")
Request	GET or POST to a URL.	POST of a SOAP XML envelope with a method and its arguments.
Key parameters	Can be sent in the URL (<code>urlparameter</code>) or used to choose an element of the response (<code>path</code> with <code><base></code>).	Not used: the method arguments are fixed in the XML file.
Response	JSON.	SOAP XML, which Databbix converts into JSON (section 10.4.1).
<code>status_code</code> in <code><transform></code>	HTTP status code of the response.	Always 0; errors arrive as a <i>Fault</i> in the JSON itself.
Discoveries	Yes.	No.
Authentication	HTTPAuth, RESTToken, JDE.	Oasis (WS-Security) or HTTPAuth.

10.2 Configuration: the [Webservice] section

Parameter	Description
BaseURL	<i>Mandatory.</i> Base URL. Query URLs are relative to it.
UserName, Password	Credentials.
Credentials	Additional credential of the form <code>key=value</code> (environment, role, method...), depending on the authentication engine. Can be repeated.

Parameter	Description
Authenticator	Name of an authentication method defined in the XML files or, if none exists, of an authentication engine: <code>HTTPAuth</code> , <code>JDE</code> or <code>Oasis</code> (section 10.6).
Timeout	Maximum wait time in milliseconds (default: 20000).
AuthPersistFile	File where the authentication state (for example, the token) is saved between restarts.
VerifyCertificate	Verify the server certificate (default: <code>yes</code>).
CertCAFile	CA file used instead of the system certificate store.
ClientCertificate	Client certificate (PEM) for TLS authentication.
ClientPrivKey	Private key of the client certificate. Mandatory if <code>ClientCertificate</code> is used.
ClientPrivKeyPass	Password of the private key, if encrypted.
DisableContentEncoding	Disables <i>Content-Encoding</i> , for old servers (default: <code>no</code>).

```
[General]
Name = JDEPROD
Engine = Webservice
QueryFile = /etc/databbix/query-jde.xml

[Zabbix]
Server = zabbix.example.com
Port = 10051

[Data]
Module = JDE

[Webservice]
BaseUrl = http://10.1.1.45:9888/restservice
UserName = monitor
Password = secret
Authenticator = JDE
```

10.3 REST queries

```
<Queries engine="Webservice">

<query name="JDEEntStatus" module="JDE" type="REST" transport="GET" encode="Json">
  <url>entserverinstanceinfo</url>
  <parameter urlparameter="instanceName" order="1" type="STRING"/>
  <item name="ts.jde.enterprise.kerneljobs"
    path="entServerInstanceInfoDetail:[0]:kernelJobs" type="INT"/>
</query>

</Queries>
```

Element / attribute	Description
<code>type="REST"</code>	<code><query></code> attribute. Mandatory.
<code>transport</code>	Mandatory. GET or POST.
<code>encode</code>	Response format. Only <code>Json</code> is supported.

Element / attribute	Description
<url>	<i>Mandatory.</i> Path relative to BaseURL .
<parameter urlparameter>	Sends the key parameter in the URL, as ?name=value (section 10.3.2).
<base path>	Array of the response that is iterated to find the element given by the key (section 10.3.3).
<parameter path>	With <base>: field of each element compared with the key parameter.
<item path type>	Path of the value inside the JSON (section 10.3.1) and type: INT or STRING . Both mandatory, except in built_in items.
<transform>	Function that processes the response before the values are extracted (section 12.6).

Without <transform>, a response with an HTTP status other than 200 is an error and all the query's items become not supported.

10.3.1 JSON paths

A path says how to reach a value from the root of the JSON document. It is made of steps separated by colons (:):

Step	Meaning
name	Member name of an object.
[n]	Element n of an array, starting at 0.
[]	All the elements of an array. Only in <base> and in the <item> of a discovery.
*	All the members of an object. Only in <base> and in the <item> of a discovery.

An item's path must end at a simple value (text, number, boolean or null). Given this document:

```
{
  "server": { "name": "ent01", "state": "RUNNING", "up": true },
  "jobs": [
    { "id": 1, "name": "backup", "status": "OK", "duration": 120 },
    { "id": 2, "name": "purge", "status": "FAILED", "duration": 30 }
  ],
  "counters": { "cpu": 12.5, "memory": 2048 },
  "note": null
}
```

Paths and values obtained

path	type	Value
server:name	STRING	ent01
server:up	INT	1
server:up	STRING	true
jobs:[1]:status	STRING	FAILED
jobs:[0]:duration	INT	120
counters:cpu	STRING	12.5
note	STRING	(empty string)
server:host	STRING	<i>not supported:</i> the member does not exist
jobs:[5]:status	STRING	<i>not supported:</i> the array has 2 elements
server	STRING	<i>error:</i> not a simple value

In the *not supported* cases, if the item has a `default` attribute, that value is sent. With `type="INT"`, booleans are turned into 1 and 0.

Note. Member names cannot contain colons, because the colon is the step separator. If the service uses them, restructure the response with a transformation (section 12.6).

10.3.2 Parameters in the URL

Each `<parameter>` with `urlparameter` adds to the URL a parameter with that name and the key value. The query runs once for each distinct key:

```
<query name="JDEInstanceState" module="JDE" type="REST" transport="GET" encode="Json"
  ">
  <url>instancestate</url>
  <parameter urlparameter="instanceName" order="1" type="STRING"/>
  <item name="ts.jde.instance.state" path="instanceState" type="STRING"/>
</query>
```

Calls and values obtained

Key	URL called	Value
ts.jde.instance.state[ENT1]	.../instancestate?instanceName=ENT1	RUNNING
ts.jde.instance.state[WEB1]	.../instancestate?instanceName=WEB1	STOPPED

On each call the service answers, for example, `{"instanceState":"RUNNING"}`, and the path `instanceState` extracts the value.

10.3.3 Filtering the response

When the service returns the list of all objects at once, it is more efficient to make a single call and pick the element in the response, just like a Database FILTER query. To do so:

- `<base path="...">` gives the array to iterate (its path ends in `[]`);
- each `<parameter path="...">` gives a field of each element that is compared with a key parameter;
- item paths are relative to the selected element.

```
<query name="LocalJobs" module="API" type="REST" transport="GET" encode="Json">
  <url>status</url>
  <base path="jobs:[]" />
  <parameter path="name" order="1" type="STRING"/>
  <item name="local.api.job.status" path="status" type="STRING" default="
UNKNOWN"/>
  <item name="local.api.job.duration" path="duration" type="INT"/>
</query>
```

With the document of the previous section as the response:

Values obtained

Key	Selected element	Value
local.api.job.status[backup]	jobs[0]	OK
local.api.job.duration[backup]	jobs[0]	120
local.api.job.status[purge]	jobs[1]	FAILED
local.api.job.status[reindex]	none	UNKNOWN (default)
local.api.job.duration[reindex]	none	<i>not supported</i>

The five keys are resolved with a single call to the service. With `type="INT"` in the `<parameter>`, the comparison is numeric (for example, `parameter path="id"` and the key `...[2]`).

10.3.4 REST discoveries

The path of a discovery's `<item>` iterates over an array (`[]`) or over the members of an object (`*`). Each element iterated is a discovered entity, and the `<field>` paths are relative to it:

```
<query name="LocalJobDiscovery" module="API" type="REST"
  transport="GET" encode="Json" discovery="yes">
  <url>status</url>
  <item name="local.api.job.discovery" path="jobs:[]" />
  <field name="JOB" path="name"/>
  <field name="JOBID" path="id"/>
</query>
```

With the document of section 10.3.1, the item value is:

```
{"data": [ {"#JOB"}:"backup", {"#JOBID"}:"1"},
           {"#JOB"}:"purge", {"#JOBID"}:"2" ] }
```

Nested iteration

A path can have several iteration steps. With this response:

```
{ "clusters": [
  { "name": "A", "nodes": [ {"host": "a1"}, {"host": "a2"} ] },
  { "name": "B", "nodes": [ {"host": "b1"} ] }
] }
```

Item path and discovered entities

Item path	Entities (and available field path="...")
clusters:[]	2: clusters A and B (name, nodes:[0]:host...)
clusters:[]:nodes:[]	3: nodes a1, a2 and b1 (host)

The `<field>` elements only see the element being iterated: in the second case they cannot obtain the cluster name, and when iterating an object with `*` they cannot obtain each member's name. When this is needed, the response is restructured with a transformation. For example, to discover the nodes together with their cluster:

```
<query name="LocalNodeDiscovery" module="API" type="REST"
  transport="GET" encode="Json" discovery="yes">
  <url>clusters</url>
  <script language="Javascript"><![CDATA[
    function flatten(status_code, result) {
```

```

        if (status_code != 200) throw "ZBX_NOTSUPPORTED";
        var res = { nodes: [] };
        for (var i = 0; i < result.clusters.length; i++) {
            var c = result.clusters[i];
            for (var j = 0; j < c.nodes.length; j++)
                res.nodes.push({ cluster: c.name, host: c.nodes[j].host });
        }
        return res;
    }
    ]]>
</script>
<transform language="Javascript" function="flatten"/>
<item name="local.api.node.discovery" path="nodes:[]" />
<field name="CLUSTER" path="cluster"/>
<field name="HOST" path="host"/>
</query>

```

```

{"data": [ {"#CLUSTER": "A", "#HOST": "a1"}, {"#CLUSTER": "A", "#HOST": "a2"},
           {"#CLUSTER": "B", "#HOST": "b1"} ]}

```

10.4 SOAP queries

A SOAP query calls a method of a SOAP service with fixed arguments and extracts the values from the response:

```

<query name="GoliveWS" module="golive" type="SOAP" transport="POST">
  <url>FoundationEnvironmentManager</url>
  <method uri="http://oracle.e1.bssv.JP000000/" name="getCurrentUserProfile"
    action="">
    <argument name="environment" type="string" value="JPD920"/>
  </method>
  <item name="golive.ws.addressnumber" path="addressNumber" type="INT" default="0" />
  <item name="golive.ws.language" path="userProfile:language" type="STRING"/>
  <item name="golive.ws.exectime" built_in="exec_time"/>
</query>

```

Element / attribute	Description
type="SOAP"	<query> attribute. Mandatory.
transport="POST"	Mandatory.
<url>	<i>Mandatory.</i> Service path, relative to BaseURL.
<method>	Method invoked: uri (method namespace), name (method name) and action (value of the SOAPAction HTTP header, may be empty).
<argument>	Method argument: name, type (string or int) and value. There can be several. Values are fixed.
<item path type>	As in REST, applied to the JSON obtained from the response.
<transform>	Function that processes the response JSON (section 10.4.2).

With this definition, Databbix sends to BaseURL/FoundationEnvironmentManager a POST with the header SOAPAction: "" and a body similar to this one:

```
<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/" ...>
  <SOAP-ENV:Body>
    <ns1:getCurrentUserProfile xmlns:ns1="http://oracle.e1.bssv.JP000000/">
      <environment xsi:type="xsd:string">JPD920</environment>
    </ns1:getCurrentUserProfile>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

If the monitor uses the `Oasis` authentication engine, a *WS-Security* SOAP header with the user and password from the `.conf` is also added.

10.4.1 Converting the SOAP response to JSON

Databbix converts into JSON the *method response element*, that is, the first child of `<Body>`. That element is the root of the JSON: its name is not part of the paths. The conversion follows these rules:

1. An element with child elements becomes an object. Each child is a member with its local name, *without the namespace prefix*.
2. If an element has several children with the same name, that member is an array with all their values, in order.
3. An element without children becomes a *string*, even if its content is a number or a boolean, and even if it has an `xsi:type` attribute.
4. XML attributes are ignored.
5. The order of the object's members may not match the XML order. This does not affect paths.

Since all simple values are strings, transformation functions must convert them to operate on them (`Number(r.sessions)` in JavaScript, `int(r['sessions'])` in Python). Items do not need this: `type="INT"` sends the text as is and turns `true/false` into `1/0`.

Example. Service response:

```
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <soap:Body>
    <ns2:getCurrentUserProfileResponse xmlns:ns2="http://oracle.e1.bssv.JP000000/">
      <addressNumber>4242</addressNumber>
      <active xsi:type="xsd:boolean">true</active>
      <sessions xsi:type="xsd:int">3</sessions>
      <userProfile>
        <language>E</language>
        <dateFormat>MDY</dateFormat>
      </userProfile>
      <role>ADMIN</role>
      <role>USER</role>
    </ns2:getCurrentUserProfileResponse>
  </soap:Body>
</soap:Envelope>
```

Resulting JSON (which the `<transform>` function receives, if there is one):

```
{
  "addressNumber": "4242",
```

```

"active": "true",
"sessions": "3",
"userProfile": { "language": "E", "dateFormat": "MDY" },
"role": [ "ADMIN", "USER" ]
}

```

Paths and values obtained

path	type	Value
addressNumber	INT	4242
active	INT	1
sessions	INT	3
userProfile:language	STRING	E
role:[0]	STRING	ADMIN
role:[1]	STRING	USER

Repeated elements. An element only becomes an array if it appears *more than once*. If the user in the example had a single `<role>`, the JSON would be `"role": "ADMIN"` and the path `role:[0]` would fail. When an element may appear once or several times, normalise the response with a transformation (section 10.4.2).

SOAP errors

If the service answers with a *Fault*, the converted element is the `<Fault>` itself:

```

<soap:Body>
  <soap:Fault>
    <faultcode>soap:Server</faultcode>
    <faultstring>User not authorized</faultstring>
  </soap:Fault>
</soap:Body>

```

```

{ "faultcode": "soap:Server", "faultstring": "User not authorized" }

```

Without a transformation, the item paths do not exist in this JSON and the items become not supported (or take their default value). With a transformation, the error can be detected and handled.

10.4.2 Transforming SOAP responses

The `<transform>` function receives (`status_code`, `result`), as in REST, but `status_code` is always 0 and `result` is the JSON obtained with the rules above. This example detects *Faults*, normalises role so that it is always an array and computes derived values:

```

<query name="LocalUserProfile" module="golive" type="SOAP" transport="POST">
  <url>FoundationEnvironmentManager</url>
  <method uri="http://oracle.e1.bssv.JP000000/" name="getCurrentUserProfile"
    action=""/>
  <script language="Javascript"><![CDATA[
    function profile(status_code, r) {
      if (r.faultcode !== undefined) {
        log_info("SOAP Fault:", r.faultstring);
        throw "ZBX_NOTSUPPORTED";
      }
    }
  ]]>

```

```

    var roles = r.role;
    if (roles === undefined)
        roles = [];
    else if (!(roles instanceof Array))
        roles = [ roles ];           // a single <role>
    return {
        address: Number(r.addressNumber),
        nroles:   roles.length,
        admin:    roles.indexOf("ADMIN") >= 0,
        language: r.userProfile ? r.userProfile.language : ""
    };
}
]]>
</script>
<transform language="Javascript" function="profile"/>
<item name="local.golive.address" path="address" type="INT"/>
<item name="local.golive.nroles" path="nroles" type="INT"/>
<item name="local.golive.admin" path="admin" type="INT"/>
<item name="local.golive.language" path="language" type="STRING"/>
</query>

```

Values obtained with the response of the previous example

Key	Value
local.golive.address	4242
local.golive.nroles	2
local.golive.admin	1
local.golive.language	E

The same function in Python:

```

def profile(status_code, r):
    if 'faultcode' in r:
        log_info("SOAP Fault:", r.get('faultstring'))
        raise Exception("ZBX_NOTSUPPORTED")
    roles = r.get('role', [])
    if not isinstance(roles, list):
        roles = [roles]
    return {
        'address': r.get('addressNumber'),
        'nroles': len(roles),
        'admin': 'ADMIN' in roles,
        'language': r.get('userProfile', {}).get('language', '')
    }

```

10.5 Runtime data

The Webservice engine offers this runtime data:

Name	Value
DATABBIX.PROGRAM_NAME	databbix, or queryexplorer in that utility.
CONN.TYPE	Webservice.

Since queries in this engine have no `<when>` conditions, this data can only be read with `getRuntimeValue` in `queryexplorer`.

10.6 Authentication

10.6.1 Authentication engines and methods

Two things must be distinguished:

Authentication engine. Written in C++ inside Databbix, it implements a complete mechanism: how credentials are sent, how a token is obtained and renewed, etc. There are four: HTTPAuth, RESTToken, JDE and Oasis.

Authentication method. Defined in a query file with `<authentication>`. It takes an engine (`type` attribute) and customises it for a particular service with JavaScript or Python functions and other elements (login URL, headers, where the token goes...). It is identified by its `name` attribute.

Engine	Services	Direct use	Customisation with a method
HTTPAuth	REST and SOAP	Yes: Credentials = method=BASIC...	Optional: <code><auth-generate></code> , <code><header></code> , <code><verify-call></code> .
RESTToken	REST	No (error <i>RESTToken cannot be directly used</i>)	Mandatory: <code><login></code> , <code><token></code> , <code><verify-call></code> .
JDE	REST	Yes	Not supported: a method of this type only gives it another name.
Oasis	SOAP	Yes	Not supported: a method of this type only gives it another name.

10.6.2 How the authentication is chosen

The `Webservice.Authenticator` parameter of the `.conf` contains a name. Databbix looks it up in this order:

1. **An authentication method** in the monitor's query files: an `<authentication>` whose `name` matches. If found, it uses the engine given by its `type`, customised with the method's functions.
2. **An authentication engine** with that name. If found, it is used directly, without customisation, with the credentials from the `.conf`. (The name with the `Authenticator` suffix is also accepted, such as `HTTPAuthAuthenticator`.)
3. If neither is found, the monitor cannot connect and logs the error.

Note. Since methods are looked up first, a method whose `name` matches an engine's (for example, `name="HTTPAuth"`) hides that engine. Use your own names for methods (`AISAuth`, `ApiKeyAuth...`).

10.6.3 Examples

Engine used directly: there is no method called JDE in the XML files, so the JDE engine is used as is.

```
[Webservice]
BaseURL = http://jdesm.example.com:8999/manage/mgmtrestservice/
Authenticator = JDE
UserName = jde_admin
Password = secret
```

Authentication method: the XML file defines AISAuth, which customises the RESTToken engine, and the .conf uses it by name.

```
<Queries engine="Webservice">
  <authentication type="RESTToken" name="AISAuth">
    <script language="Javascript"> ... </script>
    <login type="JsonPost"> ... </login>
    <token type="Header" name="jde-AIS-Auth"/>
    <verify-call language="Javascript" function="verifyCall"/>
  </authentication>
  <query ...> ... </query>
</Queries>
```

```
[Webservice]
Authenticator = AISAuth
```

Programming the functions of an authentication method is described in section 12.7.

Chapter 11

MQTT engine

The MQTT engine subscribes to an MQTT broker and keeps the last message received on each *topic*. Zabbix items read those values or discover the available *topics*. Unlike the other engines, it runs no operation when Zabbix asks for a value: data arrives continuously from the broker, through a receiver thread.

11.1 Configuration: the [MQTT] section

Parameter	Default	Description
Server	<i>mandatory</i>	Broker name or IP.
Port	1883	Broker port.
Id	<i>mandatory</i>	MQTT client identifier. Must be unique on the broker.
UserName	<i>mandatory</i>	User.
Password	<i>mandatory</i>	Password.
Timeout	20000	Maximum connection time, in ms.
Subscription	#	<i>Topic</i> hierarchy to subscribe to. Write \043 for #.
DiscoveryDelay	10000	After startup, time in ms to wait before answering discoveries, to allow all <i>topics</i> to be received.
AbsentDelay	10000	Time in ms before retrying an item whose <i>topic</i> has no new messages.

```
[General]
Name = PLANT1
Engine = MQTT
QueryFile = /etc/databbix/query-mqtt.xml

[Zabbix]
Server = zabbix.example.com
Port = 10051

[Data]
Module = MQTT

[MQTT]
Server = mqtt.example.com
Id = databbix-plant1
UserName = zabbix
Password = secret
```

```
Subscription = plant1/\043      # plant1/#
```

11.2 Query files

MQTT queries have no elements of their own: the operation is always the same (read the last value of a *topic*). The distribution's `query-mqtt.xml` file contains the two queries needed:

```
<Queries engine="MQTT" databbix="2.5.0" query="2.3.0">

  <query name="ItemValue" module="MQTT">
    <parameter order="1" type="STRING"/>
    <item name="ts.mqtt.getvalue"/>
  </query>

  <query name="ItemDiscover" module="MQTT" discovery="yes">
    <parameter order="1" type="STRING"/>
    <item name="ts.mqtt.discovery"/>
    <field name="TOPIC"/>
  </query>

</Queries>
```

Warning. In this engine, item names are fixed: the engine only answers `ts.mqtt.getvalue` and `ts.mqtt.discovery`. An `<item>` with any other name loads without error, but its values are always *not supported*.

Rules for MQTT queries:

- `<parameter>` elements only have `order` (or `value`) and `type`. The parameter is the *topic* or, in discoveries, the *topic* pattern.
- `<item>` elements only have `name`; they accept no attributes saying where the value comes from.
- A discovery must have exactly one `<parameter>` and one `<field>`. The `<field>` only has `name`: the macro takes the full *topic* name as its value.

11.3 Reading values

The key `ts.mqtt.getvalue[topic]` returns the content of the last message published on that *topic*. Given these messages received from the broker:

Last messages received

Topic	Content
plant1/line1/temperature	21.5
plant1/line1/pressure	1.02
plant1/line2/temperature	22.1
plant1/status	{"mode": "auto", "alarms": 0}

Values obtained

Key	Value
<code>ts.mqtt.getvalue[plant1/line1/temperature]</code>	21.5
<code>ts.mqtt.getvalue[plant1/status]</code>	<code>{"mode":"auto","alarms":0}</code>
<code>ts.mqtt.getvalue[plant1/line3/temperature]</code>	<i>not supported: never received</i>

The content is sent as is. If it is JSON, as in `plant1/status`, its fields can be extracted in Zabbix with *JSONPath* preprocessing on the item.

A value is only sent to Zabbix when a new message has arrived since the last time. If none has arrived, the item is retried after **AbsentDelay** milliseconds, instead of repeating the previous value.

11.4 Topic discovery

The key `ts.mqtt.discovery[pattern]` returns all the received *topics* that match the pattern. The pattern uses MQTT wildcards:

Wildcard	Meaning
<code>+</code>	Exactly one level, any value.
<code>#</code>	Any number of levels.

With the messages of the previous section:

Patterns and discovered topics

Key	Values of {#TOPIC}
<code>ts.mqtt.discovery[plant1/+/temperature]</code>	plant1/line1/temperature plant1/line2/temperature
<code>ts.mqtt.discovery[plant1/line1/#]</code>	plant1/line1/temperature plant1/line1/pressure
<code>ts.mqtt.discovery[plant1/+]</code>	plant1/status plant1/line1/temperature plant1/line1/pressure plant1/line2/temperature

Note. Unlike standard MQTT, in Databbix a pattern also matches *topics* with more levels than itself: it works as a prefix. This way, `plant1/+` discovers in one go all the *topics* under any sublevel of `plant1`. To narrow the result to a specific level, write the pattern with all its levels (`plant1/+/temperature`).

The value of the first item is:

```
{"data": [ {"{#TOPIC}": "plant1/line1/temperature"},
            {"{#TOPIC}": "plant1/line2/temperature"} ]}
```

In Zabbix, create a discovery rule with that key and this item prototype:

`ts.mqtt.getvalue[{#TOPIC}]`

Only *topics* for which some message has already been received are discovered. That is why discoveries are delayed during the first **DiscoveryDelay** milliseconds after startup.

11.5 Runtime data

The MQTT engine offers this runtime data:

Name	Value
DATABBIX.PROGRAM_NAME	databbix, or queryexplorer in that utility.
CONN.TYPE	MQTT.

Since queries in this engine have no `<when>` conditions, this data can only be read with `getRuntimeValue` in `queryexplorer`.

Part IV

Tools and usage

Chapter 12

Script programming

Scripts let you add your own logic where the XML alone falls short: converting formats, computing derived values, restructuring or filtering a service response and programming the authentication dialogue with a web service. This chapter explains the environment scripts run in and how to write them for each of those uses.

12.1 Available languages

Language	Name in the XML	Interpreter
JavaScript	Javascript	Duktape 2.5, embedded in Databbix. Implements ECMAScript 5.1.
Python	Python	The system's CPython, embedded in Databbix.

All the scripts of a query collection (including those of authentication methods) must be written in the same language.

12.1.1 JavaScript

- Write the code in ECMAScript 5.1: `var` and `function`. More modern constructs such as `let`, `const`, arrow functions or template literals may not be available.
- The standard language objects are available (`Math`, `Date`, `JSON`, `RegExp...`), but there is no access to files, network, timers or modules (`require`).
- The `Date.parse()` function only reliably recognises ISO 8601 dates, for example: `2026-09-29T10:15:00Z`. If the database returns dates in another format, convert them in the SQL (`to_char`, `extract(epoch...)`) or parse them by hand in the script.

12.1.2 Python

- The Python version installed on the system is used, with its standard library: `re`, `datetime`, `json`, etc. can be imported.
- The code runs inside the Databbix process. Avoid slow or blocking operations (network, disk): they delay the whole monitor.

12.2 Where scripts are written

A script is a `<script>` element with the `language` attribute. The code is written inside a `CDATA` section so that the characters `<`, `>` and `&` need not be escaped:

```
<script language="Javascript"><![CDATA[
  function kb2mb(value) {
    return Math.round(Number(value) / 1024);
  }
  ]]>
</script>
```

It can appear in three places:

In `<Queries>`. Global script. Its functions are available to all the queries of the collection.

In `<query>`. Query script. Its functions are only available to that query. It can also use the global functions.

In `<authentication>`. Authentication script. Its functions are used by the elements of that authentication method.

A script defines functions, which are then referenced by name:

Reference	Use
<code>transform</code> attribute of <code><item></code>	Transformation of a value (sections 12.4 and 12.5).
<code><transform></code> element of <code><query></code>	Transformation of a web service response (section 12.6).
Functions of <code><authentication></code>	Web service authentication (section 12.7).

12.3 Execution environment

12.3.1 Scope and state

Each monitor has its own interpreter. Within it:

- Global scripts run once, when the collection is loaded.
- Each query has its own variable space, which starts as a copy of the global variables and to which the query's scripts are added.
- That space is kept while the monitor runs. A global variable modified by a function keeps its value in the next execution *of the same query*, but other queries do not see it.

This makes it possible, for example, to compute a rate from two consecutive readings:

```
<query name="LocalCommitRate" module="PG">
  <script language="Javascript"><![CDATA[
    var previous = null;           // kept between executions
    function rate(value) {
      var now = { t: Date.now() / 1000, v: Number(value) };
      var r = 0;
      if (previous !== null && now.t > previous.t)
        r = (now.v - previous.v) / (now.t - previous.t);
      previous = now;
      return r.toFixed(2);
    }
  ]]>
</script>
```

```
<sql><![CDATA[
  select sum(xact_commit) as commits from pg_stat_database ]]>
</sql>
<item name="local.pg.commitrate" column="commits" transform="Javascript:rate"/>
</query>
```

Note. State is lost when the service restarts. Also, in Python the copy of the global variables is shallow: a list or dictionary created in a global script is the same object in all queries. If you need independent state, create it inside the query's script.

12.3.2 Logging functions

Scripts can write to the monitor's log with two functions, which accept any number of arguments:

Function	Log level
<code>log_debug(...)</code>	Debug (visible with LogLevel 7 or higher).
<code>log_info(...)</code>	Information (visible with LogLevel 6 or higher).

Messages appear with the prefix `From Javascript:` or `From Python:`.

```
<script language="Javascript"><![CDATA[
  function state(value) {
    log_debug("state received:", value);
    return value == "OPEN" ? 1 : 0;
  }
]]>
</script>
```

12.3.3 Data types

Databbix passes data to functions in two ways:

- **As text:** the value of a column or of a JSON path. Even if the column is numeric, the function receives a string; use `Number(value)` in JavaScript or `float(value)` in Python to operate on it.
- **As an object:** web service responses, the JSON of a discovery and credentials. Databbix decodes the JSON before calling the function and encodes the returned object as JSON. If the text received is not valid JSON, the function receives the string as is.

12.3.4 Errors

If a function throws an exception, the operation fails and the exception message appears in the monitor's log. The effect depends on the use: in an item transformation, the item becomes not supported; in the transformation of a web service response, all the query's items do.

To deliberately signal that a piece of data is not available (without logging it as an error), a web service transformation can throw exactly the string `ZBX_NOTSUPPORTED`:

```
throw "ZBX_NOTSUPPORTED";           // JavaScript
```

```
raise Exception("ZBX_NOTSUPPORTED") # Python
```

Warning. In JavaScript you must throw the string, not an `Error` object: `throw new Error("ZBX_NOTSUPPORTED")` produces the text `Error: ZBX_NOTSUPPORTED`, which Databbix does not recognise.

12.4 Transforming values in SQL queries

The `transform` attribute of an `<item>` applies a function to the column value before sending it to Zabbix. It has the form `Language:function`:

```
<item name="local.ora.lastbackup.age" column="end_time" transform="Javascript:age"/>
```

Signature	function(value)
value	Column value, as text.
Returns	The value sent to Zabbix. It is converted to text: a number is sent as is, <code>true/false</code> as <code>true/false</code> (JavaScript) or <code>True/False</code> (Python).

The function is called once per item and execution. If the query is a `FILTER` query, it is called for the row matching the item key.

12.4.1 Example: dates

Seconds elapsed since a date, and a date as a UNIX timestamp. So that `Date.parse` works with any database, the SQL returns the date in ISO 8601 format:

```
<script language="Javascript"><![CDATA[
    function age(date) {
        return Math.round((Date.now() - Date.parse(date)) / 1000);
    }
    function toUnix(date) {
        return Math.round(Date.parse(date) / 1000);
    }
    ]]>
</script>

<query name="LocalLastBackup" module="DB">
    <sql><![CDATA[
        select to_char(max(completion_time) at time zone 'UTC',
                        'YYYY-MM-DD"THH24:MI:SS"Z') as end_time
        from v$backup_set
    ]]>
    </sql>
    <item name="local.ora.lastbackup"      column="end_time" transform="
Javascript:toUnix"/>
    <item name="local.ora.lastbackup.age" column="end_time" transform="
Javascript:age"/>
</query>
```

12.4.2 Example: mapping states to numbers

Zabbix works better with numeric values for triggers. A mapping table avoids writing the logic in each database's SQL:

```
<script language="Javascript"><![CDATA[
    var STATES = { "OPEN": 0, "MOUNTED": 1, "STARTED": 2 };
    function instanceState(value) {
        var v = STATES[String(value).trim()];
        return (v === undefined) ? 99 : v;
    }
    ]]
```

```

    ]]>
</script>

<query name="LocalInstanceStatus" module="Instance">
  <sql><![CDATA[ select status from v$instance ]]></sql>
  <item name="local.ora.instance.status" column="status"
        transform="Javascript:instanceState"/>
</query>

```

12.4.3 Python example

The same idea in Python, extracting a version number from a text:

```

<script language="Python"><![CDATA[
import re

def major_version(value):
    m = re.search(r'(\d+)\.', value)
    return int(m.group(1)) if m else 0
]]>
</script>

<query name="LocalPGVersion" module="PG">
  <sql><![CDATA[ select version() as v ]]></sql>
  <item name="local.pg.version.major" column="v" transform="Python:major_version"/
  >
</query>

```

Note. In Python, indentation is significant: the code inside CDATA must start in the first column, as in the example.

12.5 Transforming discoveries

In a discovery query, the `transform` of the `<item>` receives the whole discovery JSON as an object and must return an object with the same structure. It is used to filter entities or add computed macros.

Signature	function(lld)
lld	Object { data: [{ "{#MACRO}": value, ... }, ...] }.
Returns	Object with the same shape, which is sent to Zabbix.

```

<query name="LocalTSDiscovery" module="DB" discovery="yes">
  <script language="Javascript"><![CDATA[
    function dataOnly(lld) {
      var res = { data: [] };
      for (var i = 0; i < lld.data.length; i++) {
        var e = lld.data[i];
        // skip system tablespaces
        if (/^(SYSTEM|SYSAUX)$/.test(e["{#TSNAME}"]))
          continue;
        // new macro: lower-case name
        e["{#TSLOWER}"] = e["{#TSNAME}"].toLowerCase();
        res.data.push(e);
      }
    }
  ]]>
</script>
</query>

```

```

    }
    return res;
  }
]]>
</script>
<sql><![CDATA[
  select tablespace_name from dba_tablespaces where status = 'ONLINE' ]]>
</sql>
<item name="local.ora.tspace.discovery" transform="Javascript:dataOnly"/>
<field name="TSNAME" column="tablespace_name"/>
</query>

```

Note. Filtering in the SQL (`where`) is more efficient. The transformation is useful when the criterion is hard to express in SQL or when the same XML file is used with several databases.

12.6 Transforming web service responses

In REST queries, the query's `<transform>` element processes the service response *before* the values are extracted with the items' paths (`path`):

```

<query name="..." module="..." type="REST" transport="GET" encode="Json">
  <url>...</url>
  <transform language="Javascript" function="myFunction"/>
  ...
</query>

```

Signature	function(status_code, result)
status_code	HTTP status of the response (200, 404, 500...). In SOAP queries it is 0.
result	Response body decoded as JSON, or the text string if it is not JSON. In SOAP, the response converted into JSON (section 10.4.1).
Returns	Object to which the <code>path</code> of the items and <code><field></code> elements is applied.

Warning. Without `<transform>`, a response with a status other than 200 is an error. With `<transform>`, the function receives *all* responses and is responsible for handling error codes.

The transformation runs once per service call. Its typical uses are three: handling errors, restructuring the output and filtering it.

12.6.1 Handling error codes

The JD Edwards service returns a 500 error with its own code when the instance agent is down. The function turns it into a state Zabbix can watch, and marks as not supported what makes no sense to measure:

```

<query name="JDEInstanceState" module="JDE" type="REST" transport="GET" encode="Json">
  <url>instancestate</url>
  <script language="Javascript"><![CDATA[
    function instancestate(status_code, result) {
      if (status_code == 200)

```

```

        return result;
    if (status_code == 500) {
        switch (result.errorCode) {
            case 13: return { instanceState: "AGENT_DOWN" };
            case 14: throw "ZBX_NOTSUPPORTED";
        }
    }
    throw "HTTP error " + status_code;
}
]]>
</script>
<transform language="Javascript" function="instancestate"/>
<parameter urlparameter="instanceName" order="1" type="STRING"/>
<item name="ts.jde.instance.state" path="instanceState" type="STRING"/>
</query>

```

12.6.2 Restructuring the output

Often the figure you want in Zabbix is not directly in the response: it has to be counted, summed or computed. The function builds a new object with the values already computed, and the items point to it:

```

<query name="LocalJobSummary" module="API" type="REST" transport="GET" encode="Json"
>
<url>jobs</url>
<script language="Javascript"><![CDATA[
    function summary(status_code, result) {
        if (status_code != 200)
            throw "ZBX_NOTSUPPORTED";
        var r = { total: 0, running: 0, failed: 0, maxDuration: 0 };
        for (var i = 0; i < result.jobs.length; i++) {
            var j = result.jobs[i];
            r.total++;
            if (j.state == "RUNNING") r.running++;
            if (j.state == "FAILED") r.failed++;
            if (j.duration > r.maxDuration) r.maxDuration = j.duration;
        }
        return r;
    }
    ]]>
</script>
<transform language="Javascript" function="summary"/>
<item name="local.api.jobs.total" path="total" type="INT"/>
<item name="local.api.jobs.running" path="running" type="INT"/>
<item name="local.api.jobs.failed" path="failed" type="INT"/>
<item name="local.api.jobs.maxdur" path="maxDuration" type="INT"/>
</query>

```

The same technique works for responses that are not JSON: if the service returns text, **result** is a string that the function can parse and turn into an object.

12.6.3 Filtering the output

In a REST discovery, the function can keep only the elements of interest. Here, of all the instances returned by the service, only the *enterprise* servers are discovered:

```

<query name="JDEDiscoverEnterprise" module="JDE" type="REST"
    transport="GET" encode="Json" discovery="yes">

```

```

<url>instancesinfo</url>
<script language="Javascript"><![CDATA[
    function filterEnterprise(status_code, result) {
        if (status_code != 200)
            throw "ZBX_NOTSUPPORTED";
        var res = { result: [] };
        for (var i = 0; i < result.result.length; i++)
            if (result.result[i].targetType == "entserver")
                res.result.push(result.result[i]);
        return res;
    }
    ]]>
</script>
<transform language="Javascript" function="filterEnterprise"/>
<item name="ts.jde.enterprise.discovery" path="result: []"/>
<field name="INSTANCE" path="instanceName"/>
<field name="TYPE" path="targetType"/>
</query>

```

12.6.4 SOAP responses

In SOAP queries the function receives the JSON obtained from the XML response and `status_code` is always 0. Errors must be detected by looking for a *Fault* in the JSON itself, bearing in mind that a repeated element may arrive as an array or as a single value. Section 10.4.2 shows a complete example in JavaScript and Python.

12.6.5 Combining with item transformations

The items of a REST query also accept the `transform` attribute, which is applied afterwards to the value of each path, just as in SQL queries (section 12.4). In INT items, a `true/false` result is turned into 1/0.

12.7 Programming web service authentication methods

An authentication method (section 10.6) customises an authentication engine with its own functions. Only the HTTPAuth and RESTToken engines can be customised. The method's functions decide how the credentials or the token are obtained and how to detect that it is necessary to authenticate again.

12.7.1 Credentials

All functions that build credentials receive the same object, generated from the monitor file:

```

[Webservice]
UserName = monitor
Password = secret
Credentials = environment=JPD920
Credentials = role=*ALL

```

```

{ "username": "monitor", "password": "secret",
  "environment": "JPD920", "role": "*ALL" }

```

`UserName` and `Password` become the keys `username` and `password`; each `Credentials` line adds a key. This way, the XML file describes the mechanism and the `.conf` file holds the data of each installation.

12.7.2 Call verification

After each call to the service, Databbix asks the `<verify-call>` function whether the response is valid:

Signature	<code>function(status_code, result)</code>
Returns	Object with: <code>call_ok</code> (<code>true</code> if the call is valid), <code>error</code> (error text) and, in HTTPAuth only, <code>reauthenticate</code> (<code>true</code> if it is necessary to authenticate again).

If `call_ok` is `false`, Databbix authenticates again and repeats the call *once*. In HTTPAuth, if `reauthenticate` is also `false`, the call fails straight away with the given error (invalid credentials).

12.7.3 Methods based on the RESTToken engine

The engine obtains a token with a *login* request and sends it in all subsequent calls.

```
<authentication type="RESTToken" name="name">
  <script language="Javascript"> ...functions... </script>
  <login type="JsonPost">
    <url>login/path</url>
    <request-function language="Javascript" function="...">
    <accept-function language="Javascript" function="...">
  </login>
  <token type="Header|Cookie" name="...">
  <verify-call language="Javascript" function="...">
</authentication>
```

Element	Description
<code><login></code>	Login request. Sent as a POST to BaseURL + <code><url></code> , with a JSON body. Use <code>type="JsonPost"</code> .
<code><request-function></code>	<code>function(credentials)</code> . Returns the object sent as the body of the login request.
<code><accept-function></code>	<code>function(status_code, result)</code> . Analyses the login response and returns <code>{authentication_ok, token, error}</code> .
<code><token></code>	Where the token is sent in calls: in a header (<code>type="Header"</code>) or in a cookie (<code>type="Cookie"</code>), with the name <code>name</code> .
<code><verify-call></code>	Verification of each call. Mandatory.

Complete example for the JD Edwards AIS server:

```
<authentication type="RESTToken" name="AISAuth">
  <script language="Javascript"><![CDATA[
    function authGenerate(credentials) {
      // body of the login request
      return {
        username: credentials.username,
        password: credentials.password,
        role: credentials.role,
        environment: credentials.environment
      };
    }
  ]]>
</script>
<login type="JsonPost">
  <url>login/path</url>
  <request-function language="Javascript" function="authGenerate">
  <accept-function language="Javascript" function="...">
</login>
<token type="Header|Cookie" name="...">
<verify-call language="Javascript" function="...">
</authentication>
```

```

function authAccept(status_code, result) {
  if (status_code == 200)
    return { authentication_ok: true,
            token: result.userInfo.token };
  var errors = {
    400: "Bad request",
    403: "Invalid credentials",
    415: "Wrong Content-Type"
  };
  return { authentication_ok: false,
          error: errors[status_code] || ("Error " + status_code) };
}

function verifyCall(status_code, result) {
  // 444: token expired -> authenticate again
  if (status_code == 403 || status_code == 444)
    return { call_ok: false, error: "Invalid token" };
  return { call_ok: true };
}
}]>
</script>
<login type="JsonPost">
  <url>v2/tokenrequest</url>
  <request-function language="Javascript" function="authGenerate"/>
  <accept-function language="Javascript" function="authAccept"/>
</login>
<token type="Header" name="jde-AIS-Auth"/>
<verify-call language="Javascript" function="verifyCall"/>
</authentication>

```

Note. If `WebService.AuthPersistFile` is set, the token is saved in that file and reused after a restart, without requesting a new one. The file contains a secret: protect it like the `.conf` itself.

12.7.4 Methods based on the HTTPAuth engine

The engine uses HTTP authentication mechanisms. Without `<authentication>`, it takes the credentials directly from the `.conf` (section 3.15). With `<authentication>`, it accepts:

Element	Description
<code><auth-generate></code>	<code>function(credentials)</code> . Returns the effective credentials: <code>{method, username, password}</code> with method BASIC, DIGEST or NTLM; or <code>{method: "HEADER", header, value}</code> to send a fixed header.
<code><header></code>	Additional header sent in all calls, with the <code>name</code> and <code>value</code> attributes. The value can also be given as the element's text, and accepts <code>\${...}</code> variables.
<code><verify-call></code>	Verification of each call. Optional: if absent, only a response with status 200 is valid, and any other status is treated as a credentials error.

Example: a service that authenticates with an API key in the `X-API-Key` header. The key is configured in the `.conf`, and the XML file also adds a fixed header with the API version:

```
<authentication type="HTTPAuth" name="ApiKeyAuth">
  <script language="Javascript"><![CDATA[
    function apiKey(credentials) {
      return { method: "HEADER",
                header: "X-API-Key",
                value:  credentials.apikey };
    }
    function verify(status_code, result) {
      if (status_code == 401)
        return { call_ok: false, reauthenticate: false,
                  error: "API key rejected" };
      return { call_ok: true };
    }
  ]]>
</script>
<auth-generate language="Javascript" function="apiKey"/>
<header name="Accept-Version" value="2"/>
<verify-call language="Javascript" function="verify"/>
</authentication>
```

```
[Webservice]
BaseURL = https://api.example.com/v2/
Authenticator = ApiKeyAuth
Credentials = apikey=3f9a...c21
```


Chapter 13

The queryexplorer utility

`queryexplorer` is an interactive tool to test a monitor's configuration without starting the daemon and without sending anything to Zabbix. It loads the same `.conf` file and the same XML files the monitor would use, connects to the data source and lets you run items one by one and see the result. It is used to:

- check that a monitor file and its XML files are correct;
- check that the connection to the database or service works (and, for web services, that authentication is correct);
- test a new query and its transformations before creating the items in Zabbix;
- inspect contexts, environment variables and runtime data, and evaluate `<when>` conditions.

It is installed in `/opt/trustserver/bin/queryexplorer`.

13.1 Invocation

```
queryexplorer [options] -f <monitor.conf> [command [arguments] ...]
```

Option	Description
<code>-f, --conf-config-file <i>f</i></code>	<i>Mandatory.</i> Monitor file to test. If the path does not start with <code>/</code> , it is looked up in the monitor directory (<code>/etc/databbix/dbs</code>).
<code>-l, --log-console</code>	Also shows the log on the console. Highly recommended: without it, loading errors only go to the log file.
<code>-v, --logLevel <i>n</i></code>	Log level (overrides the one in <code>databbix.conf</code>). With 9, SQL statements, HTTP calls and their responses are shown.
<code>-c, --config-file <i>f</i></code>	Global configuration file. Default: <code>/etc/databbix/databbix.conf</code> .
<code>-p, --config-path <i>d</i></code>	Monitor directory, to resolve relative paths in <code>-f</code> .
<code>-m, --module-path <i>d</i></code>	Additional module directories.
<code>-h, --help</code>	Shows help.

On startup, `queryexplorer` does the same as a monitor:

1. reads `databbix.conf` and the file given with `-f`;

2. loads the XML query files;
3. connects to the data source.

If any of these steps fails, it shows the error and exits. Therefore, *starting without errors already validates the configuration and the connection.*

Note. `queryexplorer` writes its log to the global log file (`LogPath/LogFile` in `databbix.conf`). Run it as the `databbix` user (or another user with write permission on that file) and with the same environment variables as the service.

```
sudo -u databbix env TNS_ADMIN=/etc/databbix ODBCINI=/etc/databbix/odbc.ini \
LD_LIBRARY_PATH=/opt/trustserver/lib64 \
/opt/trustserver/bin/queryexplorer -l -f oraprod.conf
```

`queryexplorer` opens its own connection, so it can be used while the service is running.

13.2 Interactive mode

With no commands on the command line, `queryexplorer` enters interactive mode:

```
$ queryexplorer -l -f pgprod.conf

Databbix QueryExplorer: Interactive Query Tool Version 2.5.0-2639.
Databbix Database Engine Version 2.5.0-2639.
(C) Trustserver SL 2014-2024. All Rights Reserved.

QueryExplorer> get tez.pg.sessions[sales]
Item: tez.pg.sessions[sales], value: 12
QueryExplorer> quit

Quitting Databbix. See you soon!
```

- The `Tab` key completes command names and, depending on the command, item, query, context or variable names.
- The arrow keys browse the history, which is saved in `~/.queryexplorer_history`.
- Exit with `quit` or `Ctrl-D`.
- Commands are case-insensitive. Arguments are separated by spaces, so an item key cannot contain spaces.

13.3 Non-interactive mode

If commands are given after the options, they are run in order and the program exits. Each command takes its arguments from the words that follow it:

```
queryexplorer -l -f pgprod.conf \
  get 'tez.pg.sessions[sales]' \
  get 'tez.pg.sessions[postgres]' \
  get tez.pg.db.discovery
```

Quote keys with brackets so that the shell does not interpret them. This mode is useful in check scripts after an installation or upgrade.

13.4 Commands

13.4.1 Items and queries

Command	Description
<code>get <i>key</i></code>	Runs the item's query and shows its value, exactly as the monitor would compute it: with the key's parameters and context and applying transformations. It ignores the cache: the query is always run.
<code>listItems</code>	Lists the names of all the items in the collection.
<code>listItems full</code>	Lists all queries with their text (SQL or URL) and, for each item, its parameters and the column or path it comes from.
<code>listQueries</code>	Lists the query names.
<code>descQuery <i>name</i></code>	Shows a query (by its name or by one of its items' names): text and list of items.
<code>descItem <i>item</i></code>	Shows the query an item belongs to and where its value comes from.

13.4.2 Contexts

Command	Description
<code>getDefinedContexts</code>	Contexts declared in the XML files, with their default value.
<code>getSupportedContexts</code>	Contexts supported by the connection, with their current value.
<code>getContext <i>name</i></code>	Current value of a context in the connection.
<code>setContext <i>name value</i></code>	Changes a context of the connection. Useful to check that it can be entered (for example, a particular database).
<code>setContext <i>name</i></code>	Resets the context to its default value.
<code>setContext</code>	Resets all contexts to their default values.

The `get` command switches to the context given by the key and, when done, leaves the connection in the context it was in.

13.4.3 Variables and expressions

Command	Description
<code>getEnvironment <i>name</i></code>	Value of an environment variable, as the monitor sees it (combining the XML file, <code>databbix.conf</code> and the <code>.conf</code>).
<code>getRuntimeNames</code>	Names of the runtime data offered by the connection.
<code>getRuntimeValue <i>name</i></code>	Value of a runtime datum.
<code>evaluateExpression <i>expression</i></code>	Evaluates an expression like <code><when></code> conditions. Shows 1 if it holds and 0 otherwise.

Note. In `queryexplorer`, `runtime('DATABBIX.PROGRAM_NAME')` returns `queryexplorer` instead of `databbix`. A condition can use it to behave differently during testing.

Warning. In interactive mode, `evaluateExpression` takes the whole rest of the line as the expression. In non-interactive mode it takes a single argument, so the expression must be quoted for the shell to pass it as one parameter:

```
queryexplorer -l -f oraprod.conf \
  evaluateExpression "runtime('DB.DATABASE') = 'Oracle'"
```

13.4.4 Other

Command	Description
help	List of commands.
quit	Leaves interactive mode.

13.5 Use cases

13.5.1 Validating a monitor file and its connection

```
queryexplorer -l -f oraprod.conf listQueries
```

If the command lists the queries, the `.conf` is correct, the XML files loaded without errors and the connection works. Otherwise, the console log says what failed: a missing mandatory parameter, a syntax error in an XML file (with the query name) or the driver's connection error.

13.5.2 Validating web service authentication

In a `WebService` monitor, authentication happens on the first call. Requesting an item with a detailed log is enough to see the login request, the response and the result of the authentication functions:

```
queryexplorer -l -v 9 -f aisprod.conf get ts.ais.scheduler.state
```

13.5.3 Testing a new query

```
QueryExplorer> descQuery LocalInvalidObjects
QueryExplorer> get local.ora.invalid[HR]
Item: local.ora.invalid[HR], value: 3
QueryExplorer> get local.ora.invalid[NOSUCH]
Item: local.ora.invalid[NOSUCH], value: 0
QueryExplorer> get local.ora.schema.discovery
Item: local.ora.schema.discovery, value: {"data":[{"#SCHEMA":"HR"}, ...]}
```

The second `get` checks that the `default` attribute works when the row does not exist. If a value is not as expected, repeat with `-v 9` to see the statement run, the rows obtained and the calls to the transformation functions.

13.5.4 Checking alternative SQL

To find out which branch of a `<multiple>` will be used with a database:

```
QueryExplorer> getRuntimeValue DB.DATABASE
DB.DATABASE = 'Oracle'
QueryExplorer> evaluateExpression runtime('DB.DATABASE') = 'Oracle'
Result: 1
```

13.5.5 Checking contexts

To verify that the user can enter all the databases of an SQL Server before enabling a discovery across contexts:

```
QueryExplorer> getDefinedContexts
Contexts defined in query file:
- DATABASE, default: master
QueryExplorer> setContext Database sales
QueryExplorer> getContext Database
Actual value for context 'Database': 'sales'
QueryExplorer> setContext
```


Chapter 14

Practical guide: creating a new query

This chapter walks through the whole process with an example: monitoring the number of invalid objects in each Oracle schema.

14.1 Step 1: write and test the SQL

First, write the query and test it with the same user Databbix uses, for example with SQL*Plus or psql. Check that it returns one row per monitored object and that the columns have clear aliases:

```
select owner as schema_name, count(*) as invalid_count
from dba_objects
where status = 'INVALID'
group by owner;
```

14.2 Step 2: choose the query type

- Does it always return one row? → basic query.
- Does it return one row per object, and you want items per object? → FILTER.
- Is the set of possible values huge and only a few are of interest? → BIND.
- Do you want Zabbix to create the items automatically? → a **discovery** query plus a **FILTER** one.

In the example, a schema with no invalid objects does not appear in the result, so **FILTER** is used with `default="0"`, plus a discovery for the schemas.

14.3 Step 3: write the XML

It is best to create your own file instead of modifying the distribution's files, which are overwritten on upgrade. Save it as `/etc/databbix/query-local.xml`:

```
<?xml version="1.0" encoding="UTF-8"?>
<Queries engine="Database" query="2.5.0">

<query name="LocalSchemaDiscovery" module="DB" discovery="yes">
  <sql><![CDATA[
    select username as schema_name
    from dba_users
    where oracle_maintained = 'N'
  ]]></sql>
```

```

    ]]>
  </sql>
  <item name="local.ora.schema.discovery"/>
  <field name="SCHEMA" column="schema_name"/>
</query>

<query name="LocalInvalidObjects" module="DB" select="FILTER" maxCache="60">
  <sql><![CDATA[
    select owner as schema_name, count(*) as invalid_count
    from dba_objects
    where status = 'INVALID'
    group by owner
  ]]>
  </sql>
  <parameter column="schema_name" order="1" type="STRING"/>
  <item name="local.ora.invalid" column="invalid_count" default="0"/>
  <item name="local.ora.invalid.exectime" built_in="exec_time"/>
</query>

</Queries>

```

Item names. Item names must be unique across all the files the monitor loads. Using a prefix of your own (local., your company name...) avoids clashes with the distribution's items (tez.*, ts.*).

14.4 Step 4: add the file to the monitor

QueryFile in the monitor *replaces* the global list, so the main file must be included too:

```

[General]
Name = ORAPROD
QueryFile = /etc/databbix/query.xml
QueryFile = /etc/databbix/query-local.xml

```

14.5 Step 5: test with queryexplorer

Before creating anything in Zabbix, test the query with **queryexplorer** (chapter 13), without touching the service:

```
sudo -u databbix /opt/trustserver/bin/queryexplorer -l -f oraprod.conf
```

```

QueryExplorer> get local.ora.schema.discovery
QueryExplorer> get local.ora.invalid[HR]
QueryExplorer> get local.ora.invalid.exectime[HR]

```

If the XML file has an error, **queryexplorer** never shows the prompt and the message explains which query fails. If a value is not as expected, repeat with **-v 9** to see the SQL and the rows obtained. When everything is correct, restart the service so that the monitor loads the new file:

```
systemctl restart databbix
```

14.6 Step 6: create the items in Zabbix

On the ORAPROD host (or in a template):

1. A discovery rule of type *Zabbix agent (active)* with key:
`local.ora.schema.discovery`
2. A numeric item prototype of type *Zabbix agent (active)* with key:
`local.ora.invalid[{-#SCHEMA}]`
3. Optionally, a trigger prototype, for example:
`last(/ORAPROD/local.ora.invalid[{-#SCHEMA}])>0`

Data arrives when Databbix next requests the active item list, that is, within `Zabbix.Refresh` seconds at most (300 by default), or immediately if the service is restarted.

Chapter 15

Troubleshooting

15.1 Log levels

Level	What is logged
3	Errors (default value).
4	Warnings.
5	Relevant events: monitor startup, connection errors and retries.
6	Operational information: executions and time to the next cycle.
7	Debug, including the monitor's full configuration at startup.
9	Also, the text of every SQL statement run and every row obtained.
10	Also, the full structure of the XML files parsed.

Table 15.1: Log levels

The level can be set for the whole daemon in `databbix.conf` or for a single monitor in its file. Levels 9 and 10 produce a lot of output: use them only while testing.

Security. High log levels are meant for debugging and record sensitive data. From level 7, the log includes the session tokens obtained by the web service authenticators. From level 9, it also includes the full credentials (user, password and any other data) sent to the service, because they may be the cause of the problem being debugged. Log files are created with permissions 0640; even so, when you finish debugging, lower the level and delete the logs written at those levels.

Each monitor writes to its own file, by default `/var/log/databbix/<Name>.log`. The main process writes to `databbix.log`.

15.2 Common problems

Symptom	Likely cause and solution
The monitor does not start: <i>“General.Name is not present”</i> or another mandatory parameter	That parameter is missing from the <code>.conf</code> . Check the <code>[General]</code> , <code>[Zabbix]</code> and <code>[Data]</code> sections and the engine section.
<i>“Monitor initialization error”</i> with a syntax error	Error in the XML file: unknown attribute, query without <code>module</code> , duplicate item... The message names the affected query.

Symptom	Likely cause and solution
<i>“Item name ... already defined”</i>	Two queries define the same item name in the loaded files.
<i>“Zabbix server error ..., retry in 60 seconds”</i>	The Zabbix server cannot be reached, or the server does not know the host. Check Server , Port , the firewall and that the Zabbix <i>Host name</i> matches General.Name .
<i>“Error connecting to data source”</i>	Wrong connection string, user or password, or TNS_ADMIN/ODBCINI not set for the service.
The item shows as not supported in Zabbix	The item name does not exist in the loaded XML files, the number of key parameters does not match, the requested row does not exist (use default), the basic query returns 0 or several rows, or the SQL fails (see the log at level 9).
<i>“Excessive execution time”</i>	The query takes longer than maxExecTime/MaxQueryExecTimeLog . Optimise the SQL or raise maxCache to run it less often.
<i>“Too many errors ..., exiting”</i>	The monitor exceeded MaxErrors consecutive failed executions and stopped. Fix the cause and restart the service.
New items take a while to receive data	Databbix requests the item list every Zabbix.Refresh seconds. Wait or restart the service.
Changes to the XML or .conf files are not applied	They are only read at startup. Restart the service.

Part V

Appendices

Appendix A

XML quick reference

This appendix summarises, for each query type, which elements and attributes are mandatory, which are optional and which are not accepted. In the outlines, square brackets mark optional parts and the vertical bar separates alternatives.

A.1 <Queries> level (all engines)

```
<Queries [engine="Database|Webservice|MQTT"] [query="2.5.0"] [databbix="2.5.0"]>
  <environment name="VAR" default="value"/>                                <!-- all -->
  <script language="Javascript|Python"> ... </script>                    <!-- all -->
  <context name="Database" module="MOD" default="value"/>                <!-- Database -->
  <authentication type="RESTToken|HTTPAuth" name="N"> ...                <!-- Webservice -->
  <query ...> ... </query>
</Queries>
```

In all queries, `name` and `module` are mandatory, and `maxCache`, `maxExecTime` and `<script>` are optional. `<field>` elements are only accepted in discovery queries.

A.2 Constant query (all engines)

```
<query name="..." module="..." type="Constant">
  <item name="key" value="value"/>
</query>
```

Mandatory: `type="Constant"`; one or more `<item>` with `name` and `value`.

Optional: `transform` on the `<item>` elements.

Not accepted: `<sql>`, `<url>`, `<parameter>`, `<field>`.

A.3 Database engine

A.3.1 Basic query

```
<query name="..." module="...">
  <sql> text </sql>
  [<default context="C" value="v"/>]
  [<parameter context="C" order="N"/>]
  <item name="key" column="COL" [transform="L:f"]/>
  [<item name="key" built_in="exec_time|exec_success"/>]
</query>
```

Mandatory: one <sql>; one or more <item> with column (or built_in).
Optional: <default>; context <parameter>; transform on items.
Not accepted: select; <parameter> with column or bind; <field>; <sql context>.

The statement must return exactly one row.

A.3.2 Filter query

```
<query name="..." module="..." select="FILTER">
  <sql> text </sql>
  [<default context="C" value="v"/>]
  [<parameter context="C" order="1"/>]
  <parameter column="COL" order="N" [type="STRING|INT"]/>
  <item name="key" column="COL" [default="v"] [transform="L:f"]/>
</query>
```

Mandatory: select="FILTER"; one <sql>; at least one <parameter> with column; one or more <item> with column (or built_in).
Optional: <default>; context <parameter>; default and transform on items.
Not accepted: <parameter> with bind; <field>; <sql context>.

A.3.3 Bind variable query

```
<query name="..." module="..." select="BIND">
  <sql> text with :var </sql>
  [<default context="C" value="v"/>]
  [<parameter context="C" order="1"/>]
  <parameter bind=":var" order="N" [type="STRING|INT"]/>
  <item name="key" column="COL" [transform="L:f"]/>
</query>
```

Mandatory: select="BIND"; one <sql>; at least one <parameter> with bind; one or more <item> with column (or built_in).
Optional: <default>; context <parameter>; transform on items.
Not accepted: <parameter> with column; <field>; <sql context>.

The statement must return exactly one row for each key.

A.3.4 Discovery query

```
<query name="..." module="..." discovery="yes">
  [<sql context="C"> text </sql>]
  <sql> text </sql>
  [<default context="C" value="v"/>]
  [<parameter context="C" order="N"/>]
  <item name="key" [transform="L:f"]/>
  <field name="MACRO" column="COL" | context="C"/>
</query>
```

Mandatory: discovery="yes"; one main <sql>; at least one <field> with column or context; one or more <item> with name only.
Optional: <sql context> (one per context iterated); <default>; context <parameter>; transform on items.
Not accepted: select; <parameter> with column or bind; column, built_in and default on items.

A.4 Webservice engine

A.4.1 REST query

```
<query name="..." module="..." type="REST" transport="GET|POST" encode="Json">
  <url>relative/path</url>
  [<transform language="L" function="f"/>]
  [<parameter urlparameter="name" order="N" [type="STRING|INT"]/>]
  <item name="key" path="a:b:[0]:c" type="INT|STRING" [default="v"]/>
</query>
```

Mandatory: type="REST"; transport; encode="Json"; one <url>; one or more <item> with path and type (or built_in).

Optional: <transform>; <parameter> with urlparameter; default and transform on items.

Not accepted: <sql>; <field>; <parameter> with column, bind or context.

A.4.2 REST filter query

```
<query name="..." module="..." type="REST" transport="GET|POST" encode="Json">
  <url>relative/path</url>
  <base path="list:[]">
  [<transform language="L" function="f"/>]
  <parameter path="field" order="N" [type="STRING|INT"]/>
  [<parameter urlparameter="name" order="M"/>]
  <item name="key" path="field" type="INT|STRING" [default="v"]/>
</query>
```

Mandatory: the same as a REST query, plus <base> and at least one <parameter> with path.

Optional: <transform>; <parameter> with urlparameter; default and transform on items.

Not accepted: <field>. Item paths are relative to the selected element.

A.4.3 REST discovery query

```
<query name="..." module="..." type="REST" transport="GET|POST" encode="Json"
  discovery="yes">
  <url>relative/path</url>
  [<transform language="L" function="f"/>]
  [<parameter urlparameter="name" order="N"/>]
  <item name="key" path="list:[]" [transform="L:f"]/>
  <field name="MACRO" path="field"/>
</query>
```

Mandatory: discovery="yes"; type="REST"; transport; encode="Json"; one <url>; one or more <item> with path (with [] or *); <field> with path.

Optional: <transform>; <parameter> with urlparameter; transform on items.

Not accepted: type, built_in and default on items.

A.4.4 SOAP query

```
<query name="..." module="..." type="SOAP" transport="POST">
  <url>service/path</url>
  <method uri="name:space" name="method" [action="action"]>
    [<argument name="arg" type="string|int" value="value"/>]
  </method>
  [<transform language="L" function="f"/>]
  <item name="key" path="a:b" type="INT|STRING" [default="v"]/>
</query>
```

Mandatory: type="SOAP"; transport="POST"; one <url>; one <method> with name; one or more <item> with path and type (or built_in).

Optional: <argument>; <transform>; default and transform on items.

Not accepted: discovery; <field>. <parameter> elements are accepted but have no effect in this version.

A.5 MQTT engine

A.5.1 Value reading

```
<query name="..." module="...">
  <parameter order="1" [type="STRING"]/>
  <item name="ts.mqtt.getvalue"/>
</query>
```

Mandatory: one <parameter> (the *topic*); the <item> ts.mqtt.getvalue.

Optional: —

Not accepted: attributes on the <item> other than name; <field>.

A.5.2 Topic discovery

```
<query name="..." module="..." discovery="yes">
  <parameter order="1" [type="STRING"]/>
  <item name="ts.mqtt.discovery"/>
  <field name="MACRO"/>
</query>
```

Mandatory: discovery="yes"; exactly one <parameter> (the pattern); the <item> ts.mqtt.discovery; exactly one <field>, with name only.

Optional: —

Not accepted: attributes on the <item> or <field> other than name.

Appendix B

Default values

Parameter	Value
-c (global file)	/etc/databbix/databbix.conf
-p (monitor directory)	/etc/databbix/dbs
General.QueryFile	/etc/databbix/query.xml
General.Engine	Database
General.RetryOnError	60
General.MaxCache	10
Zabbix.Refresh	300
Zabbix.MaxItemsPerPacket	100
Zabbix.PassivePort	0
Data.MaxErrors	5
Data.ConnLostOnDefaultCtxError	yes
Log.LogPath	/var/log/databbix
Log.LogLevel	3
Log.LogMaxSize	0
Log.LogMaxBackupCopies	4
Log.MaxQueryExecTimeLog	10
Webservice.Timeout / MQTT.Timeout	20000 ms
MQTT.Port	1883