ej Technologies

perfino Manual

Index

Introduction To Perfino

What is perfino?

perfino is a **monitoring tool for the JVM**. It is intended for **in-production** use and adds **extremely low overhead** to monitored applications. Its mode of operation is characterized as APM, short for "application performance management". Rather than collecting performance data at a low level and with a broad scope, perfino presents selected operations at a high semantic level, called "business transactions". In addition, scalar data is monitored from a variety of sources. Based on that data, threshold violations can result in alerts that help you safeguard the quality of service of your applications.

perfino is intended to run with your application at all times. This enables it to focus on **historic data**, showing you how performance characteristics evolve over long periods of time. Data is automatically made less granular as time goes by, so you can look back years into the past with only a slow rate of storage space consumption.

perfino is designed to **monitor multiple VMs** and trace the interactions between them. Whether you have a number of fixed VM installations or a cloud deployment with hundreds of VMs, perfino can monitor and organize them at the same time.

The perfino UI is a **web interface** that can be used by multiple users to analyze the collected data at the same time. A system of access levels allows you to partition a single server for multiple groups.

How do I continue?

This documentation is intended to be read in sequence, with later help topics building on the content of previous ones.

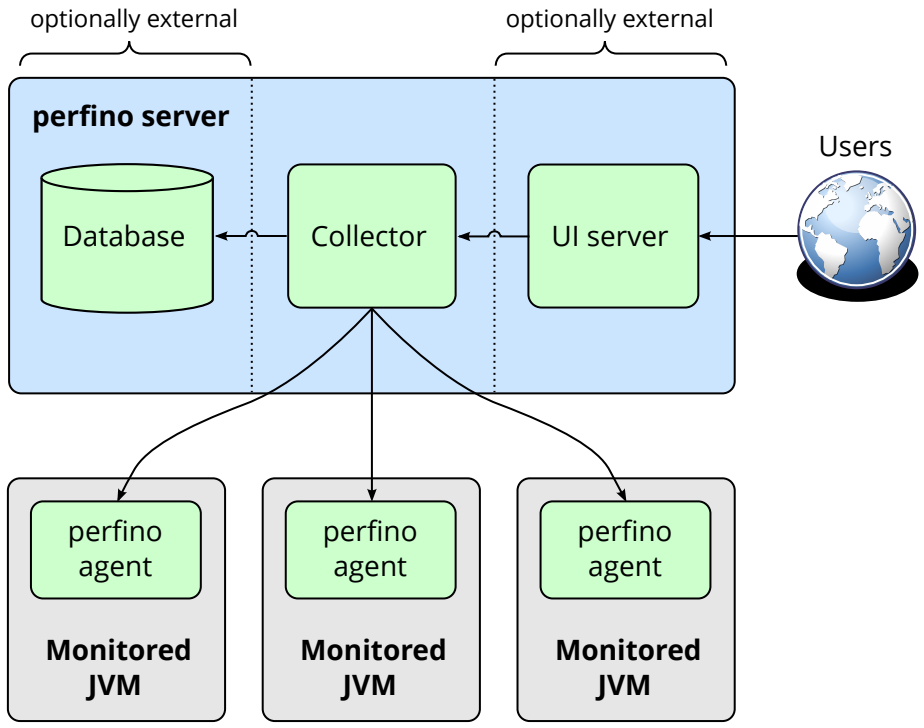
First, a broad overview over the architecture [p. 4] will help you to understand the components of perfino.

The help topics on installing perfino [p. 5] and monitoring your VMs [p. 10] will get you up and running.

Following that, the discussion of basic concepts [p. 14] and the overview of the UI [p. 18] take you to a level of understanding where you can explore perfino on your own.

Subsequent chapters build your expertise with respect to different functionality in perfino. The "Configurations" and "Advanced topics" sections are optional readings that should be consulted if you need certain features.

Perfino Architecture



perfino consists of two main parts: the **server** and the **agent**. The agent is loaded in the monitored VM and records data. The agent connects to a perfino server. The server periodically queries all connected agents and processes their data. Historical information is written to a database. Users log in with their web browser to the perfino server to analyze the recorded data.

Internally, the perfino server consists of three components:

- The **collector** accepts TCP connections from perfino agents in monitored VMs. These connections can be encrypted and authenticated, so they are viable for wide area networks. The collector also consolidates data in the database, fires triggers and generates alerts.
- The **embedded H2 database** stores all persistent data. There are two separate databases in that directory, one called "perfino" that contains recorded data and one called "config" that only contains configuration data. If you delete the "perfino" database while the perfino server is shut down, all configuration options are preserved.
- The **UI server** accepts HTTPS connections where users can log in, view and analyze the collected data and configure the server. Optionally, the UI server can be deployed as a WAR file into a separate JEE container like Apache Tomcat. It then uses RMI to talk to the collector. This is useful if the collector is located in an internal network and the perfino UI in a DMZ.

By default, the perfino server is the only process you need to run. In particular, you don't have to worry about installing and configuring an external database.

Installing Perfino

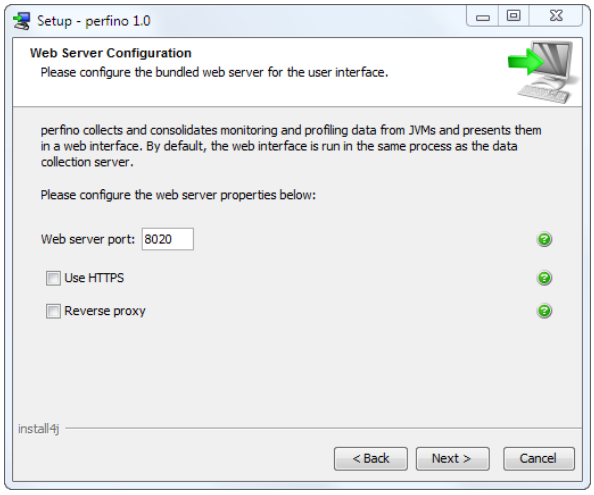
Installing perfino is done in two steps. First you run the installer. After perfino is running, you complete the post-installation setup in the browser.

Installer

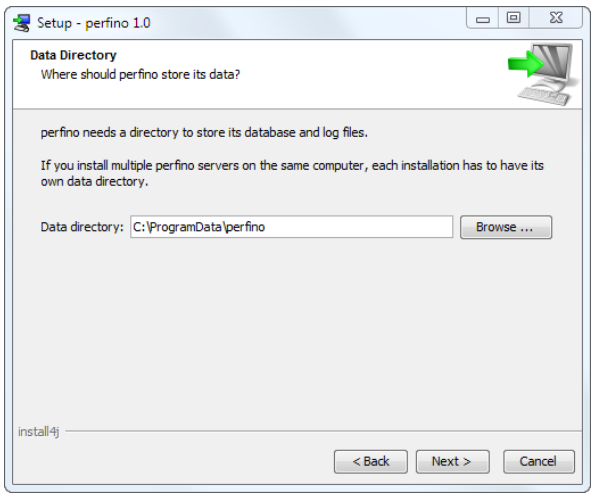
Installers are available for all major platforms. By default, the installer shows a GUI, but if you start the installer with the argument `-c`, it shows a **console interface**. This is required if you install perfino on a remote server with ssh.

For cloud-based deployments you may want to install and configure the perfino server without any user interaction. See the help topic on unattended installations [p. 103] for how to do that.

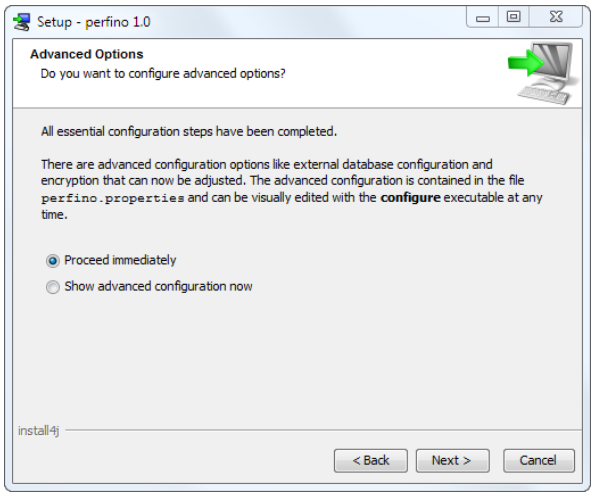
The installer asks you about all configuration options that are not configurable through the browser in the perfino UI. The most common setting that has to be adjusted is the the HTTPS/HTTP port for the web server.



Beside the installation directory, a perfino installation needs a **data directory** where all variable data is stored. This includes the database, log files and security certificates. By default, this directory is set to the program data directory on Windows (typically `C:\ProgramData`) and `/var/opt/perfino` on Unix and Mac OS X. In a scenario where you install multiple instances of perfino on the same machine, each perfino installation has to have its own data directory.

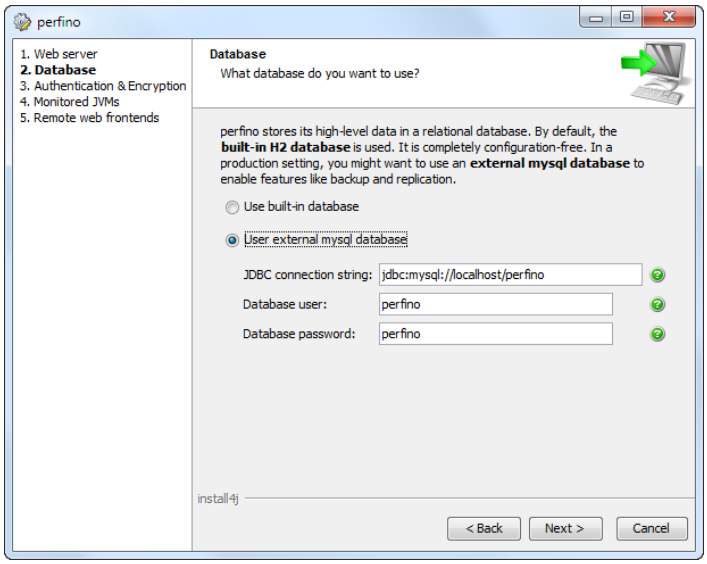


For evaluation purposes, this is all the installer needs to know. If you are deploying to a production environment, you might want to adjust other options, such as an external database.



Changing the initial configuration later on

The installer saves the configuration to the file `perfino.properties` in the installation directory. To change the configuration at a later time, you can edit that file or run the `configure[.exe]` tool in the installation directory. The configure tool requests elevated privileges on Windows and Mac OS X, so that it can overwrite the configuration file and restart the server.



The GUI of the configure tool presents the configuration options in the same way as the installer if you choose to configure advanced settings. Just like the installer, you can run the configure tool in console mode by calling it with the `-c` argument:

```
configure -c
```

The various options are briefly documented in perfino.properties with the same explanations shown in the GUI when you hover the mouse over the question mark icons. For more information, see the help on server configuration [p. 96] .

Post-installation setup

After the installer has finished, the perfino server is running and you can open your browser at

```
http[s]://localhost:[port]
```

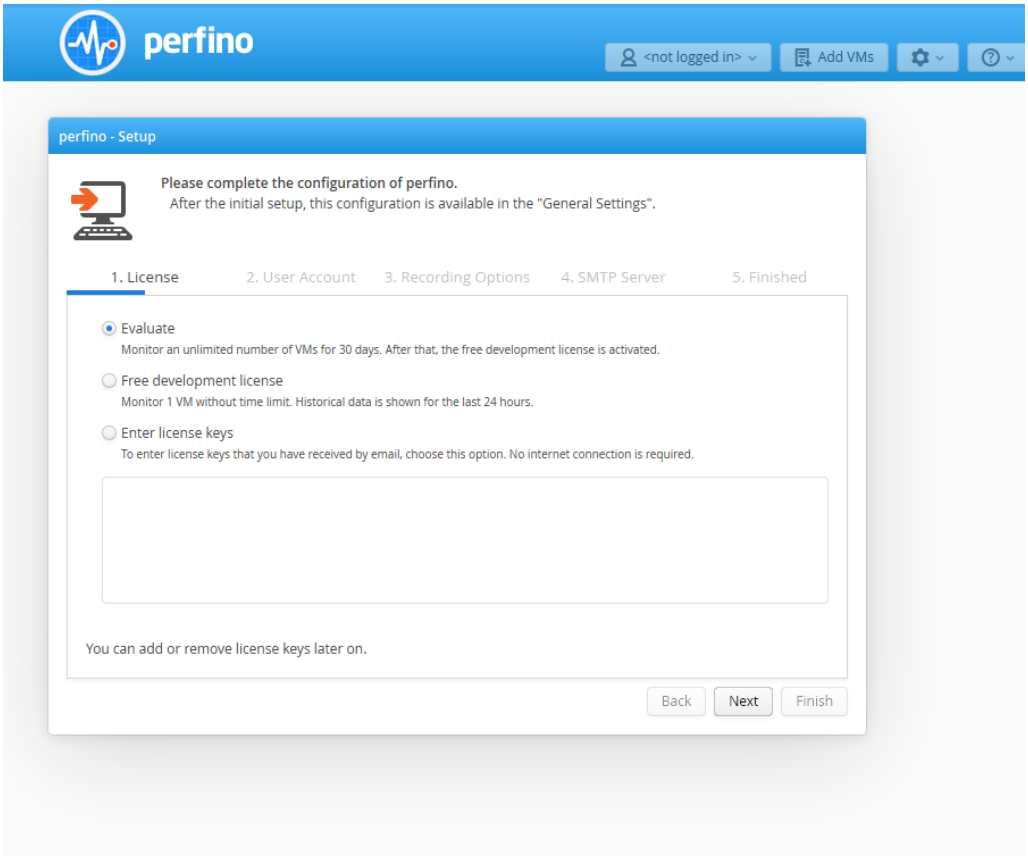
to connect to the perfino UI. Note that if you have selected HTTPS for the web server, there is no port that serves HTTP requests.

Before you can use perfino, you have to complete the installation wizard in the perfino UI. The first step of the wizard asks you about the license key.

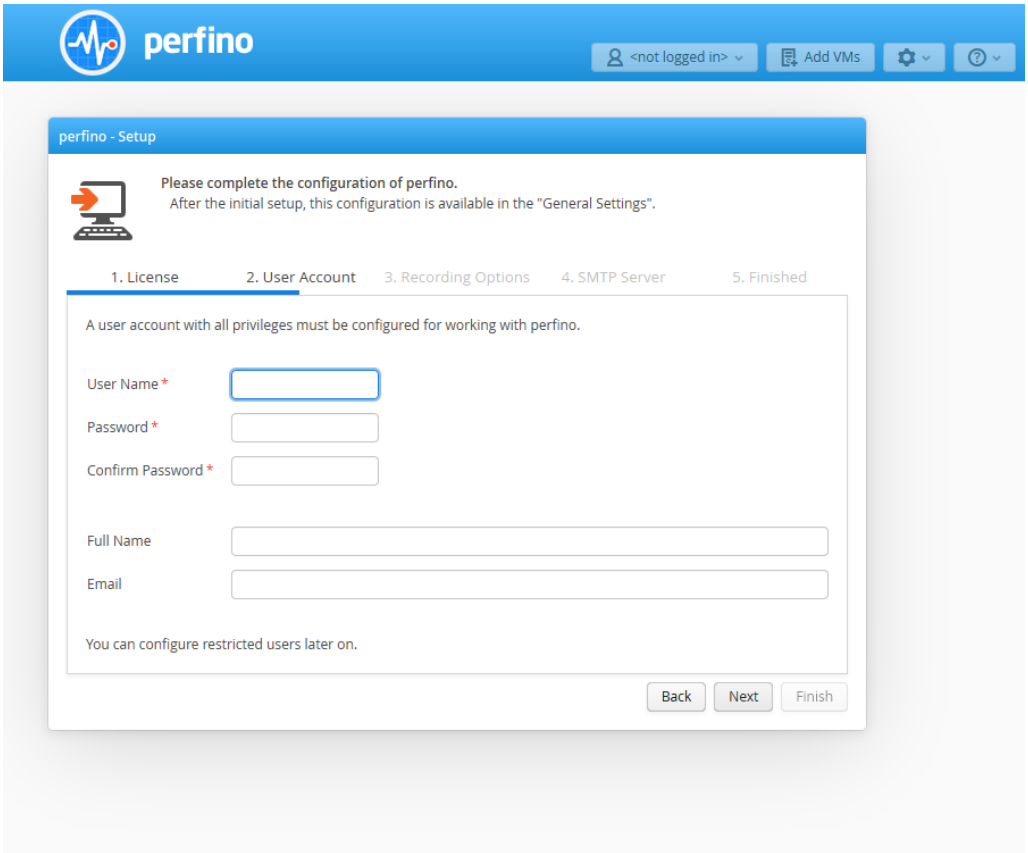
In the **free development mode**, you can monitor one VM without time restrictions. Historical data is only shown for the last 24 hours. This mode is intended for testing the perfino API during development.

With an **evaluation license**, an unlimited number of VMs can be monitored for a limited period of time. If you need more time to evaluate, contact sales@ej-technologies.com in order to get a new evaluation key. After the expiration of the evaluation key, the perfino server will continue to run, but JVM monitoring is suspended.

After purchasing a **permanent license**, you can monitor an unlimited number of VMs with no time restrictions. You can enter the permanent license key directly in the installer. After the installation, you can add a permanent license key on the "License Keys" tab of the "General settings".



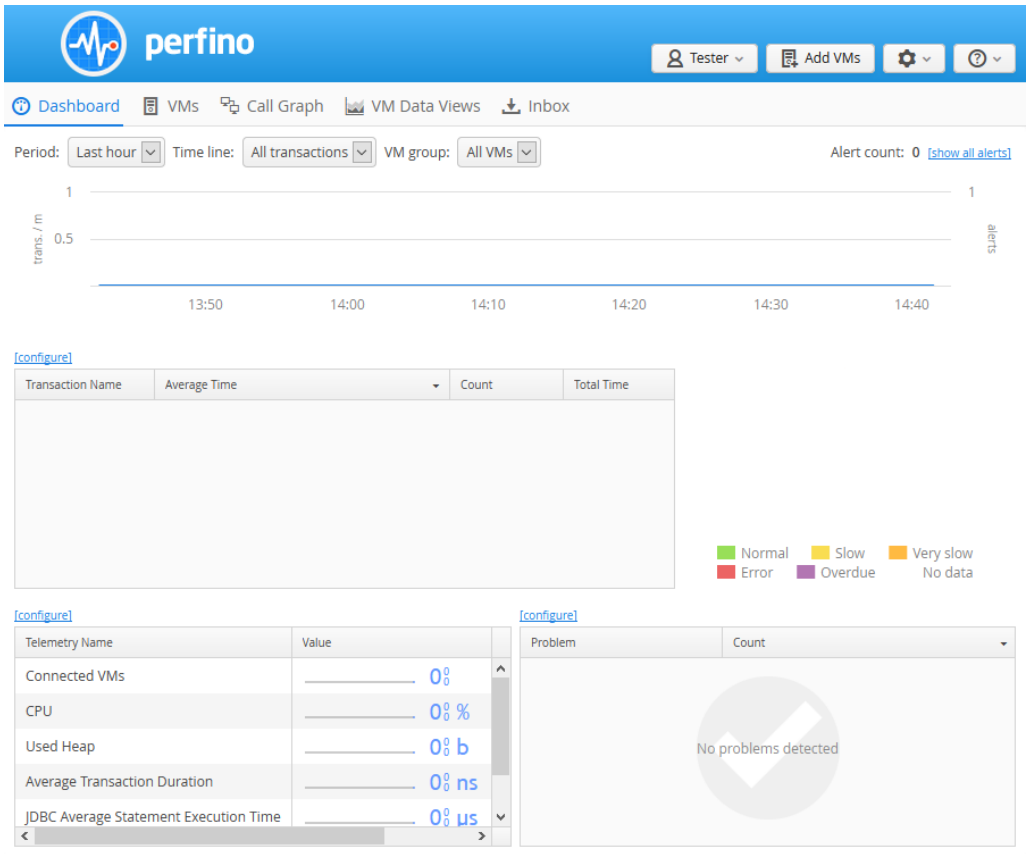
The second important step is the next one where you configure the **initial admin user**. You can configure other admin and non-admin users later on. The full name is displayed in the UI and the email can be entered so that an administrator can more easily identify or contact users.



In the "Recording Options" step you already get a chance to configure **VM groups**. However, it is not necessary to do so at this point.

The server can only send emails if it has a valid **SMTP server configuration**, so in the next step it asks you to provide this information. If you do not intend to send emails, you can skip this step.

When you complete the wizard, the server is fully initialized and the installation wizard cannot be shown again. However, all settings in the installation wizard can also be adjusted in the perfino UI. You are now presented with the empty dashboard and can continue to set up monitored JVMs as explained in the next chapter.


perfino
Tester
Add VMs
Dashboard
VMs
Call Graph
VM Data Views
Inbox
Period:
Last hour
Time line:
All transactions
VM group:
All VMs
Alert count: 0
[show all alerts]
1
0.5
trans. / m
alerts
13:50
14:00
14:10
14:20
14:30
14:40
[configure]
Transaction Name
Average Time
Count
Total Time
Normal
Slow
Very slow
Error
Overdue
No data
[configure]
Telemetry Name
Value
Connected VMs
CPU
Used Heap
Average Transaction Duration
JDBC Average Statement Execution Time
[configure]
Problem
Count
No problems detected

Monitoring JVMs

After perfino is installed, the agent can be added to JVMs on the local machine and on remote machines.

Basic mechanism

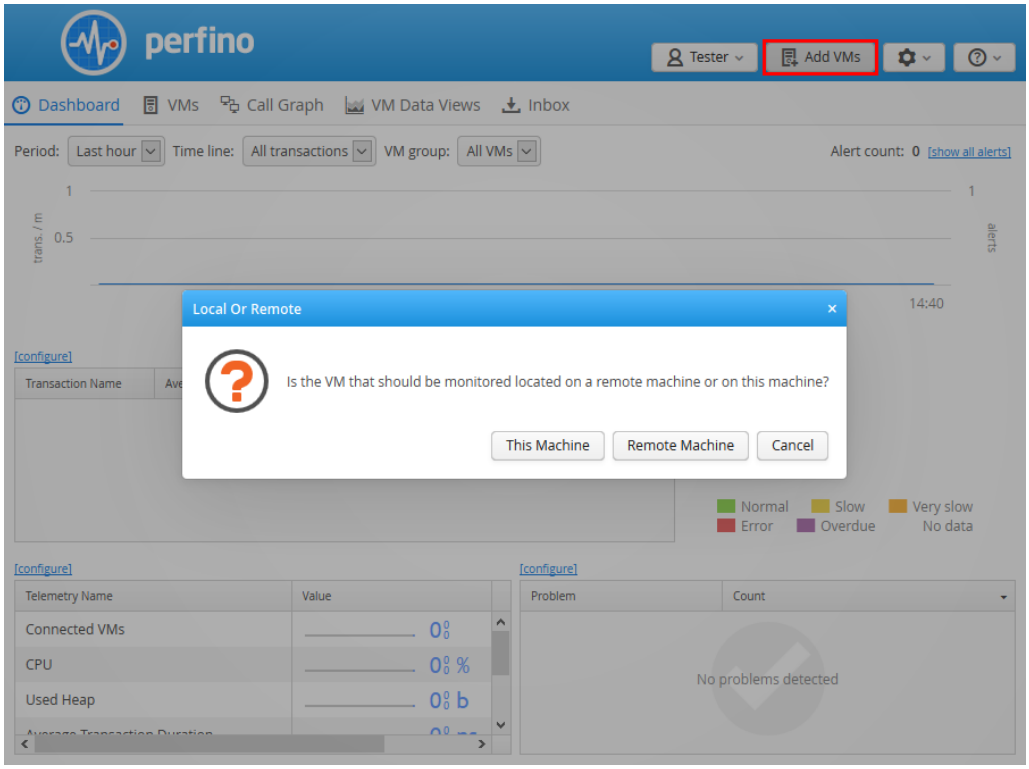
The perfino agent is a **Java agent**. It is loaded into a VM by specifying the `-javagent` VM parameter. Java agents are able to instrument classes as they are loaded and to retransform classes that have already been loaded.

To prepare a VM for monitoring, you need the perfino agent files and you have to know the actual VM parameter that needs to be inserted into the start script of your application server or your application. In a production deployment, the monitored VMs will usually be running on different machines. You do not have to install perfino on those machines, you just download the agent files from the perfino UI. Also, you will get instructions on how to construct the correct `-javaagent` VM parameter.

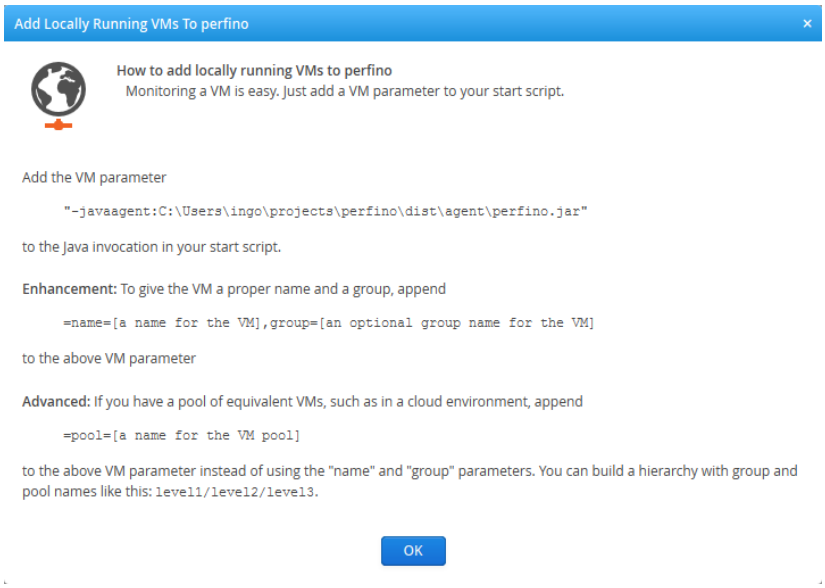
To get started, click on the "Add VMs" button in the header.

perfino server and monitored VM running on the local machine

If the perfino UI server is running locally, the perfino agent is already available in the perfino installation directory and you can easily monitor a locally running VM. This is a likely scenario if you are evaluating perfino. In that case, you will be asked whether the VM is running on the local machine or on a remote machine.



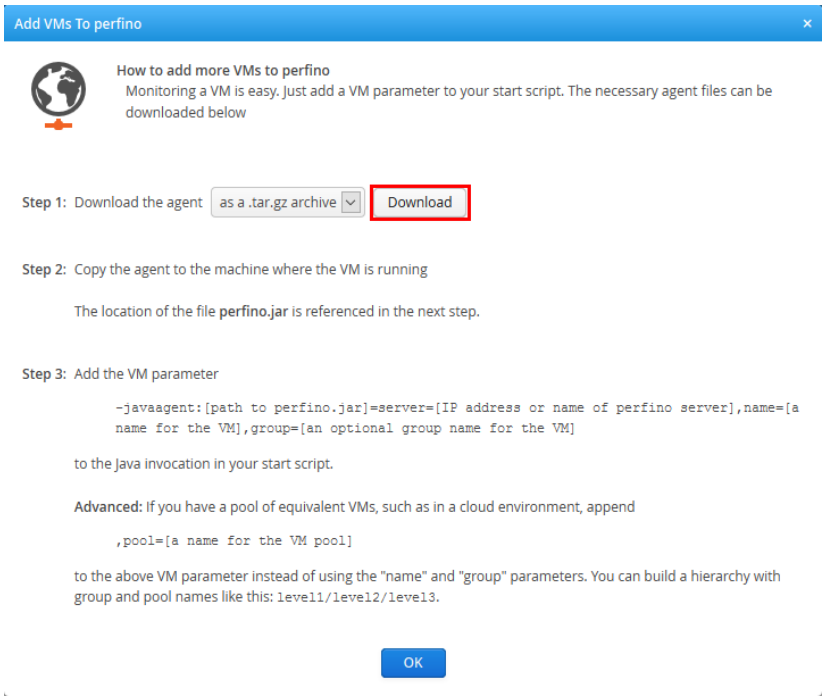
The "Add VMs" dialog displays the complete `-javaagent` VM parameter for the simplest case without any further configuration. It is suitable for monitoring a single VM.



Further instructions regarding group and pool names are explained below.

perfino server and monitored VM running on the different machines

In the general case, where the perfino server and the monitored VMs are running on different machines, you can **download the agent files** from the dialog:



perfino has an integration with JProfiler [p. 89] that allows you to perform full sampling of a monitored VM and open the resulting snapshots in JProfiler. The libraries that implement full sampling are native libraries, and are included in the archive.

Depending on the selected option, a .zip or a .tar.gz file will be saved after clicking the **[Download]** button. Extract that archive anywhere on the computer where the monitored VM is running. In

the top-level directory of that archive there is a file named `perfino.jar`, whose path will be referenced in the `-javaagent` VM parameter.

The VM parameter is not specified completely in the "Add VMs" dialog, since it depends on where you extract the archive and the name or IP address of the perfino server:

```
-javaagent:[path to perfino.jar]=server=[IP or host name]
```

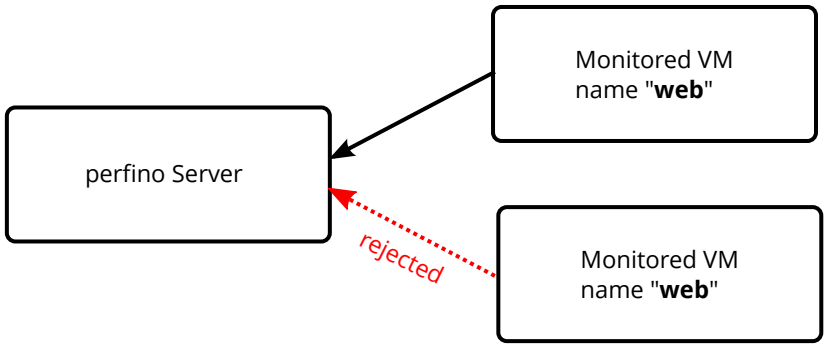
Without the server option, the agent will assume that the perfino server is running on localhost.

Naming VMs

If you monitor multiple VMs, you have to give them names in order to be able to **identify them in the perfino UI**. To assign a VM name, pass the `name` option to the `-javaagent` VM parameter:

```
-javaagent:[path to perfino.jar]=server=[IP/host],name=[VM name]
```

Each name can only be used by one VM at the same time. A second VM that requests to be monitored with the same name will be rejected by the perfino server. If you have a pool of VMs that cannot be assigned with unique names, see the section on VM pools below.



In addition to the VM name, you can group VMs into a hierarchy. By default, a VM is inserted into the top-level group. To assign it to another group, set the `group` option in the VM parameter:

```
-javaagent:[path to perfino.jar]=server=[IP/host],name=[VM name],group=[group name]
```

In the group name, separate hierarchy levels with forward slashes, as in `Web/Workers/Gen3`. A complete `-javaagent` parameter looks like this:

```
-javaagent:/opt/perfino/perfino.jar=server=192.198.0.33,name=web,group=Web/Workers/Gen3
```

VM names must be unique within the same group.

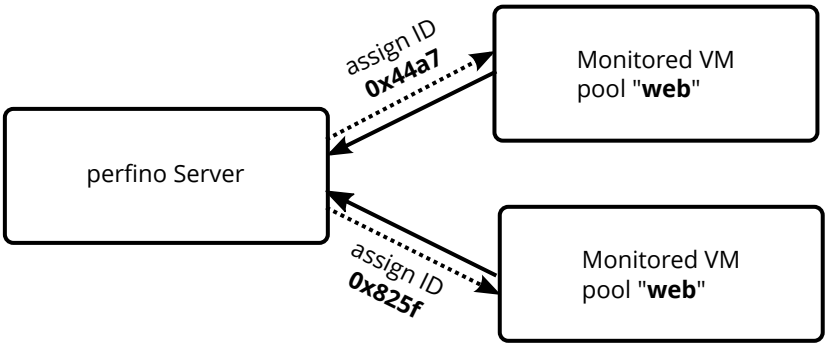
VM pools

Sometimes it is not possible to configure VMs with unique names, for example in a cloud environment where **instances are provisioned dynamically**. In that case, you can assign the VM to a VM pool:

```
-javaagent:/opt/perfino/perfino.jar=server=192.198.0.33,pool=Web/Workers
```

Like the `group` option, the `pool` option takes a hierarchical name with forward slashes as the hierarchy separator. If `pool` is specified, you cannot specify `name` or `group` and vice versa.

In a pool, a monitored VM is given a name with a unique identifier. When the VM detaches, this identifier will never be used again. Unlike for named VMs, the history of a single VM is limited to the connection time of that VM, so there is no associated long term history. However, you can follow the history of the entire pool to analyze trends and make historical comparisons.



Server port

By default, a perfino server listens for VM connections on port **8847**. This is the port that needs to be **opened in firewalls** to allow monitored VMs to reach the perfino server.

A perfino server may be configured to use a different port by adjusting the "vmPort" property [p. 96] in `perfino.properties`. This is necessary if the port is already in use or if you install multiple instances of perfino on the same machine.

To tell the agent about such a non-default port, you have to add the `port` option to the `-javagent` VM parameter. For example:

```
-javaagent:/opt/perfino/perfino.jar=server=192.198.0.33,port=8912,name=test
```

Encryption and authentication

Mutual authentication and encryption are enabled by the agent keystore file `agent.ks` in the same directory as the `perfino.jar` file. The agent keystore is generated by the server and is located in the `ssl` directory below the perfino data directory.

When the "vmUseSsl" property in `perfino.properties` is set to `true`, the agent keystore file is automatically added to the agent files that you download from the perfino UI. In that case, encryption and authentication work out of the box.

If you switch encryption and authentication on or off after you have set up your monitored VMs, you have to make the following changes manually:

- **Switching encryption on**

  Locate `[perfino data directory]/ssl/agent.ks` and copy it to the agent installations on all machines where VMs are monitored. The keystore file has to be copied to the same directory as the file `perfino.jar`. If the correct agent keystore is not present, the server will refuse the connection from the agent.
- **Switching encryption off**

  Delete the file `agent.ks` next to the file `perfino.jar` in the agent installations on all machines where VMs are monitored. If that file is present and the server does not use encryption, the agent will refuse to connect to it.

If you would like to keep the agent keystore file in a different path, add the `keystore` option to the `-javagent` VM parameter. For example:

```
-javaagent:/opt/perfino/perfino.jar=server=192.198.0.33,keystore=/sec/agent.ks,name=test
```

Basic Concepts

perfino collects data of two fundamentally different types: transactions and telemetries. Policies and thresholds are used to detect anomalous conditions while triggers take action if something is out of order.

Transactions

In perfino, you analyze your business processes with transactions [p. 25] . At a technical level, a transaction is simply a method invocation. To measure a transaction, perfino records its timing and constructs a transaction name that describes the business process.

The **transaction naming** has a significant impact on what you will see in the perfino UI.

- It enables you to understand what triggered the transaction.
- It groups all business processes with the same transaction name and so determines the granularity that is used to measure business processes.
- It can serve as a basis to filter out unwanted operations.

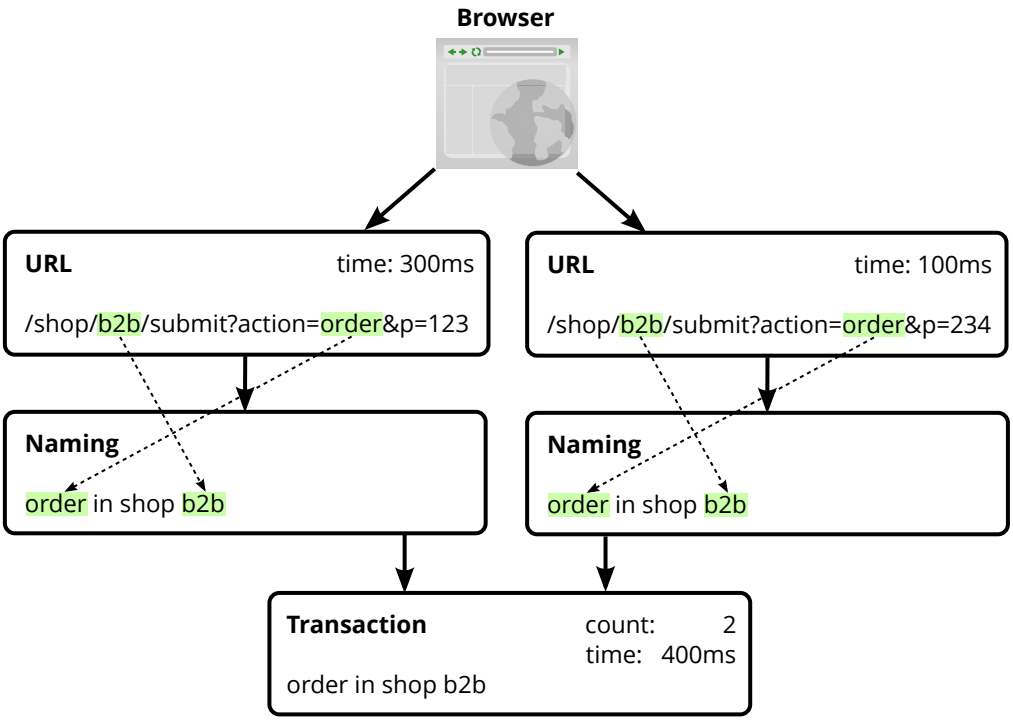
perfino cannot know what your business processes are, so configuring transactions is an important part in setting up an application for monitoring. Some frameworks are high-level by nature and so perfino can offer them as transaction types that can be configured with a minimal amount of work.

The most common example of a transaction is the invocation of a URL that is handled by your application server. In the default configuration, perfino intercepts the method that handles HTTP calls and constructs a transaction name including the first three segments of the URL. This is an arbitrary naming strategy that is just intended to get started. In your application, only the first segment of the URL may be relevant for the business process, or you may need a particular query parameter in the name.

Also, you will probably not want all URL invocations to become transactions. Many HTTP requests are for static resources, and those are not interesting in terms of business processes. In perfino, you can **discard transactions** based on the name that would be associated with a transaction. If you generate too many different transaction names, perfino's overload protection [p. 106] is activated.

The following figure shows how different URLs end up as the same transaction based on a transaction naming that

- adds the value of the query parameter "action"
- adds the fixed text "in shop"
- adds the second segment of the URL



Policies

Transactions have associated policies. The policies determine

- the acceptable timing for a transaction
- the way errors are detected and handled
- when to perform method-level sampling

For each violated condition in the policies, you can see transaction details separately in the perfino UI. For example, you can inspect slow transactions or transactions that resulted in an error separately and not cumulated with other regular transactions of the same name.

perfino gets information from the monitored application by instrumenting methods. To keep the overhead low, very few methods are instrumented. In order to get more detailed information in the case of a very slow transaction, the policy can start method level sampling [p. 44] for a transaction once it is clear that it is taking too long.

With sampling, you get a cumulated call tree and hot spots on the method level that show you where the time is actually spent.

Telemetries

The other fundamental type of data source in perfino is the periodic sampling of scalar values, like heap size or thread count. Each telemetry [p. 50] can be plotted as a time-resolved graph. In perfino, telemetries are often shown as sparklines, without defined axes and with a trailing current value.

There are many standard telemetries in perfino that collect their data from well-known subsystems of the JVM or popular databases and frameworks. In addition, integer values that exposed by

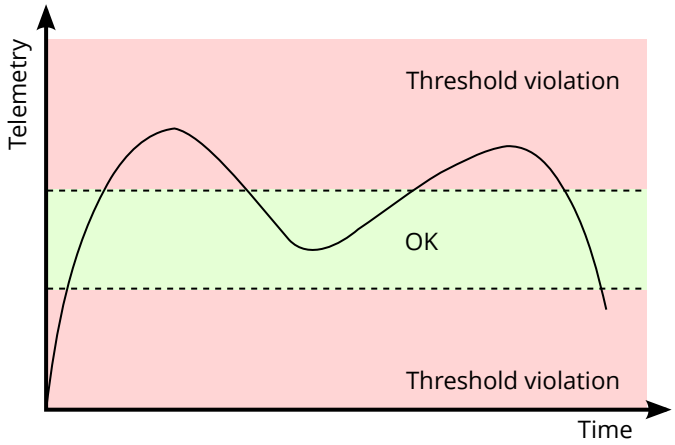
an MBean can be monitored by perfino. On a programmatic level, you can use the `@Telemetry` annotation to define custom telemetries on static methods with a numeric return value.

Thresholds

You will have different expectations with respect to different telemetries. For example, the heap usage often oscillates around a base line and where a steady increase is a sign of a bug in the application.

Or, the average duration of JDBC statements usually varies with server load and is an indicator of the health of the application.

To detect anomalous conditions, you define thresholds [p. 57] with an optional lower and an optional upper bound. Threshold violations are counted on a per-VM basis or for each VM group. They do not have actions associated with them. Often, you will not want to take any action for single threshold violations, but only for a cascade of such conditions.



Triggers and alerts

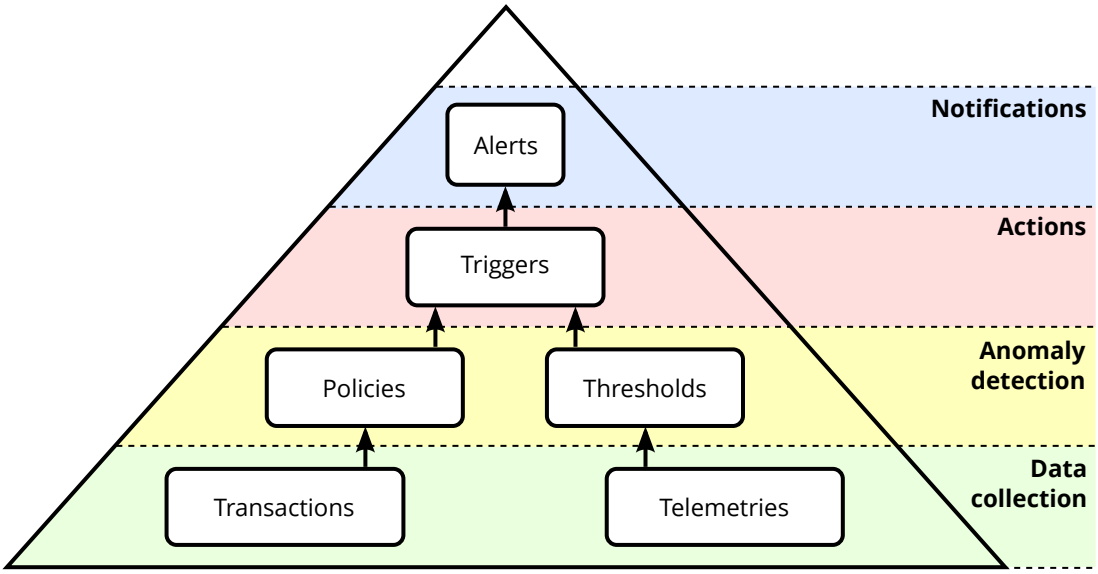
Both transactions and telemetries can lead to anomalous conditions: A transaction policy can identify a slow transaction and a telemetry threshold can be violated.

In order to take action on these conditions, you use triggers [p. 60] . Triggers do not operate on a per-VM level, they process all recursively contained VMs in a VM group. Each VM group in a hierarchy of groups has its separate triggers.

For example, you could define a trigger for all VMs that fires when the number of connected VMs falls below 20. In the same VM hierarchy, you might have a group that only contains database VMs. In that group, you might want a separate trigger that fires when the number of connected database VMs falls below 3.

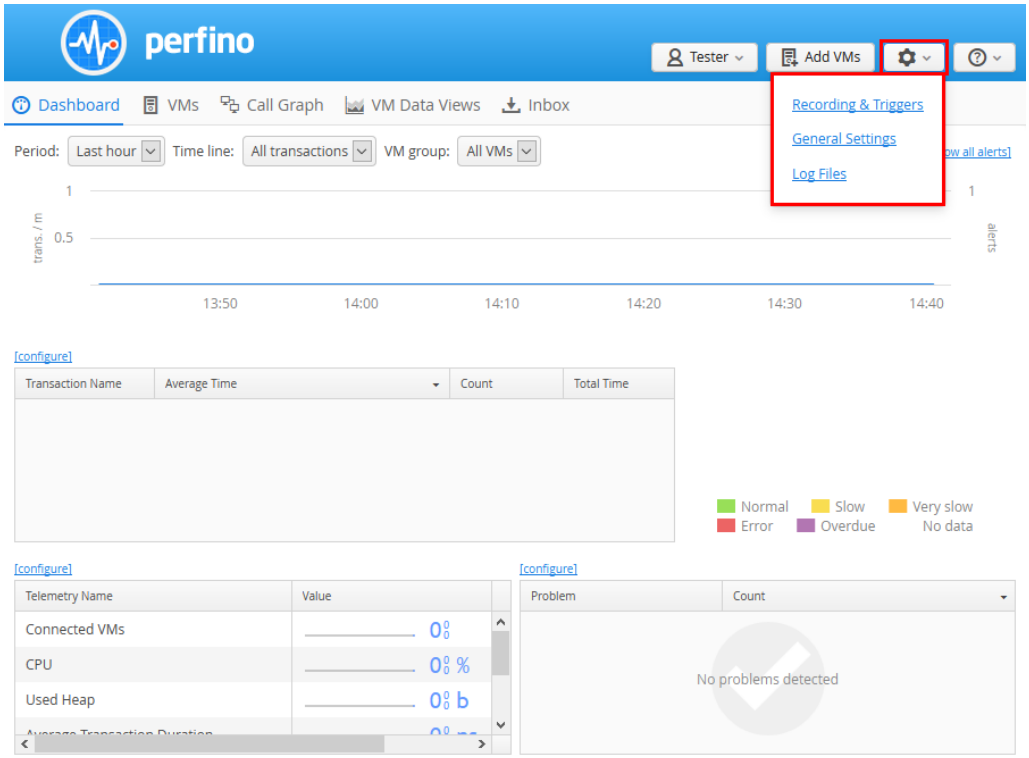
When a trigger fires, it executes its actions. Actions can start data collection, such as full VM sampling, send emails or create alerts.

Alerts [p. 65] are shown in the dashboard and are the highest level in perfino's pyramid of concepts:


Notifications
Alerts
Actions
Triggers
Anomaly detection
Policies
Thresholds
Data collection
Transactions
Telemetries

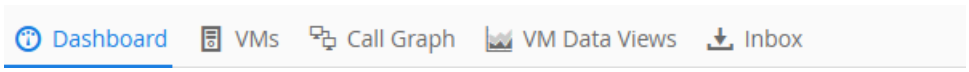
UI

The perfino UI separates data and configuration. When you log in, you are in the **data perspective**. The configuration drop-down in the top right corner gives you access to the **configuration**.

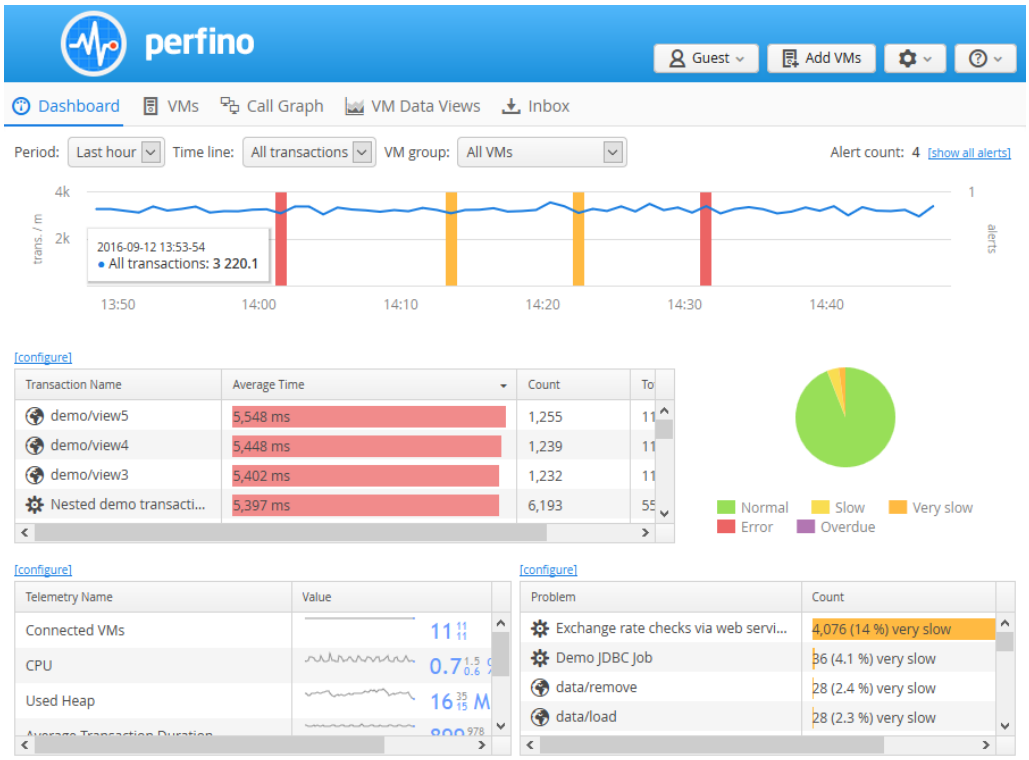


The data perspective

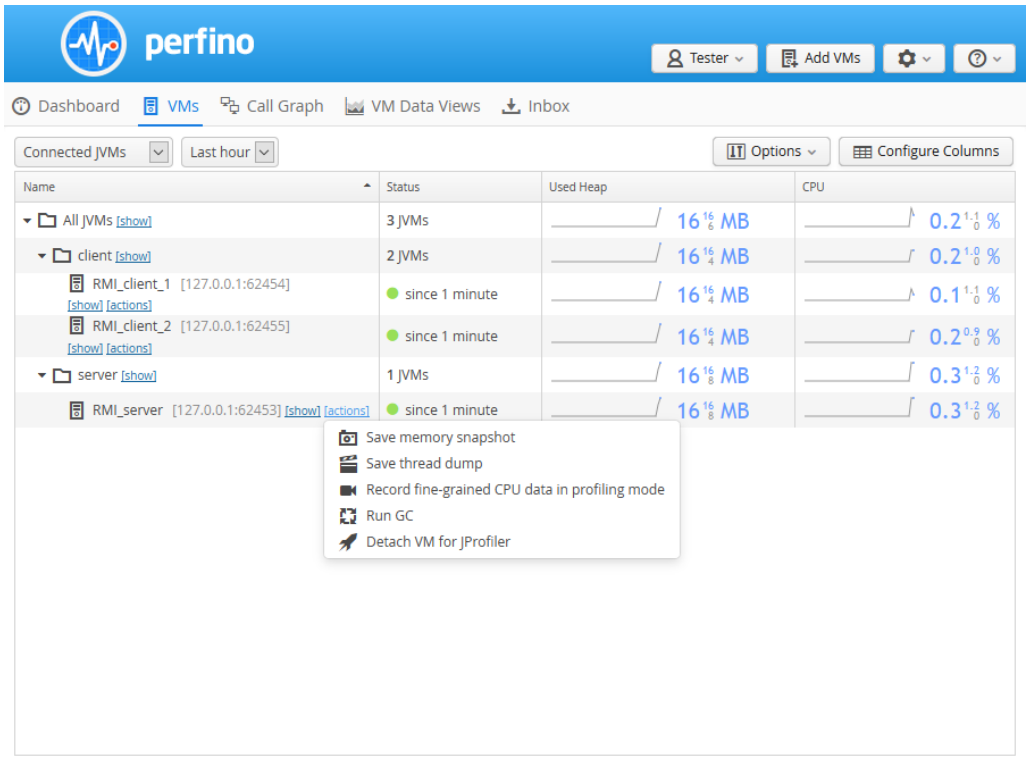
The data perspective is divided into several tabs. The tabs are ordered from more general information on the left to more specific information on the right.



After logging in, you see the **dashboard** where important information about the activity and the health of the monitored VMs is presented on a single screen. From the dashboard, you can drill-down into the more detailed "VM data views" by clicking on the items of interest, including transaction names, problems, sparklines and slices of the policy violation pie chart.

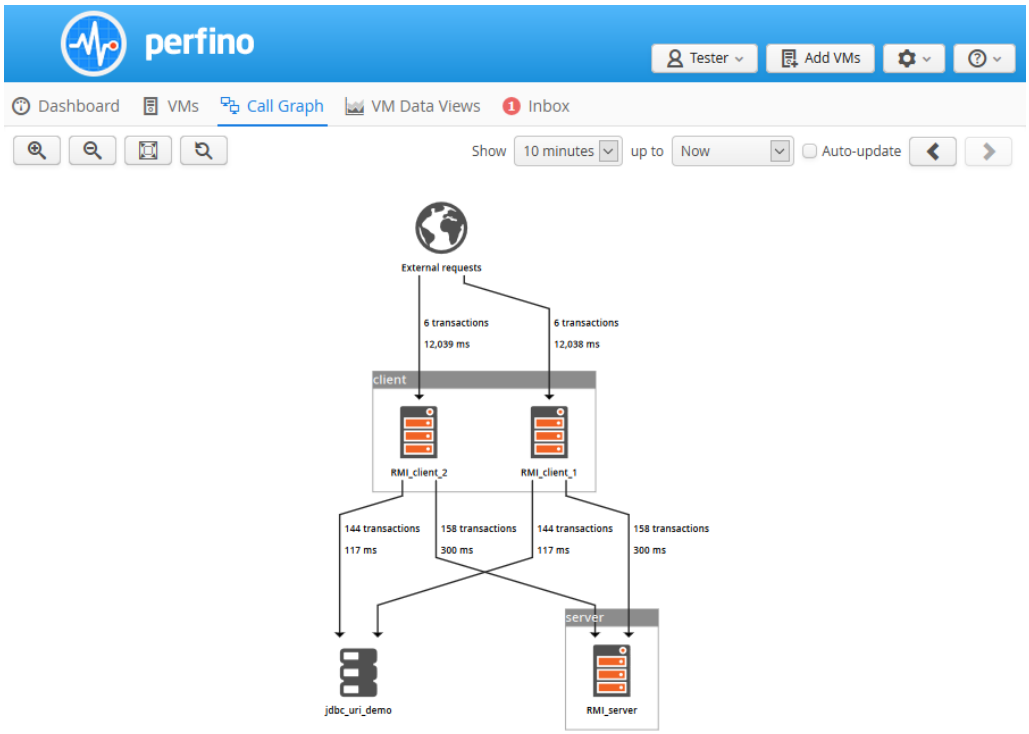


The **VMs** view shows you the hierarchy of all connected VMs, together with selected information in the form of sparklines.

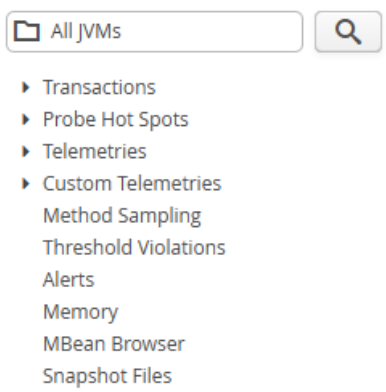


The "show" context action takes you to the "VM data views" for the selected group or VM and provides quick access to all recorded data that is available for your selection. Like in the dashboard, clicking on the sparklines takes you to the full telemetry views in the "VM data views".

The **call graph** focuses on remote calls between monitored VMs. Each VM is a node and remote calls like EJB, RMI and web service invocations are edges in the graph. Clicking on nodes and edges shows further information.

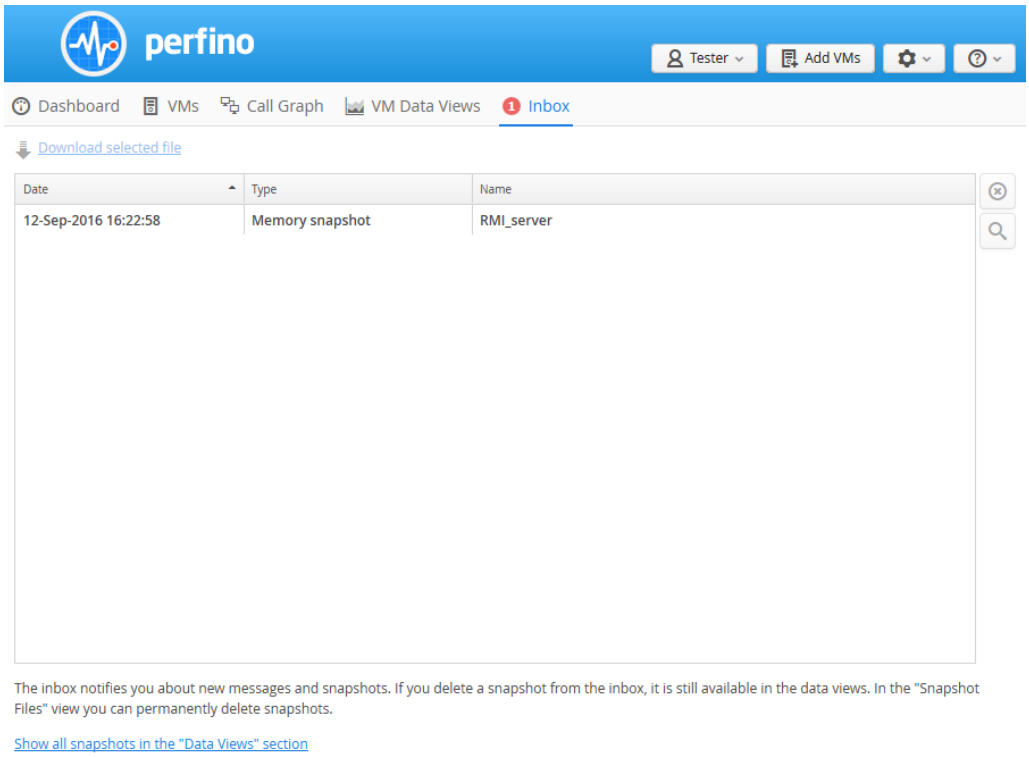


The **VM data views** tab holds all views that show data for a single selected VM or a group of VMs. The actual VM selection and the desired view can be adjusted in the view selector.



The **inbox** view shows all messages that have been sent to you. For example, if you start full-VM sampling on a particular VM, a notice will be sent to your inbox once the snapshot is ready for downloading.

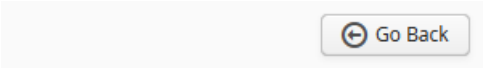
If there are unread messages in your inbox, you will see a red notification in the tab area.



Configuration

Configuration is divided into two parts: **general settings** and **recording options**. If you want to view and edit general settings, you have to be an administrator. Recording options can also be edited by the "profiler", but not by the "viewer" access level.

Both general settings, as well as recording options are modal panels. If you did not change anything, you have to click on the "Go Back" button to return to the data perspective.

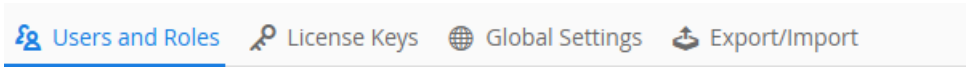


If you made a change, the "Go Back" button is replaced by **[Apply Changes]** and **[Discard Changes]** buttons. No changes to the configuration have any effect before this confirmation.

General Settings

In the general settings you administrate the perfino server. This includes

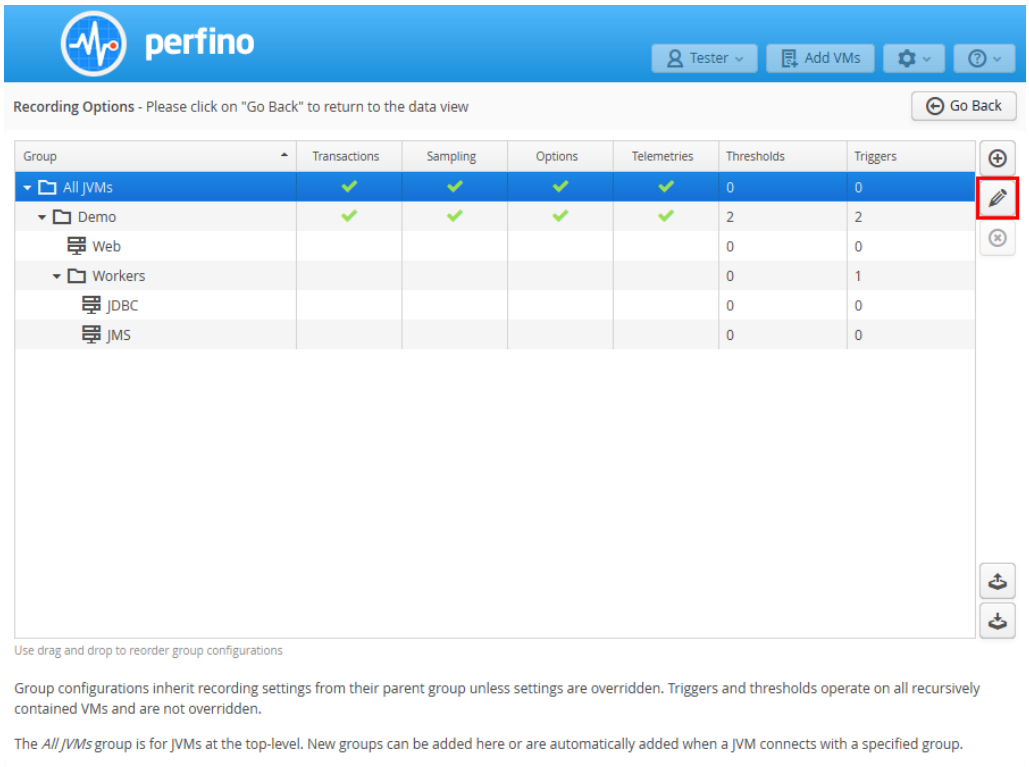
- the users and their access levels
- the license keys
- data consolidation options
- SMTP access



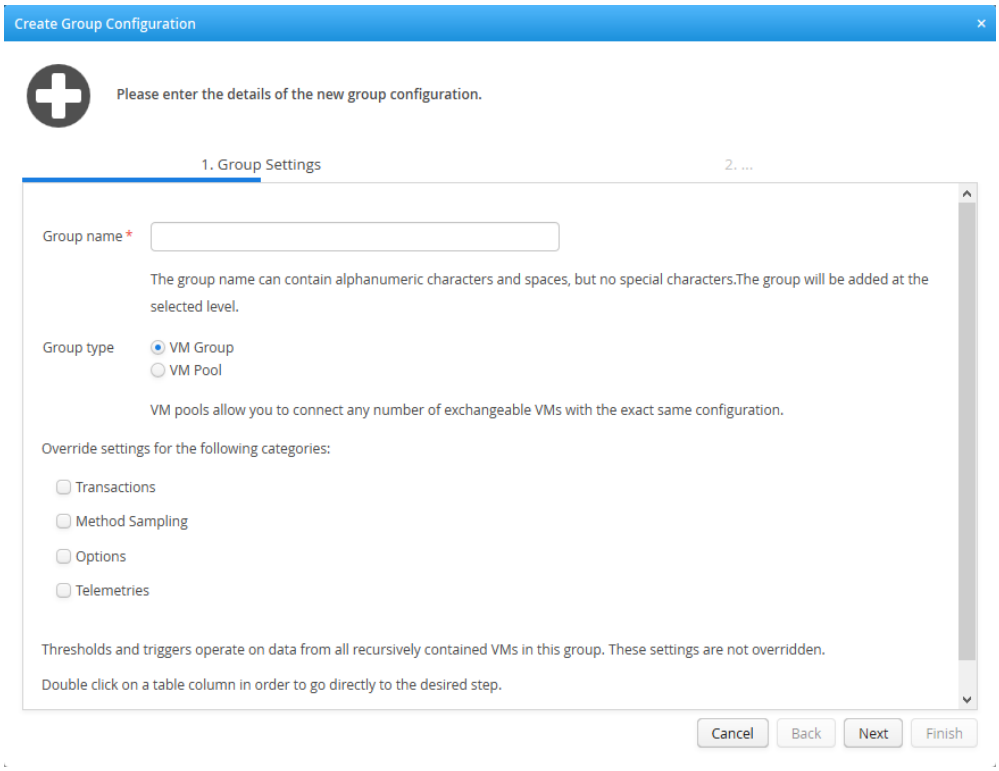
In addition, you can export and import the entire server configuration.

Recording & triggers

Recording options and triggers are configured for each VM group. You can click the edit button or double-click on any column to jump directly to the desired step.



The configuration contains recording options as well as threshold and trigger settings. Recording options are sent to the monitoring agents in each connected VM. They are inherited to nested VM groups and nested VM groups can override settings in their parent groups.

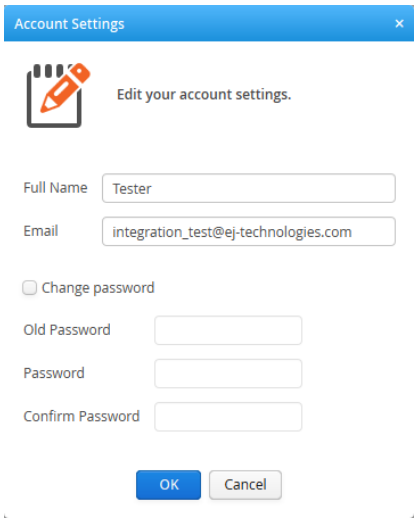


Thresholds and triggers, on the other hand are handled in the perfino server and are processed for the VM group for which they have been defined. They operate on all VMs that are contained recursively in their VM group. If you need additional thresholds or triggers that operate only on a nested VM group, you can define them on that group.

User settings

The user drop-down in the header shows selected information about your account and the current session and contains the **[Logout]** button that terminates your perfino session.

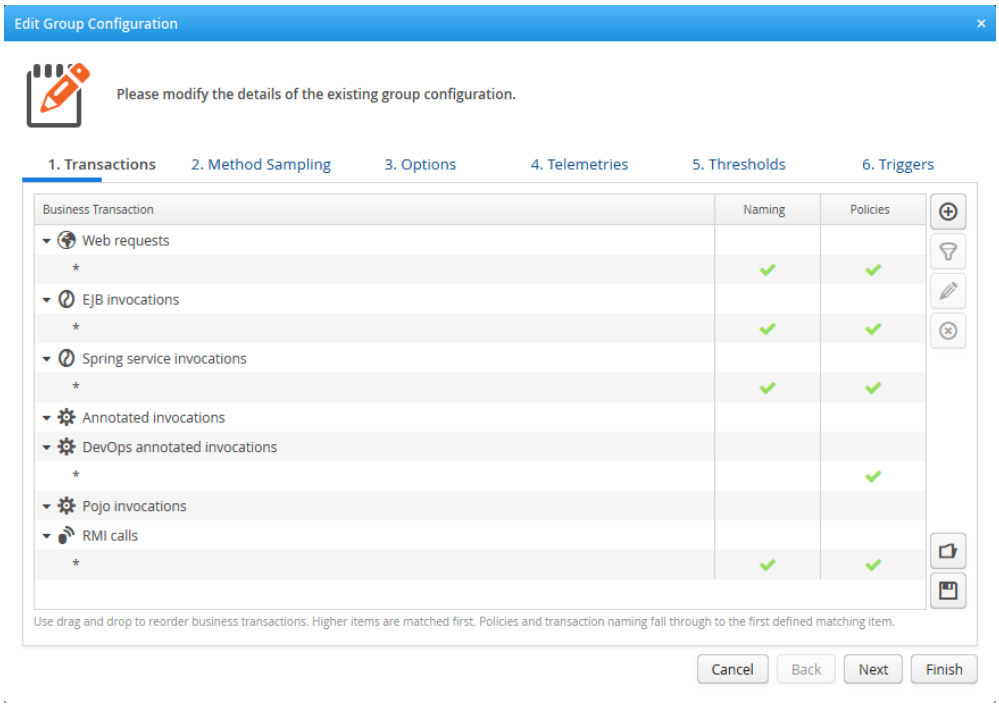
User settings, like your name and password can be adjusted by following the "Account Settings" link. If your access level is "profiler", this is where you can see the VM groups that you are allowed to modify.


Account Settings
Edit your account settings.
Full Name
Tester
Email
integration_test@ej-technologies.com
Change password
Old Password
Password
Confirm Password
OK
Cancel

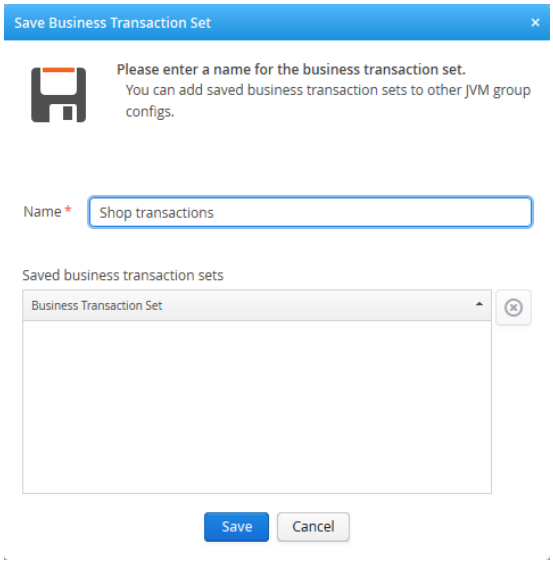
Transactions

Configuration

Transactions are configured in the recording settings. When you edit a VM group or a VM pool, the first displayed step in the wizard shows the transactions settings.

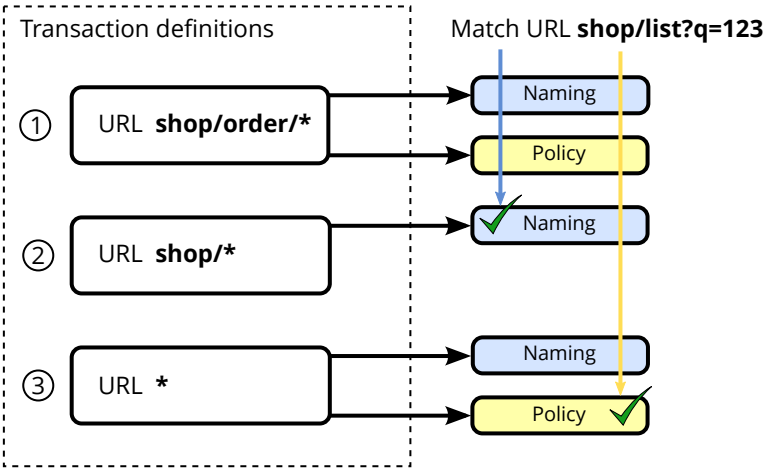


If no transactions are defined for a nested VM group, VMs get their configuration from the nearest ancestor group where transactions are defined. The "All VMs" configuration always has associated transaction definitions and is always a successful last stop for this search in the hierarchy.



When a nested group overrides transaction settings, it overrides all of them and there is no merging with the settings of a parent group. If you want to reuse selected definitions from a parent group, save them to a set and include them in the configuration of the nested group.

A transaction definition selects a segment of all possible transactions in its subsystem. For example, web transactions are defined with a URL wildcard or regular expression. Each transaction definition has an optional associated naming scheme and optional associated policies. For both naming and policies, perfino keeps looking in the list of defined transactions until it finds a matching entry.



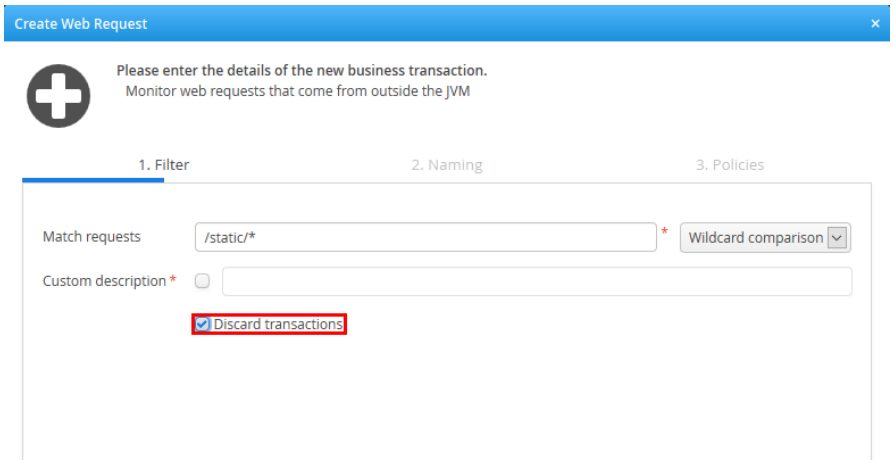
This allows you to define a series of transaction definitions that all have different naming schemes while keeping common policy definitions for a group of transaction definitions. Keep in mind that the transaction matching in perfino takes the first match, so more generic filters should be lower in the list of transaction definitions.

There are a number different transaction types in perfino. All types fall into two different categories: **predefined** transaction types and **custom** transaction types.

Predefined transaction types

The most common mechanism that starts a business transaction is via a URL invocation. In perfino, this is called a **web transaction**. In the default configuration, perfino intercepts all URLs and creates transactions for them. For your application, you will want to limit the intercepted URLs to those that are meaningful from a business perspective. To do that, you can remove the transaction definition for the URL wildcard "*" and add your own specific transaction definitions.

Another way to limit transaction matching is to use the "Discard" flag. In that case, any matching URLs will be excluded and no further matching will take place.



In the list of transaction definitions, the filter expression will be shown as the name of the transaction definition. Often, this is not descriptive enough. Use the "Custom description" check box shown above to enter a meaningful name. This is only used in the configuration and not in the displayed data where the transaction naming determines the name of the transaction.

The name of a transaction is composed of a chain of naming elements. The list of available naming elements depends on the transaction type. For web transactions you can use specific elements like "URL query parameter" and "URL segments".

An important consideration when defining complex transactions is what kind of nested transactions should be allowed. Perfino does not record a directly nested transaction that has the same name. More restrictive options are to prevent nested transactions

- **that match this entry**

  No nested transactions will be created if the nested transaction would originate from this transaction definition.
- **with the same group name**

  If you select this option, you have to configure a group name for this transaction definition. For example, you might have different entry points for a particular business transaction from a web transaction and from an EJB transaction and the URL handler always calls the EJB. If you do not want to see the EJB as a nested transaction, you can assign the same group name to the web transaction and the EJB transaction.
- **of the same transaction type**

  This option is suitable if you only want to see entry points and not the internal structure. For example, EJBs will often call other EJBs. This reentry mode prevents a tree of nested transactions in that case.
- **all further entries**

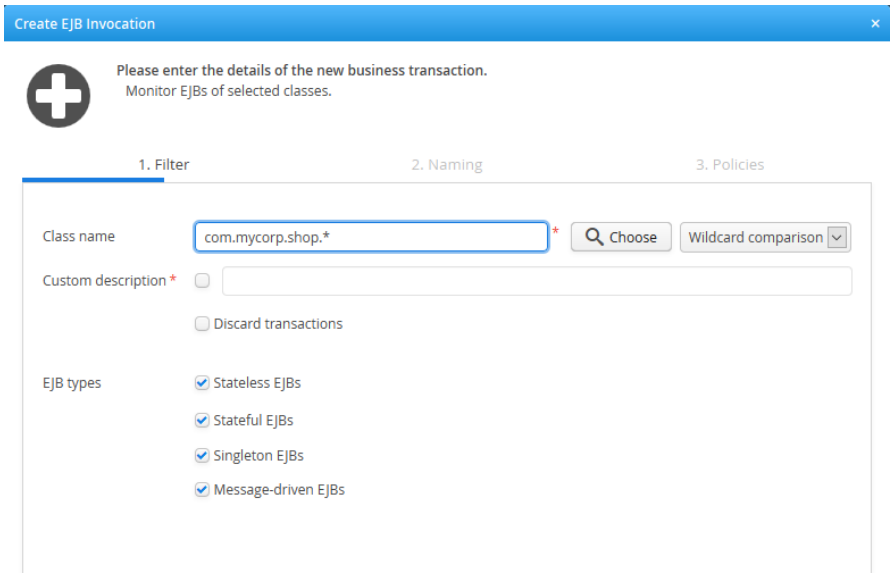
  Use this option to prevent any kind of nested transaction if a transaction is created from the current transaction definition.

The reentry inhibition only applies to the transactions that would be directly nested. If a nested transaction is allowed, its own reentry inhibition setting will be used for the next nesting level.

EJB transactions are created from calls into public methods of an EJB that is annotated with `@Stateful`, `@Stateless`, `@MessageDriven` or `@Singleton`. **Spring service invocations** are for calls into beans that are annotated with `@Component`, `@Controller`, `@Repository` or

`@Service`. You can limit transaction recording to a sub-set of those transactions for both EJB and spring transactions.

When creating transaction definitions, both these transaction types can be filtered with respect to class or package name. The discard mechanism is useful to exclude certain classes that are not generating significant data. By default, perfino shows all EJB and Spring service invocations.



RMI transactions handle incoming RMI calls into a VM. Like for EJB and Spring transactions, you can use a class filter to include or exclude implementation classes. The default configuration in perfino shows all RMI invocations.

Custom transaction types

There are many frameworks where specific interceptions can capture all important business transactions. Also, your own code will often be structured in a such a way that a small set of methods will map to your business transactions. perfino offers three mechanism for monitoring these cases. Configuration of these transaction types is more elaborate so that each of them has its own chapter in the advanced topics.

For an **annotated invocation** transaction, you specify a particular annotation class name. In the further configuration [p. 107] you can decide whether the annotation should apply to classes or methods and how inherited classes should be handled.

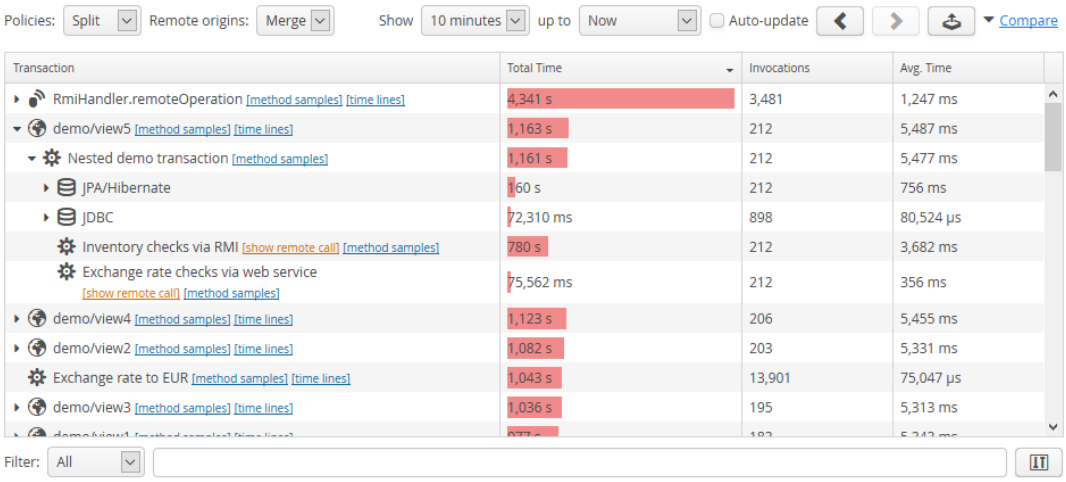
DevOps transactions are defined in your code with annotations supplied by perfino. While the naming is completely specified with those annotations, the policy must be configured in the perfino UI. This is the most maintainable way to define complex transactions. See the advanced topic about DevOps transactions [p. 112] on how to add them to your project.

If you cannot modify the code that should be monitored, **POJO transactions** allow you to specify all the same interceptions as DevOps transactions directly in the perfino UI. See the detailed explanation of POJO transactions [p. 109] for more information.

Call tree and hot spots

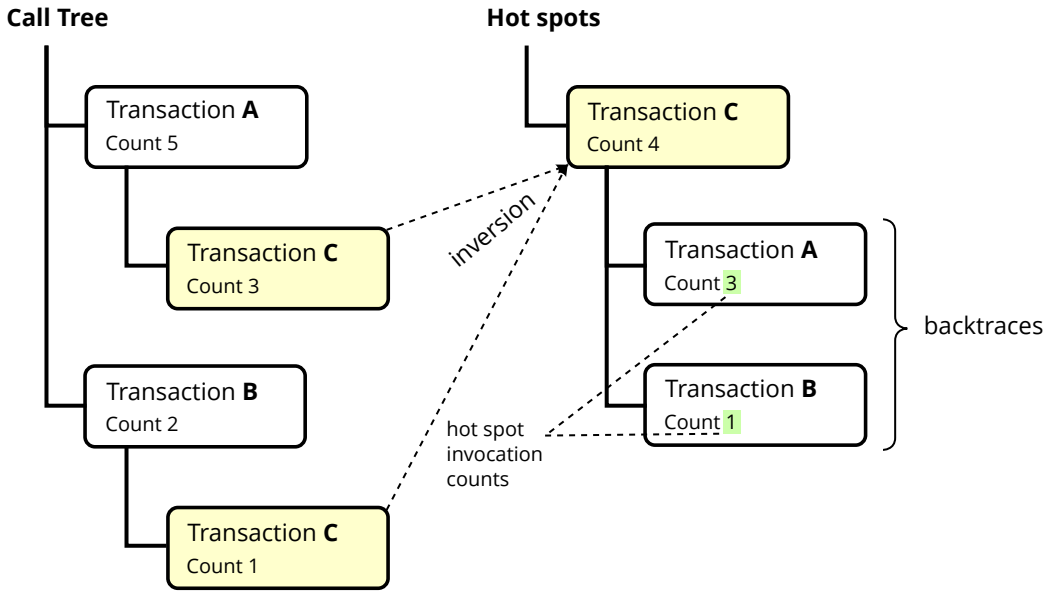
perfino builds a call tree from all recorded transactions. A call tree is a cumulated data structure that captures all the different sequences in which transactions are nested. If transaction C is called within transaction A and also within transaction B, it will occur twice in the call tree, once

as a child of A and once as a child of B. Each of these sequences can occur many times, increasing the invocation counts along the corresponding paths in the call tree.

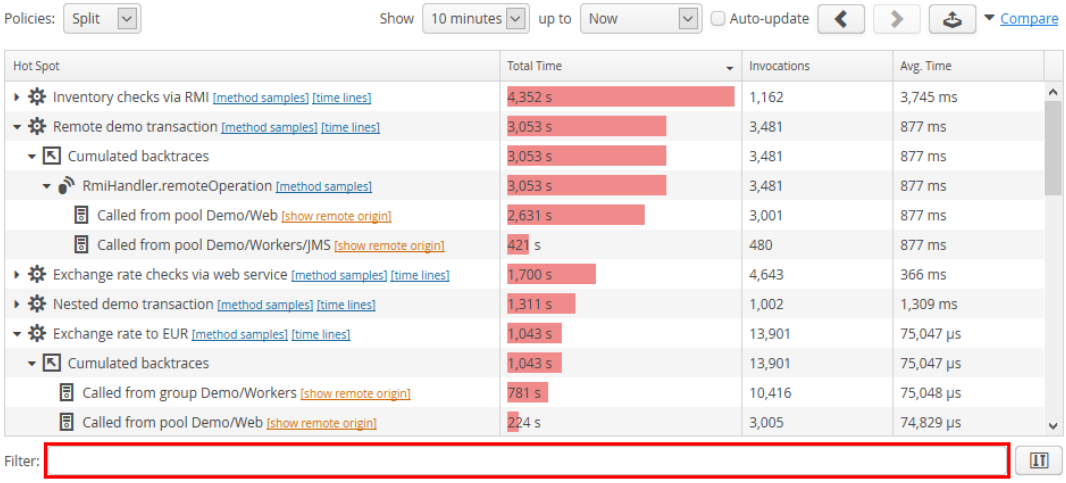


The numbers for each transaction - total time, invocation count and average time are displayed for the selected period. If the period is not fully measured, perfino will tell you the covered percentage at the bottom of the view. For example, when starting up the perfino server, the current hourly interval will not be fully measured until a full hour has passed. Similarly, if you stop the perfino server for some time, this will leave holes in the history with incomplete intervals at both sides.

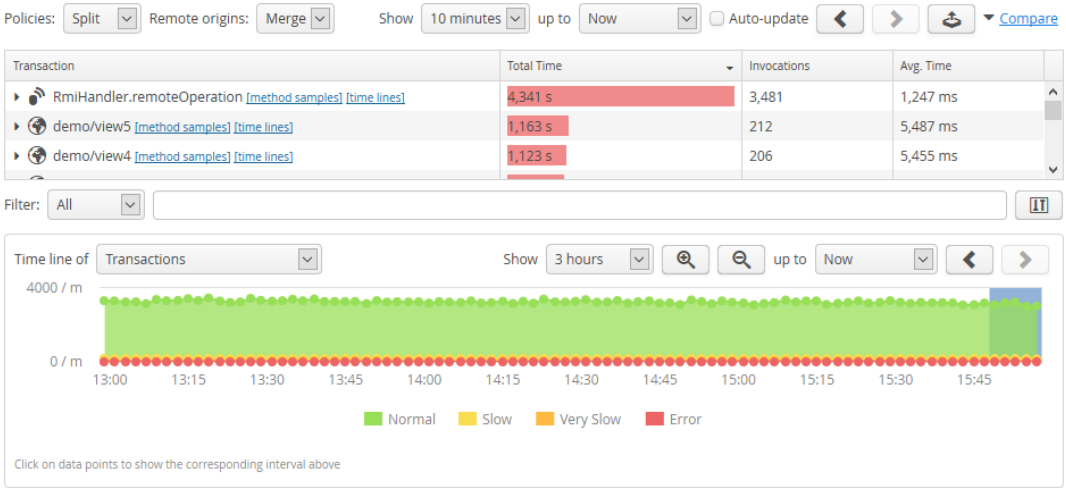
Hot spots are the inversion of the call tree, where all occurrences of each transaction are summed and shown with their **backtraces**. The invocation counts and execution times in backtraces do not refer to the transactions, but rather to the number of times the hot spot was called along this path.



To emphasize the point about backtraces, all hot spots have a child node called "Cumulated backtraces". Searching for transactions is done with the filter bar at the bottom of the view. Many views in perfino have such a filter bar.



In order to give you a chronological context for the selected interval in the call tree and hot spots views, a transaction timeline is shown at the bottom where the current interval is highlighted. You can also use this timeline to select another interval by clicking on a data point. When you display a timeline for a single selected transaction, the total transaction timeline is replaced and can be restored by clicking on the **[close]** button.

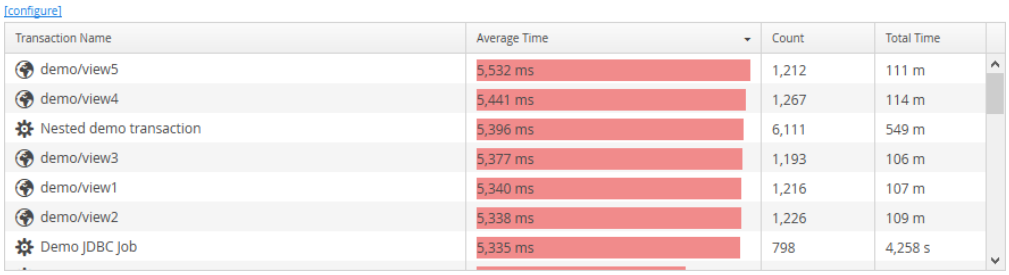


Transactions in the dashboard

Frequency, average time and total time of your most important transactions are key measures for your application. To give you a quick overview in that respect, the dashboard contains a transaction table that shows the most important transactions with these numbers from the last hour or the last day. You can sort by the different columns and the **[configure]** button at the top of the table allows you to select which of the columns should be used as a the "primary measure" and show a histogram next to the numeric values.

In the transaction table, transactions are not shown in the form of a call tree, but completely flat. This means that it also shows transactions that are not entry points but only invoked as

nested transactions. For a detailed analysis, click on the transaction row to go to the call tree view in the "VM data views" tab.



The drop-downs at the top provide three ways to adjust the range of the displayed transactions in the dashboard:

- **Period**

  By default, the dashboard shows the last running hour. Alternatively, you can switch to the last running day.
- **Request type**

  By default, all transactions are shown. Alternatively, you can restrict the display to web transactions only.
- **VM group**

  By default, all monitored VMs are shown. With the "VM group" drop-down, you can restrict the displayed data to a particular VM group and its nested groups. For example, if you select group "A/B", then all VMs that are contained in "A/B" are shown, but also VMs in "A/B/C".

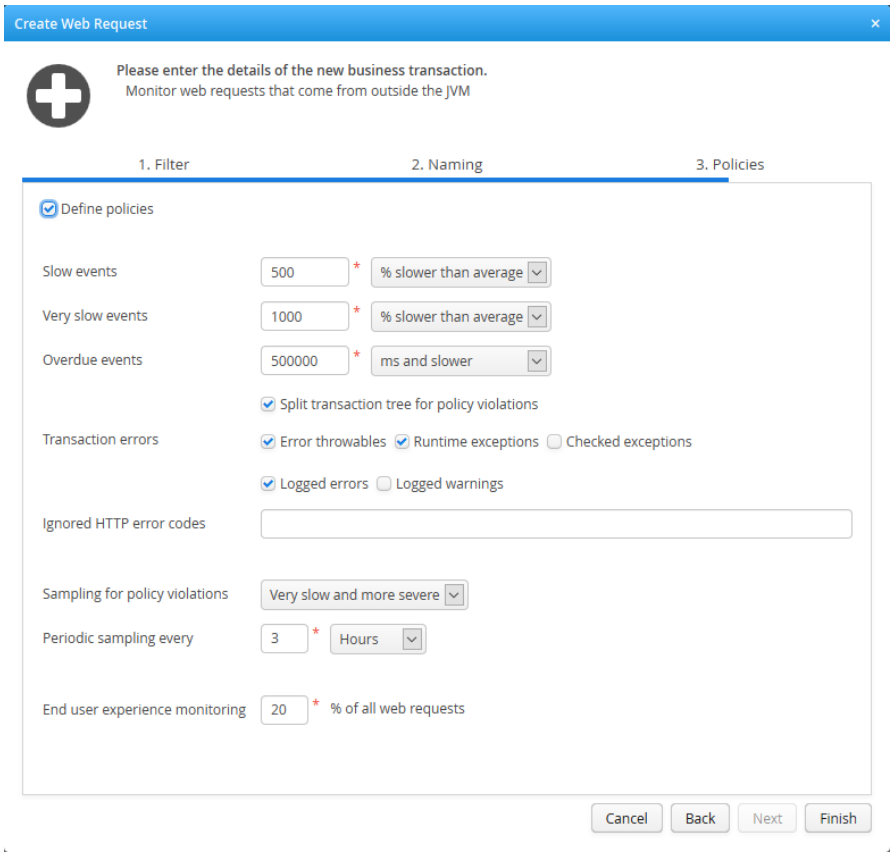
Overload protection

A common problem with new perfino installations is that web transactions have not yet been configured to map to high-level business transactions. As a consequence, too many distinct transaction names are being generated. In that case, perfino's overload protection [p. 106] mechanism caps the maximum number of transactions names and sends you an inbox message with instructions on how to fix this condition.

Policies

Configuration

Policies detect anomalous conditions with respect to transactions. Each transaction definition [p. 25] has associated policy definitions.



Policies create the following anomalous transaction states:

- **Slow**

  The threshold after which a transaction is classified as slow can be specified as an absolute time or as a relative percentage value with respect to the average duration of transactions that are generated by this transaction definition.
- **Very slow**

  Another time-based threshold like "Slow", only for a more severe category. You can partition "Slow" and "Very slow" so that "Slow" transactions can occur without notifications and "Very slow" transactions trigger notifications.
- **Overdue**

  After a third time-based threshold which is usually an absolute time and much larger than the "Very slow" threshold, you can view a transaction as deadlocked or hanging.
- **Error**

  Transactions that result in an error require special attention. There are three sources of errors:

1. **Exceptions**

   Exceptions derived from `java.lang.Error` (errors), `java.lang.RuntimeException` (runtime exceptions) and `java.lang.Exception` (checked exceptions) can be activated separately. Checked exceptions are usually handled by the application and do not necessarily result in a transaction error.

2. **Logged errors**

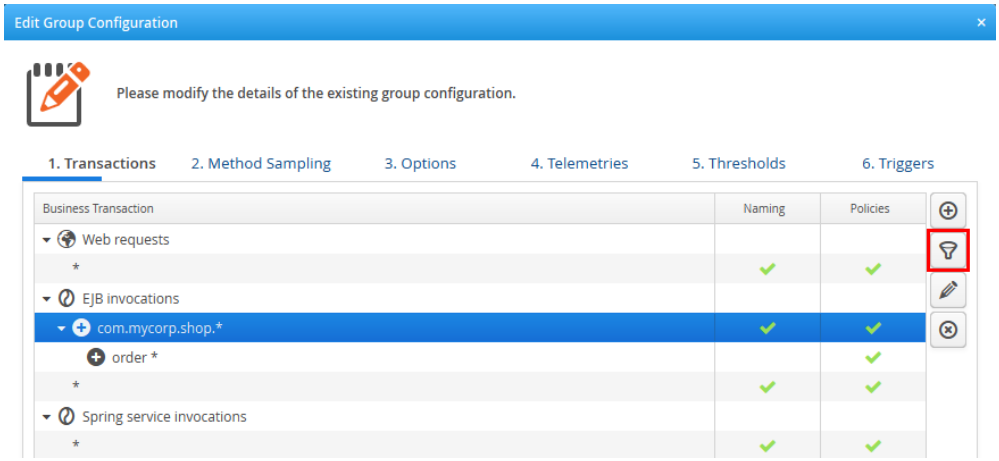
   If your application logs an error with a logging framework, perfino can mark the transaction with an error. `java.util.logging`, log4j, JBoss logging and Logback are supported by perfino. Optionally, you can also configure that warnings result in a transaction error.

3. **Specific conditions**

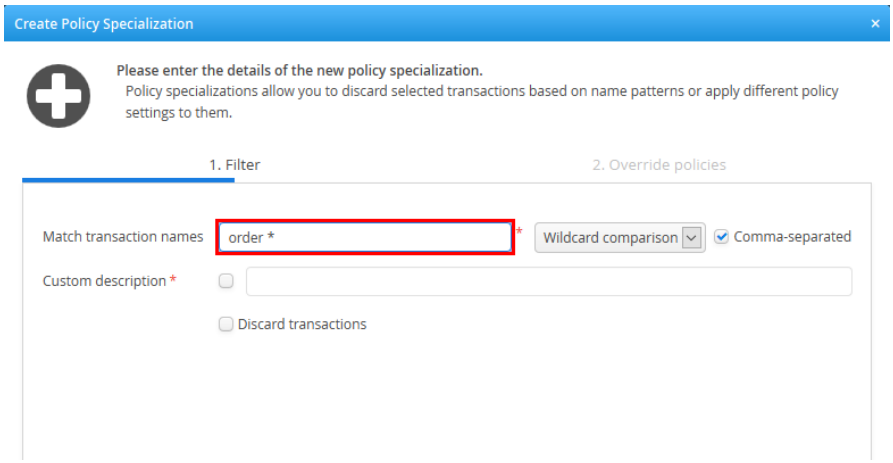
   Certain transaction types have their own built-in error conditions. For example, as shown in the screen shot above, web transactions can use the HTTP status code to detect an error. By default, all status codes >= 400 are handled as errors. If some of these status codes should not be viewed as errors, you can list them here.

If policies are not defined for a transaction definition, perfino continues in the list of transaction definitions until it finds a matching entry where policy definitions are enabled. This allows you to define the same policies for multiple transaction definitions, in the most extreme case with a "catch-all" entry that has no naming definition, but only policy definitions.

Sometimes, you want to go the other way and define more granular policies for certain transactions within a single transaction definition. For example, you might have a single EJB transaction definition which captures all your business transactions. If the acceptable durations depend on the transaction name, you can define **policy specializations** to avoid having to re-add the transaction definition and its naming scheme multiple times.

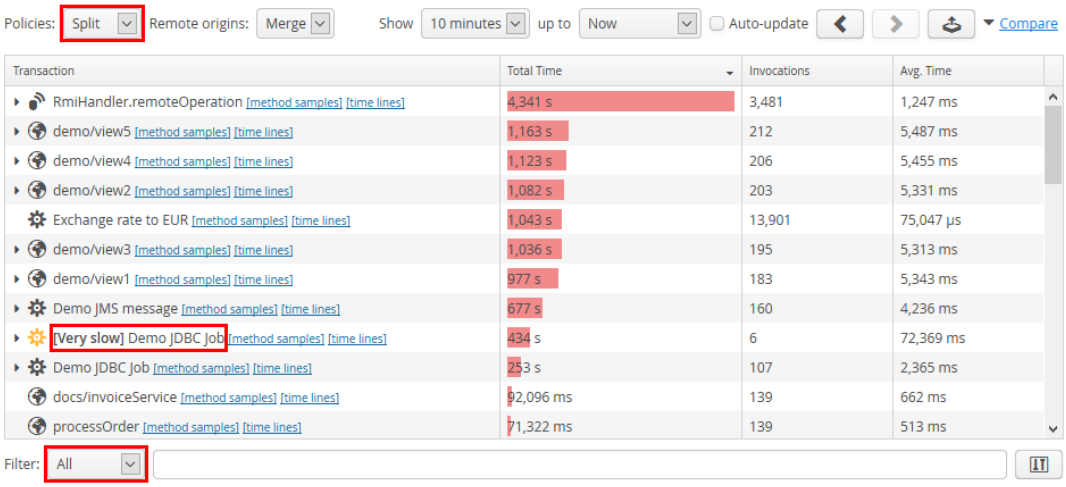


Unlike the parent transaction definition, which filters classes in this case, the policy specialization filters on the result of the transaction naming.



Policies in the call tree

Both call tree and hot spots views have a policy mode drop-down at the top. By default, the call tree is **split for different policy violations**. For example, this means that you can see slow transactions separately from normal transactions. This is important, since the cause of a slow transaction will often be visible from nested transactions or a slow database statement. Transactions with policy violations show their transaction type icon in the color of the policy violation so you can easily spot them in the call tree.



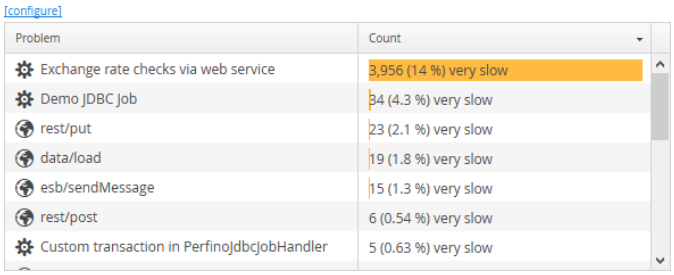
When you set the policy mode to "merged", all policy splits are added so that there is only one transaction of the same name and type on the same call tree level.

In the screen shot above, you can also see the policy selector at the bottom the view. By default it is set to "All". If you choose a particular policy violation type, the tree will be filtered immediately. For example, selecting "Error" will show all transactions that have been marked as an error, regardless how deep they are in the call tree. These transactions will be expanded and all ancestor nodes are shown, even if they are not transactions with errors. When you also add a filter text, only transactions with errors will be searched for the filter expression.

Policies in the dashboard

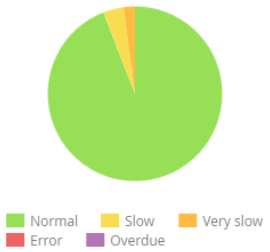
The dashboard is your first stop for checking the health of your application. That's why policy violations are featured prominently in the dashboard. The problems view shows transactions

with policy violations. By default, "Overdue", "Error" and "Very slow" are displayed. With the **[configure]** above the problems view you can select different policy violations and set absolute and percentage limits for each policy violation type. As in the transaction table, problems can originate from transactions that are not entry points but only invoked as nested transactions. For a detailed analysis, click on the problem row to go to the call tree view in the "VM data views" tab.



To visualize the proportion of policy violations with respect to the total amount of transactions, a policy pie chart is shown next to the transaction table. Numbers and percentages are shown as tool tips and by clicking on the legend entries, you can hide selected policy violation types.

Clicking on the slices of the pie chart will take you to the call tree view in the "VM data views" tab and set the filter drop-down below the call tree to the selected policy violation type.



Cross-VM Monitoring

One of perfino's core features is the ability to track calls between monitored VMs. No configuration is necessary to enable cross-VM monitoring. As soon as an outgoing and incoming call of a supported subsystem can be matched between two monitored VMs, perfino shows it as a remote call in the call graph and in the call tree.

The following remote call mechanisms are supported:

- RMI
- Web services: JAX-WS-RI, Apache Axis2 and Apache CXF
- Remote EJB calls: JBoss 7.1+ and Weblogic 11+

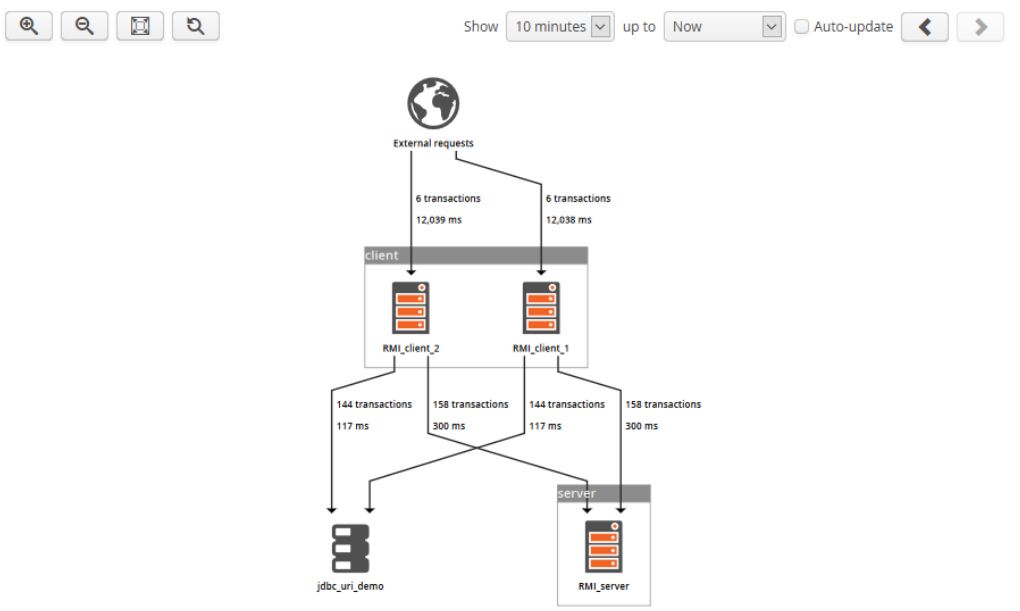
Call graph

The natural representation for remote calls is the call graph, where each pair of VMs that participate in a remote call are connected by an edge. In addition, two other node types are present in the call graph that usually represent calls to and from external processes:

- **External calls**

  The "world" node shows external calls that create transactions in monitored VMs, such as URL invocations.
- **Databases**

  Database nodes show database operations. If perfino detects that two VMs are using the same physical database, it only shows a single node in the call graph.



When a node or an edge is selected in the call graph, a **detail panel** is shown that holds several tabs with information about the associated transactions. The number and names of the tabs depend on the selected object:

- **External calls node**

  The only tab is the "Transactions" tab which shows the call tree of all transactions with external origin. Transactions are only shown in the first neighbor nodes, not across the whole graph.

- **Database node**

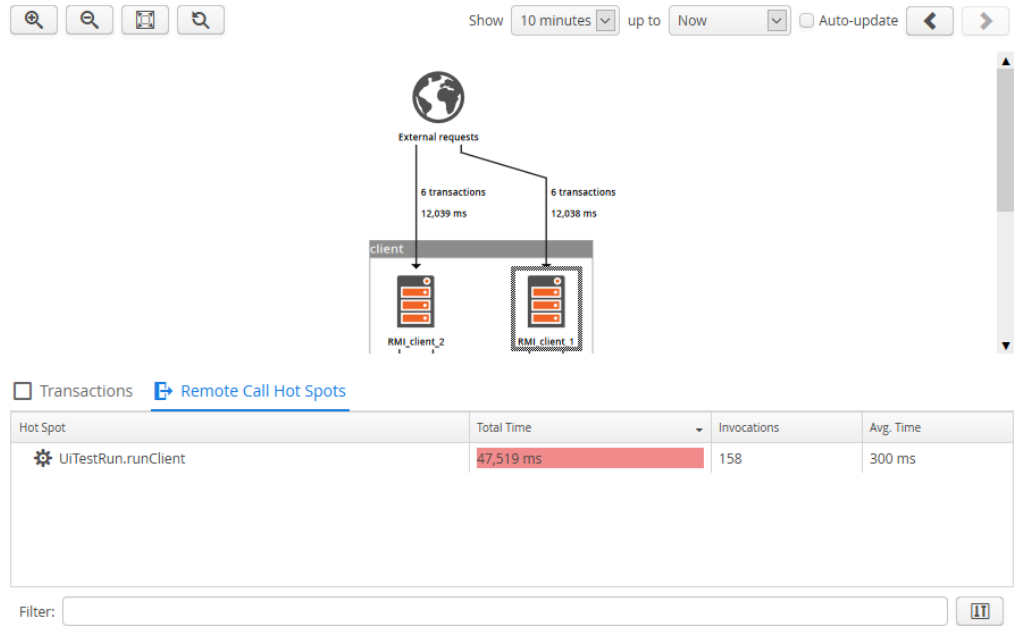
  The only tab tab is the "Remote origin hot spots" tab which shows the database operations with cumulated backtraces. To restrict displayed data to a particular VM, select an incoming edge instead.

- **VM node**

  The "Remote origin hot spots" tab shows the duration and count of handled remote calls mapped to the originating transactions from other monitored VMs. Backtraces are resolved up to the leafs that specify the VMs where the calls originated. The "Transactions" tab shows the call tree of all transactions executed in the selected VM. Finally, the "Remote call hot spots" tab shows the duration and count of remote calls originating in the selected VM. The top-level hot spots are the transactions that cause the jump into a different monitored VM. Check the outgoing edges to learn more about those calls.

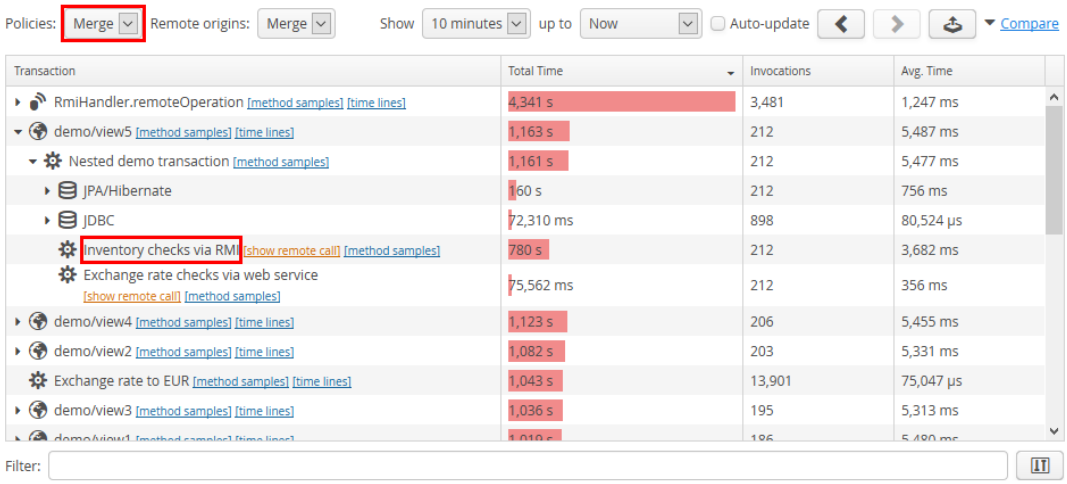
- **Edge**

  The "Call site hot spots" tab shows which transactions in the source node caused remote calls along the selected edge together with cumulated backtraces. For edges into "Database" nodes, this is the only tab and shows database operations instead of transactions. The "Execution sites" tab shows the call tree of the transactions that were caused by the selected edge. For edges starting at the "External calls" node, this is the only tab.

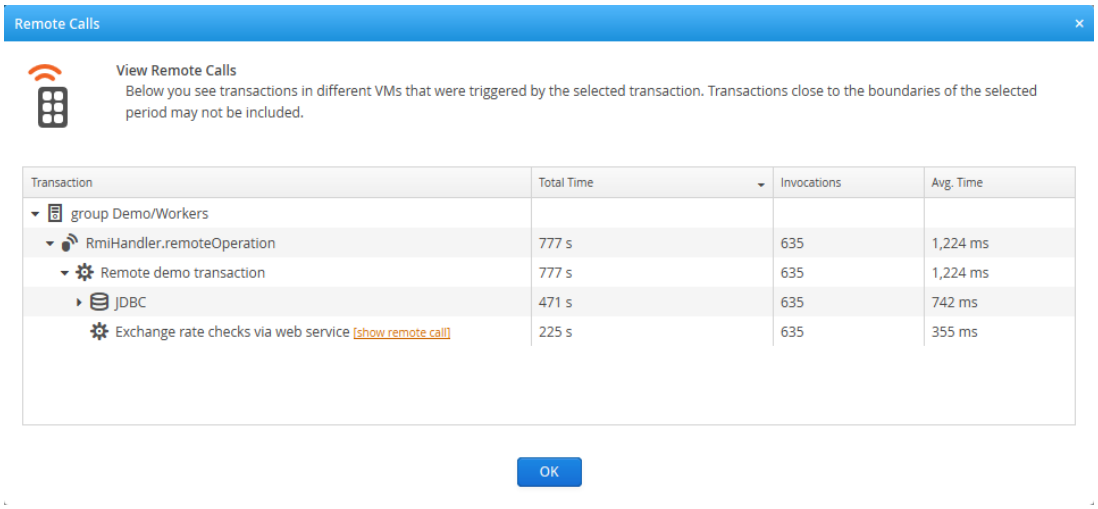


Call tree

In the call tree, you can recognize transactions that cause a remote call by the "show remote call" link that is shown next to the transaction name. When following such a link, it is advisable to select the "Merge" option for the policy split [p. 32] , otherwise the numbers on the remote side may not match the numbers in the source VM. This is because policy splitting is not tracked across VMs. In any case, there may be small discrepancies due to calls that are close to period boundaries which may be assigned differently by source and target VMs.

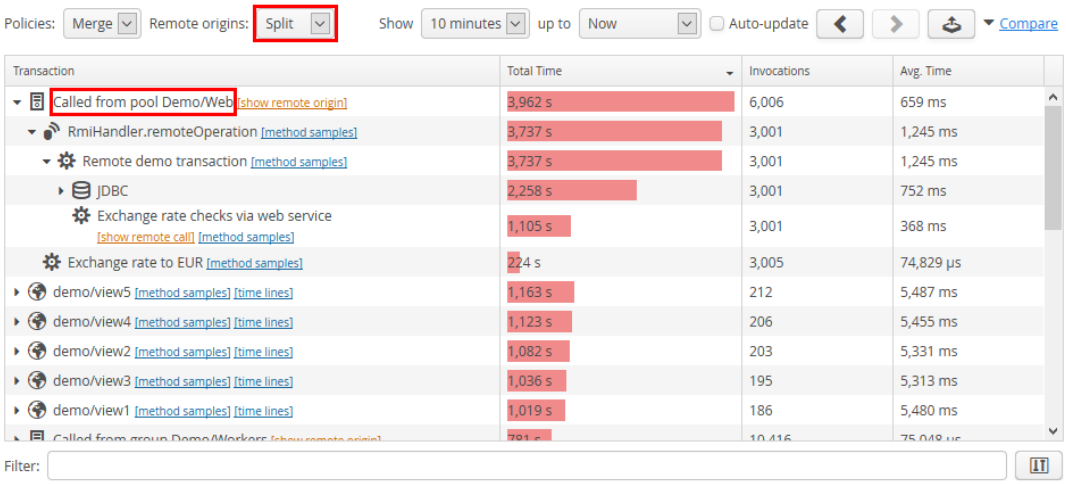


Clicking on the "show remote call" link opens a new dialog that shows the call tree of transactions in the remote VMs that were started by the selected transaction. At the top-level, the call tree is partitioned into VM groups.

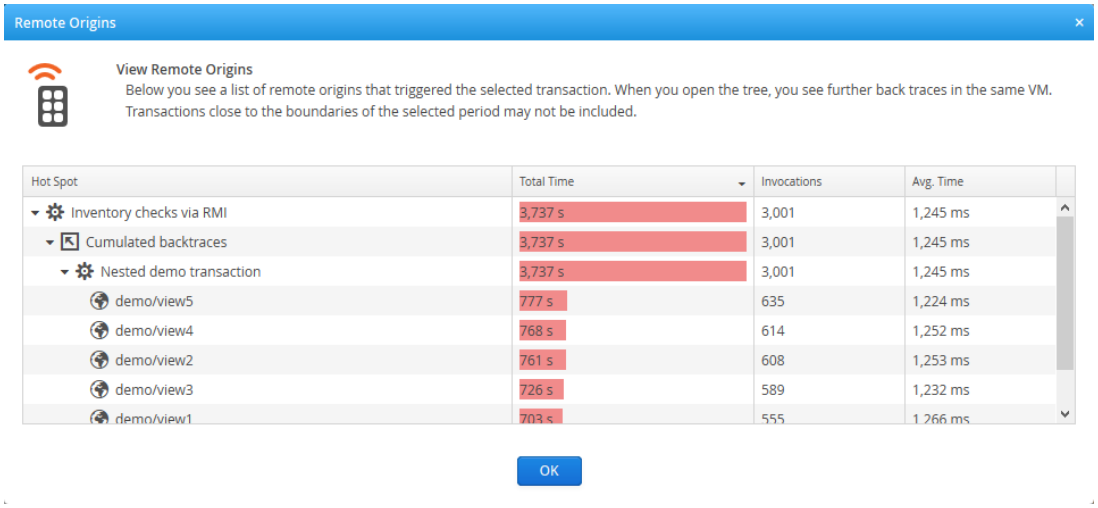


The call tree only shows the transaction in the target VMs. If there are further remote calls, you will see "show remote call" links, just like in the original call tree view. Clicking on those links replaces the call tree in the dialog and shows a **[Back In History]** button that can be used to return to the previous level. If the button disappears after clicking it, you are now looking at the first level.

It is also possible to move in the opposite direction and ask which remote calls in other VMs started transactions in the currently selected VMs. First, change the **remote origins** drop-down to "Split". By default, that drop down is set to "Merge" and you cannot see the remote origins. With the "Split" setting, "Called from ..." nodes appear at the top-level containing transactions that were started by remote calls from other monitored VMs. If you are looking at VM group or VM pool data, a split is performed for each VM group, but not for individual VMs. If the data is not granular enough, consider making your VM group structure more granular. If you are looking at the data for a single named VM, the split is performed for each remote VM.



Similarly to remote calls, clicking on the "show remote origins" link brings up a modal dialog. In the dialog, the count and duration of the selected transaction is attributed to the remote transactions that started it. The data is presented as hotspots with the innermost transactions from the originating VM at the top level. Analogous to the "remote calls" dialog, you can jump further back in a chain of remote calls when you see a "show remote origins" link next to a transaction. The **[Back In History]** button brings you closer to the original transaction.



Overload protection

If you monitor many VMs that all call each other, the number of the remote call sites grows with the square of the number of VMs. Remote call sites have a substantial overhead, since the transaction call trees have to be split for each remote call site. If too many distinct remote call sites are being created, perfino's overload protection [p. 106] mechanism prevents excessive resource usage.

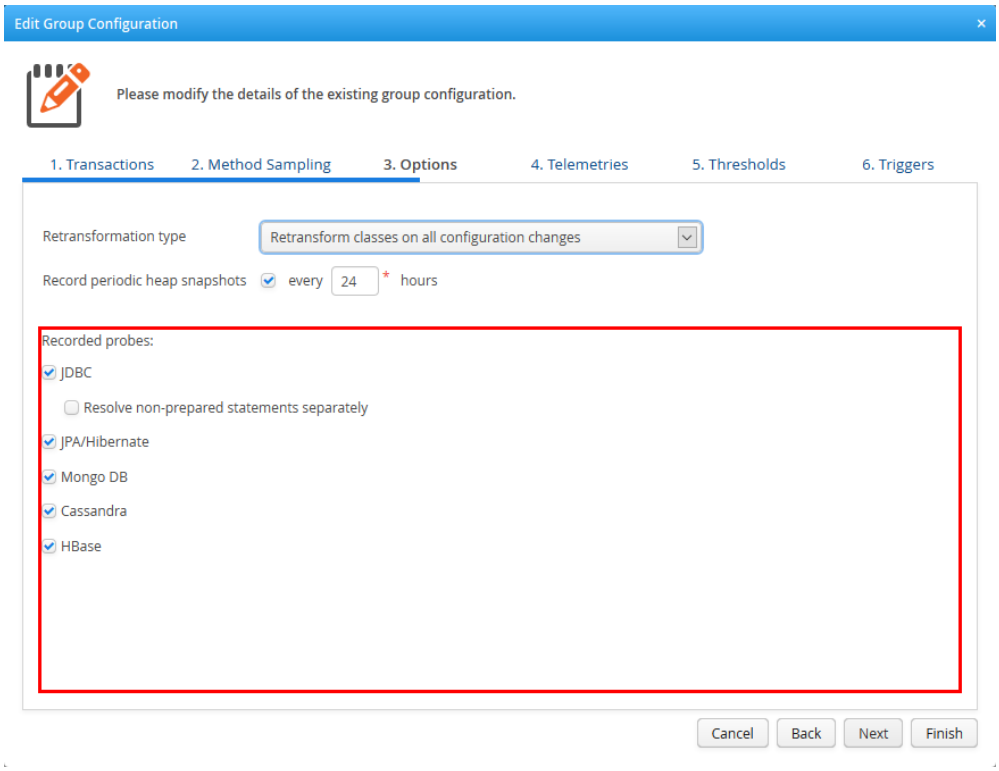
Probes

Transactions typically use external resources that are often the cause for performance problems. The most prominent example is JDBC. Just looking at the name of a slow transaction may not generate any insight, but if you can see the SQL statement that is responsible for most of the time, you can often find the cause of the slowness right away.

perfino records this kind of data with probes. Probes generate payload data that is attached to the currently running transaction. If no transaction is running, the data is discarded.

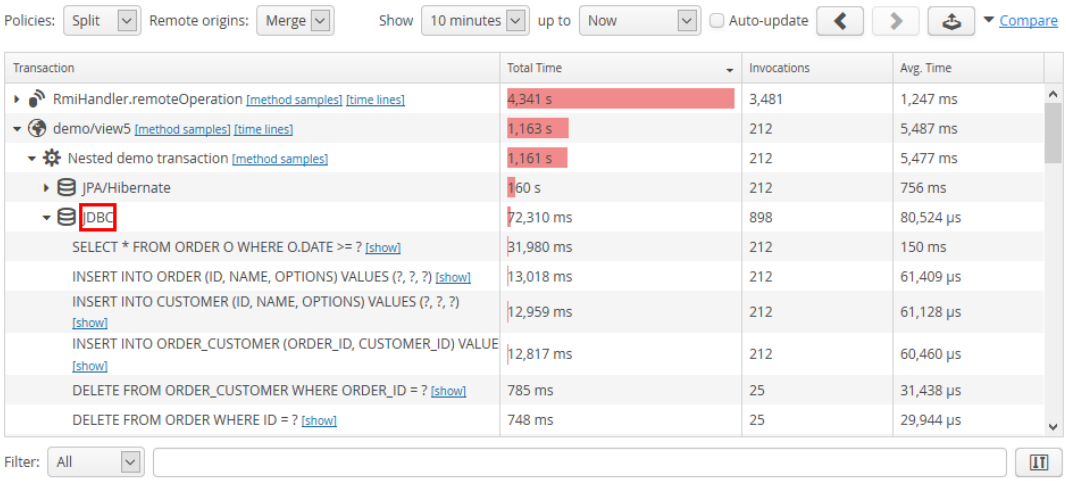
Configuration

To see the list of available probes, you can edit the recording settings of a VM group and go to the "Options" step. Most of the probes are for databases or messaging systems. You can switch off subsystems that you are not interested in. For example, if you use JPA, but do not want the overhead of measuring the duration of JPA calls, you can deselect the "JPA/Hibernate" check box.

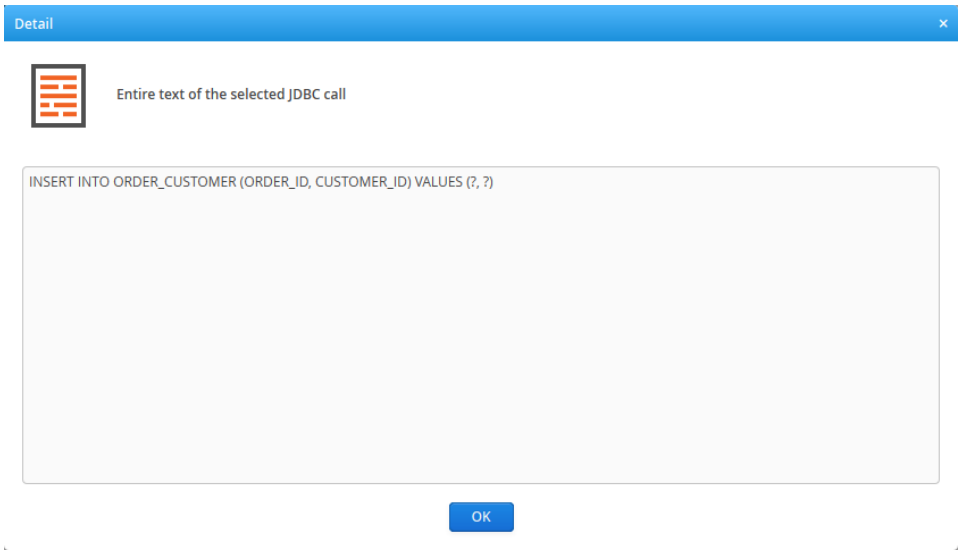


Probes in the call tree

Every transaction that performs an operation measured by a probe shows this information as child nodes in the call tree. Each probe type has a separate grouping node with the name of the probe.

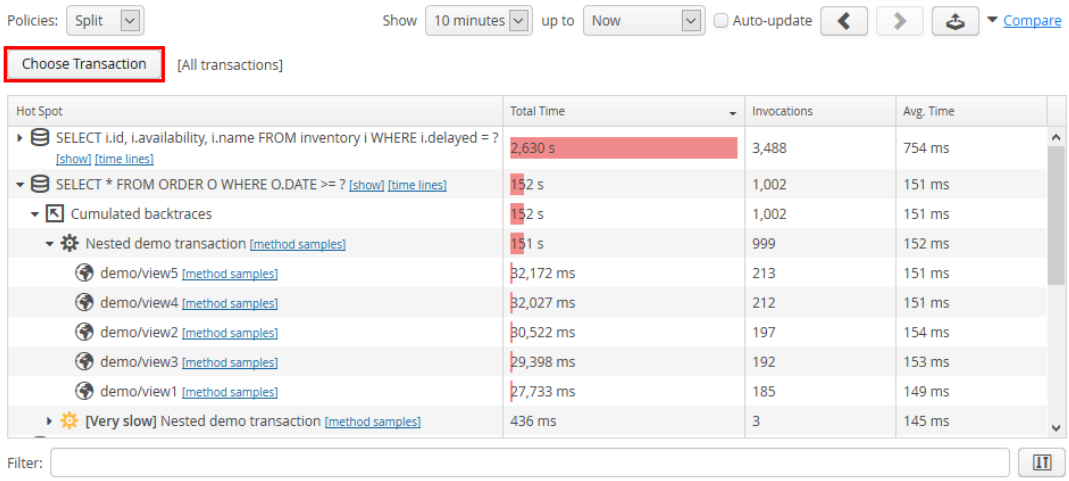


Often, the description of the operation is very long, like a complex SQL statement with many output fields, joins and conditions, and the finite length of the table column prevents further inspection. In this case, click the "show" link in the same table tell. A dialog will be shown, where you can scroll through the payload data and copy it to the clipboard.

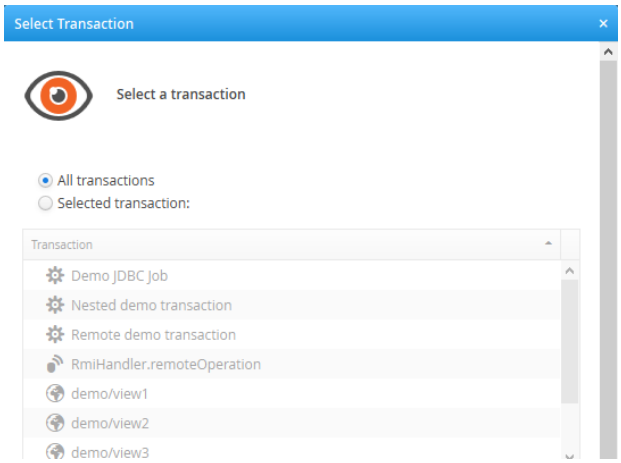


Probe hot spots

Sometimes your focus is not on transactions, but on the probe data. For example, when trying to speed up your database, the question is: What are my slowest database operations? For that purpose, perfino offers the probe hot spot views. Each probe has a separate view of this kind where a list of operations is shown together with cumulated backtraces.



The probe hot spots view can be filtered on a per-transaction basis. By default, cumulated data for all transactions in the selected time interval is shown. When clicking on the **[Choose transaction]** button, you can select one transaction from which the probe hot spots should be taken.



Probe data granularity

Probe operations can be extremely frequent and they usually don't do the same thing over and over again. perfino makes an effort to modify the displayed name of a probe so that frequently varying information is replaced with placeholders. Imagine a loop of SQL queries where an ID varies in each statement. Showing the raw SQL in the UI would mean that perfino would potentially have to remember millions of SQL statements per hour.

This is why perfino by default does not show SQL for JDBC statements that have been created with the `createStatement(...)` method. Rather, perfino shows those operations with a description of "unprepared select statement". SQL is only shown for prepared statements, without resolving the single parameters. If you do not use prepared statements and need to see more detail, the "Resolve non-prepared statements separately" option in the probe configuration enables SQL recording for all unprepared statements. Perfino will then try to replace literals in the SQL string with a placeholder to limit the maximum number of distinct SQL statements.

These considerations only apply to the call tree where data is retained indefinitely at the highest consolidation level. For sampling, where data is discarded in the medium term, perfino always records the SQL of unprepared statements.

The same strategy with respect to data granularity is used for other databases. For example, in MongoDB operations, all values are replaced with question marks.

Overload protection

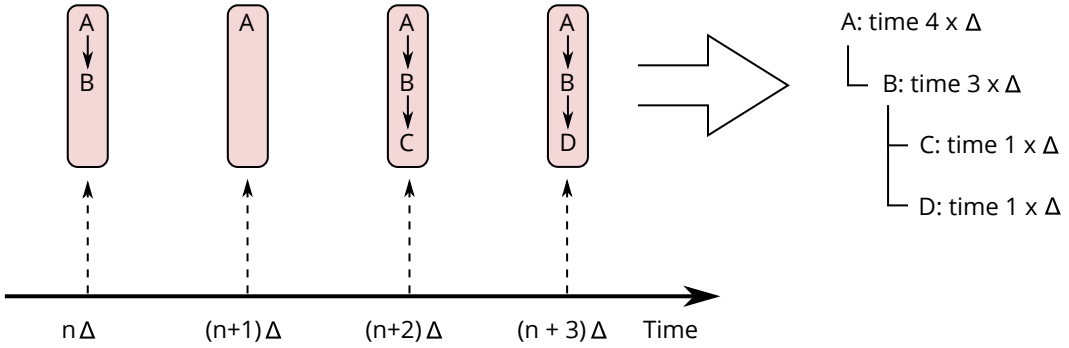
Under some circumstances, the number of distinct payload strings can grow linearly with time, for example if the prepared statement bodies contain IDs directly and not as bound variables or if the literal replacement of unprepared SQL statement fails. Too many different payload strings would overwhelm the system over time and so perfino's overload protection [p. 106] mechanism caps probe payload strings at a configurable maximum number.

Method Sampling

What is method sampling?

Policies in perfino help you to identify slow transactions, but the cause of the slowness is often not clear, since it usually originates in a subsystem that is not monitored separately. In that case, you need **more information on the method level**.

The way perfino gets this information is by periodically inspecting the thread that executes a transaction and by recording its stack trace. By comparing subsequent "samples", perfino can build an approximate call tree. It has no information on invocation counts, since it is not possible to tell if a method is still being called since the last sample was taken - or if it has exited in the meantime and is being called again. The more time is spent in a method, the better the information in the sampling call tree is. As such, sampling is a great tool to **find hot spots** in the monitored VM.



Automatic sampling

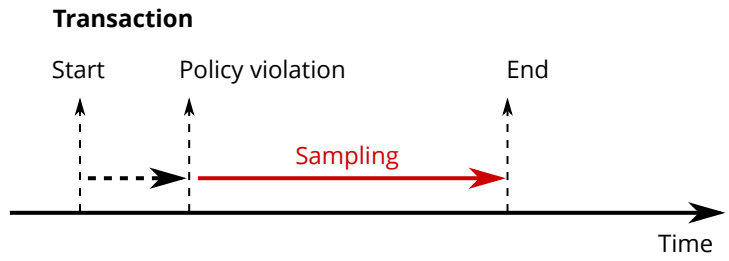
perfino can perform automatic sampling for transactions in two ways:

- **Periodic sampling**

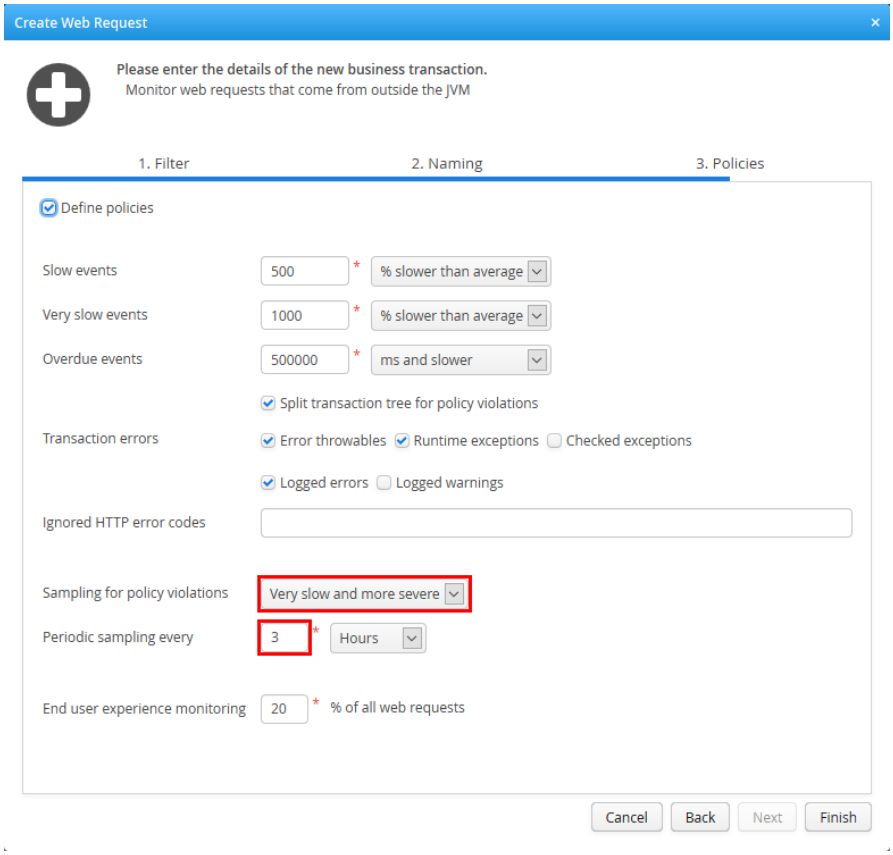
  With periodic sampling, perfino takes one transaction from a particular transaction definition every couple of minutes and samples it from start to finish. Alternatively, perfino can take every n-th transaction and sample it. The latter may be preferable for rarely executed transactions.

- **Sampling for slow transactions**

  The most interesting samples are for slow transactions, because they let you investigate why a transaction is slow. perfino can be configured to sample slow transactions automatically. When a transaction passes the threshold of a policy violation like "slow" or "very slow", perfino then starts sampling until the end of the transaction. The period of time from the beginning of the transaction until the policy violation was reached is not sampled.



Both automatic sampling modes are configured in the "policies" step of a transaction definition.



Filters

Method-level call stacks can become very deep. Too much detail is usually distracting, like the internal call structure of external libraries and the JRE. For each VM group, you can define the recording scope, either by specifying a list of packages that should be recorded (inclusive mode) or by listing packages that should not be recorded (exclusive mode). In both cases, the package filters are recursive.

For the exclusive mode, there are two types of filters:

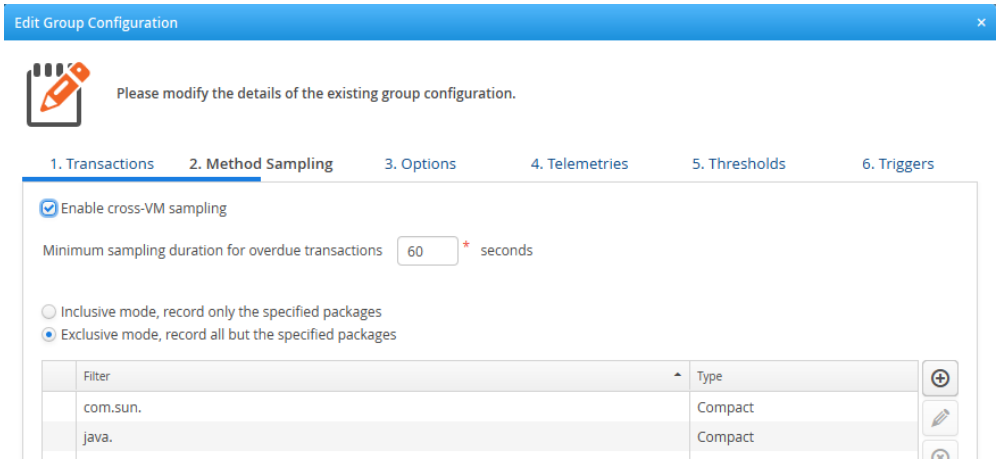
- **Compact**

  "Compact" means that internal calls will be excluded from the sampling call tree. If a filtered class is called as an entry point (like `java.lang.Thread.run()`) or from an unfiltered class, it will be displayed, but further calls within filtered classes will be hidden. This allows you to

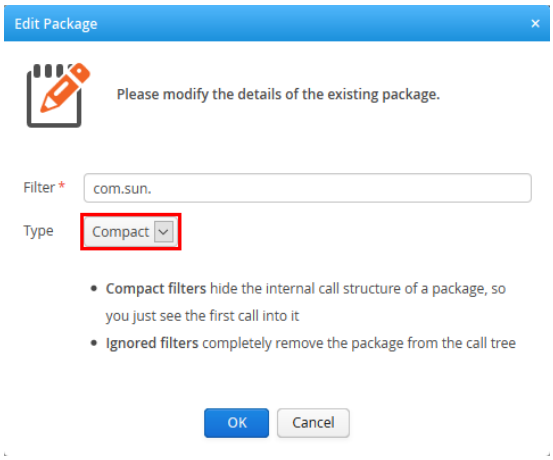
focus on the important methods that are in your code and where you are able to change something. However, you are still able to see calls into third-party code.

- **Ignored**

  "Ignored" means that filtered classes will be completely removed from the tree.



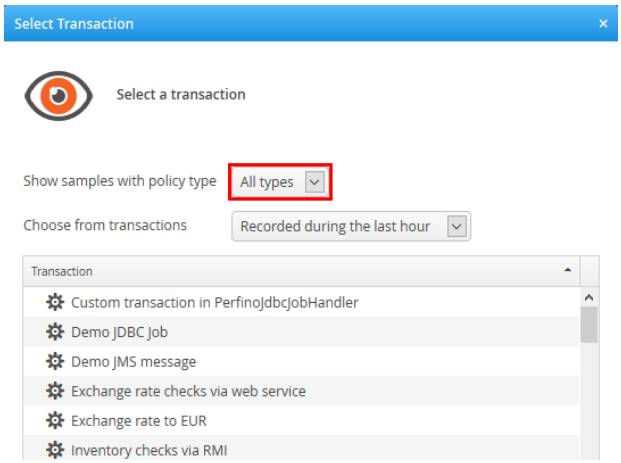
When you add a filter by browsing in archives or connected VMs, it is compact by default. To change its type to ignored, edit the filter and change the drop-down in the edit dialog.



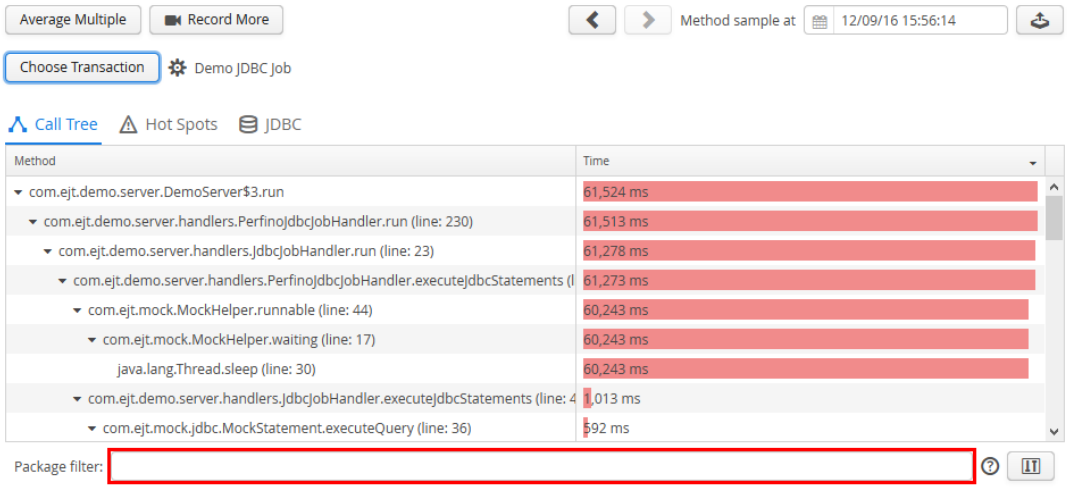
Method sampling view

To see samples in the method perfino UI, you first have to select a transaction and a policy type. While you can do that with the **[Choose transaction]** button, it is usually easier to click the "show methods samples" link next to a transaction in the call tree or hot spots views.

When choosing a transaction manually, changing the selected policy type to something other than "All types" shows the transactions for which a corresponding policy violation has actually recorded at least one sample in the selected period. In the "All types" setting, all transactions in the selected period are shown regardless of whether a sample has been recorded. In that way you can select any transaction and tell perfino to record a sample.



The selected transaction and policy type are shown in the header and the navigation buttons move between different matching samples. You can use the date chooser to jump to the sample that is closest to a selected time and date.



The sampling call tree shows methods except when sampling crosses a VM boundary. In that case, a VM node is inserted. If you want to limit sampling to the current VM, you can switch off cross-VM sampling in the sampling configuration of the VM group where the transaction is executed.

Where available, the sampling call tree shows **line numbers**. To interpret a line number, always look at the class of the parent item. Intuitively, you would expect the line number on a method item to show the line number of that method, but that would not be very useful information. The line number cannot be placed directly on the call site, because each call site can have multiple children with different associated line numbers.

After the initial inspection of a sampling call tree, you will often want to focus on a particular package. Modifying the sampling options to exclude all unwanted packages and re-recording the sample is not a good option, since the package of interest may vary or the performance problem may not be easily repeatable. To adjust your point of view in the sampling call tree, use the **package filter** at the bottom of the view. You can enter a single package, or a list of packages separated by commas.

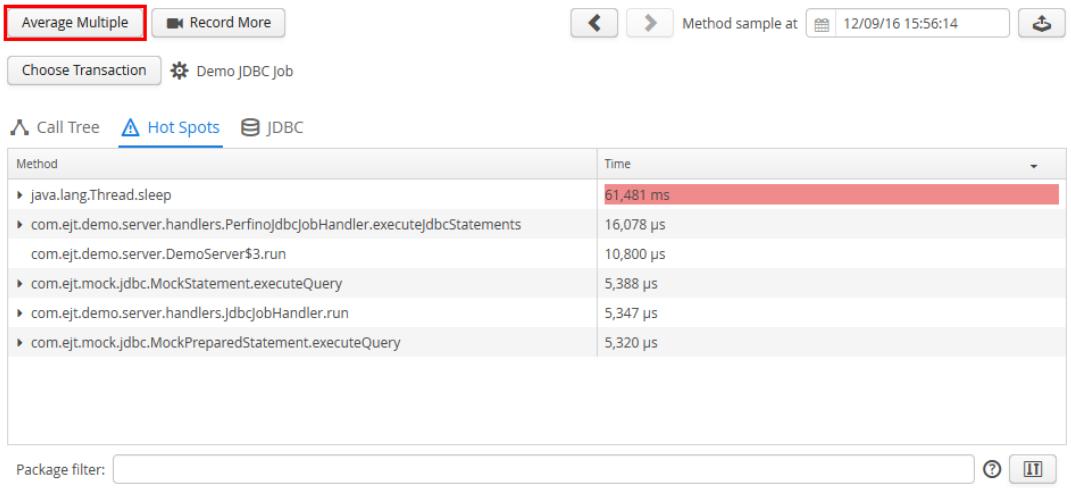
Packages are included recursively, i.e. if you include `com.mycorp.`, then packages like `com.mycorp.project` are included as well. If you want to exclude a particular package, prefix it with a minus sign. For example, if the classes in `com.mycorp.` are of interest, but `com.mycorp.algorithm` is distracting, enter

```
com.mycorp.,-com.mycorp.algorithm.
```

into the filter bar. To better understand the call structure, the **first excluded node between included nodes is always shown**, so even after excluding a package you might see a few instances of it in the call tree, just not more than two consecutive method calls in such a package. This corresponds to the "Compact" mode for exclusive filters in the "Sampling" configuration step.

If the first package in the filter bar is an exclusion (with a prefixed minus sign), then all other packages will remain visible. If the first package in the filter bar is an inclusion, then all other packages will be hidden.

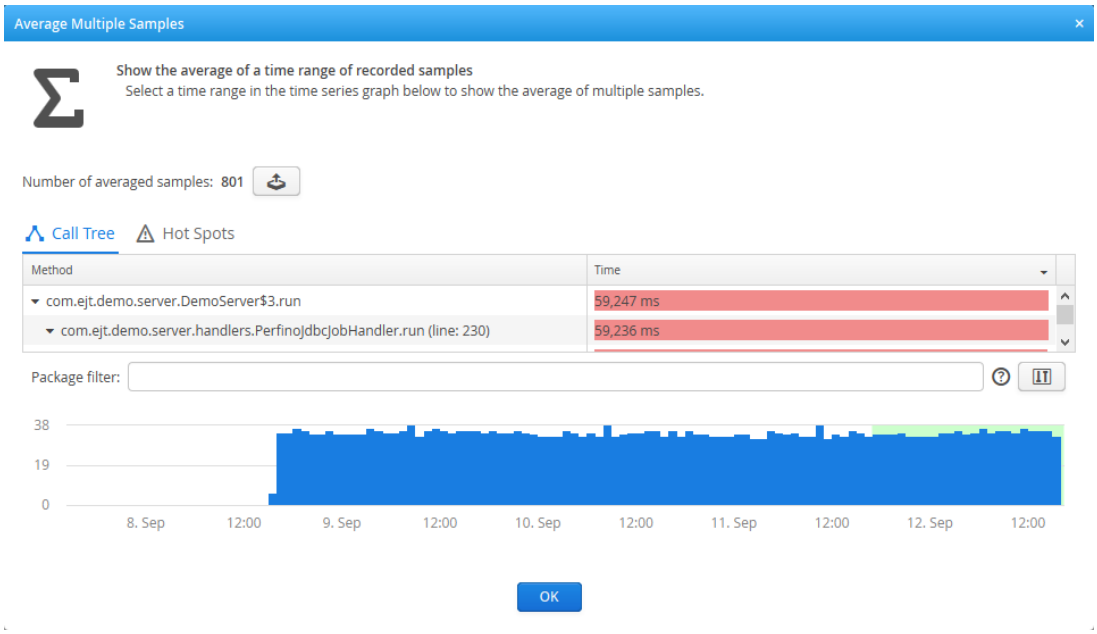
The **hot spot tab** shows the methods where most time is spent together with cumulated back traces. Note that the set of hots spots strongly depends on the granularity of the measurement which is governed by the defined filters. Defining different filters in the sampling options or applying different view filters may change the list of hot spots completely.



For each probe that recorded data during the sampling period, a separate tab is added that shows the probe hot spots [p. 40] .

Averaging multiple samples

The quality of sampling increases, the longer you sample. Especially for periodic sampling you collect a large number of comparable samples over time. perfino can average these samples to give you better statistics that comes closer to the actual execution times. When you click on the **[Average Multiple]** button highlighted in the screen shot above, a modal dialog will be shown where you can select a range of samples to be averaged.

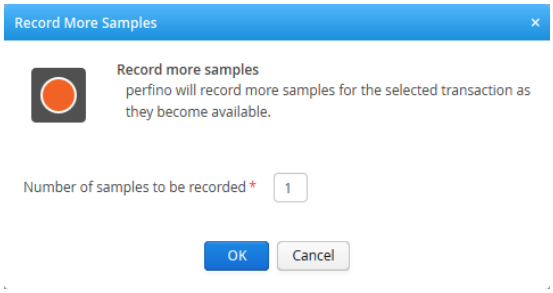


The time line at the bottom shows all available samples. Note that samples are not kept indefinitely due to their high storage space requirements. The retention threshold can be configured in the general settings. By clicking and dragging a range in the time line, you chose the samples to be averaged. The total selected number of samples is shown at the top.

Just like for the single samples, you can switch between call tree, hot spots and available probe hots.

Manual recording of samples

There may be cases where you require a current sample of a particular transaction and you cannot wait for periodic sampling or a policy violation that would trigger sampling. For that purpose, you can use the **[Record More]** button in the method sampling view.



You can even request more than one sample which will give you the ability to average the samples after they have been recorded. For rarely executed transactions, it may take a long time to fulfil the recording request.

Telemetries

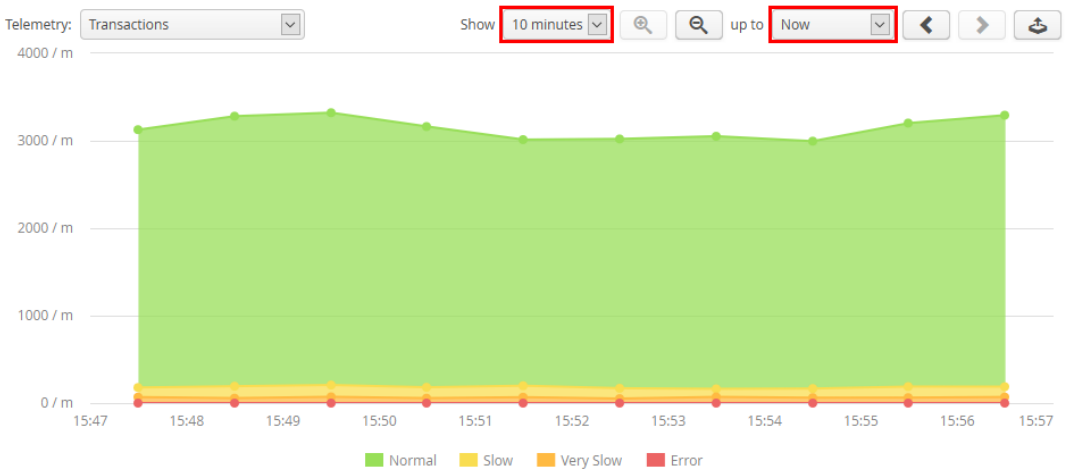
perfino observes scalar values from four different types of sources:

- Counts of entities that are managed by the perfino collector, like the number of connected VMs or the number of transactions. This is not a direct measurement in the monitored VM, but an evaluation in the perfino server.
- Telemetries that are measured inside the monitored VM, like the heap size or the average duration of a JDBC statement. Many of these telemetries are produced by probes [p. 40] .
- MBean telemetries that were configured in the recording settings.
- Devops telemetries where you have used the telemetry annotation from the perfino API.

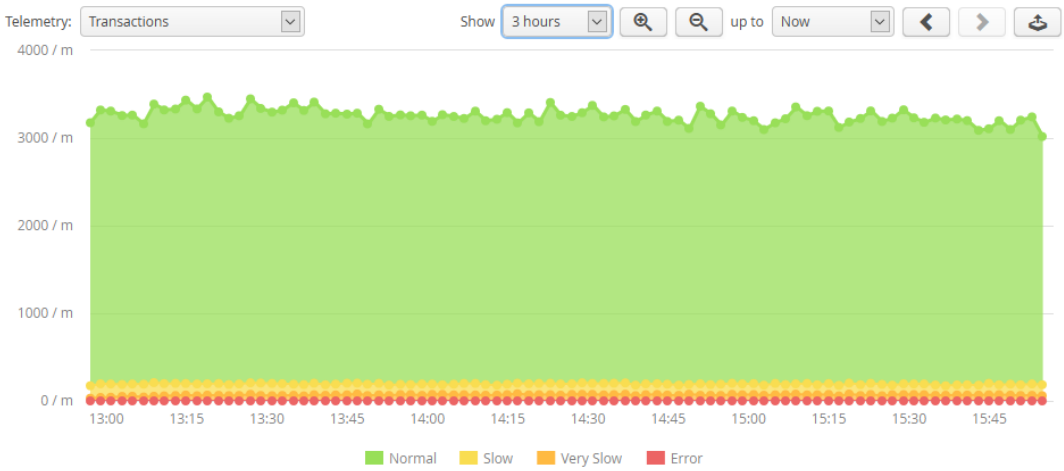
Telemetry data view

All telemetries are available from the "VM Data Views". Each standard telemetry has its own entry under the "Telemetries" node, and MBean and Devops telemetries are added under the "Custom telemetries" node. The latter is only visible if at least one custom telemetry exists.

Many telemetries are stacked area plots where the single lines add up to a total value. For example, the "Transactions" telemetry shows the total number of transactions over time, split into single lines for the various policy violations [p. 32] .



When you activate a telemetry view directly, a 10 minute interval up to the current time is shown. The data that is displayed here has been recorded with a resolution of one minute. Other telemetries that do not get its data from the observation of transactions will have a resolution of 10 seconds. You can move back in time with the navigation buttons at the top of the telemetry or the hover buttons at the edges of the telemetry itself, but at some point the one-per-minute resolution will end. perfino consolidates telemetries to progressively more coarse-grained resolutions and keeps them for progressively longer periods of time. Zooming out to an interval with a total extent of at least 3 hours, consolidated data points with a resolution of 2 minutes and a longer historical record are shown. Now you can move back further in time as compared to the previous higher resolution.



The full table with all display intervals where resolutions and retention times change is given below:

Display interval	Resolution	Retention time
10 minutes	10 seconds (1 minute for transaction-based data)	6 hours (48 hours for transaction-based data)
3 hours	2 minutes	10 days
3 days	1 hour	1 year
30 days	12 hours	unlimited

Moving to earlier or later times shifts the starting point of the displayed interval by 1/3 of the total display range. To skip full intervals, press the CTRL key while navigating.

In between those display intervals, there are many other intervals that just increase the displayed time extent, but use the same resolution. With the zoom buttons at the top or in the context menu of the telemetry, you can change the zoom level to view more or less data. Double-clicking on the telemetry zooms in at the selected point in time, if possible.

Some telemetries have multiple data lines that do not add up to a total value. In that case, there is a drop-down box above the graph and the telemetries are shown as line plots.

Detailed numbers can be obtained by hovering the mouse over the graph. in stacked area plots, you can toggle single data lines by clicking on the legend items.

In the context menu of the telemetry, there are actions for jumping to related views at the selected point in time. For all telemetries you can jump to the call tree and hot spots views. For telemetries that are calculated from probes, you can also jump to the associated probe hot spots view.

Sparklines

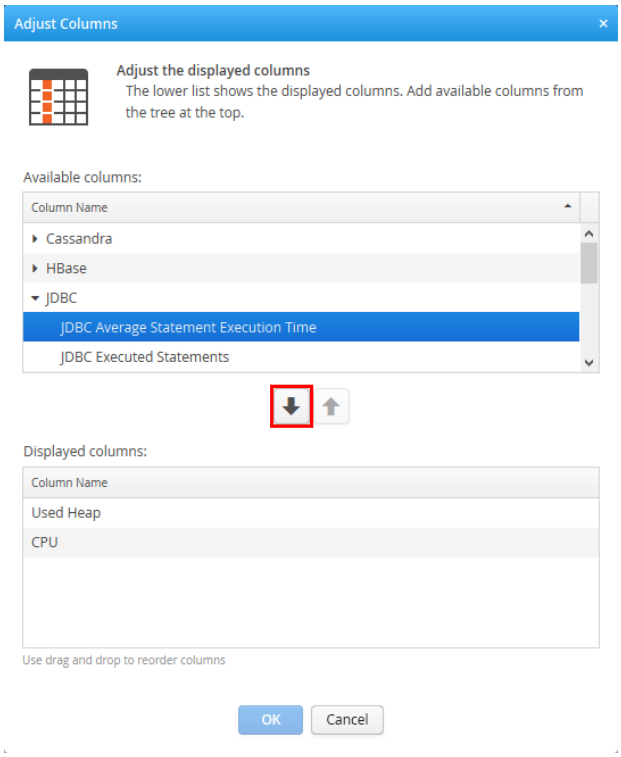
The VMs view and the dashboard show small versions of the current telemetry data as so-called "sparklines". Sparklines do not have labeled axes and are intended to give a visual impression of the recent development of an observed scalar value. The graph is followed by the current numeric value. The superscript indicates the observed maximum value, the subscript the minimum value in the displayed time range.

When you select the "Telemetries" or "Custom telemetries" category nodes in the "VM data views", a telemetry overview is shown. All contained telemetries are shown as sparklines for the last hour and the last day while the current value together with maximum and minimum values is shown in a separate column.

In the "VMs" view, you can configure a set of sparkline columns. This allows you to make a relative assessment of the different VMs and VM groups with respect to the monitored value. Sparklines can be scaled separately, with a common scale for each group or with a common scale for all VMs. This is configured in the options popup.

In the dashboard, sparklines are displayed in a table rather than as columns. The useful number of sparkline columns that you can add in the VMs view is limited due to the finite width of the table. In the dashboard, you can add a lot of sparkline rows without any such restrictions. Rather than showing data for separate VM groups or VMs, the dashboard only shows data for the selected VM group.

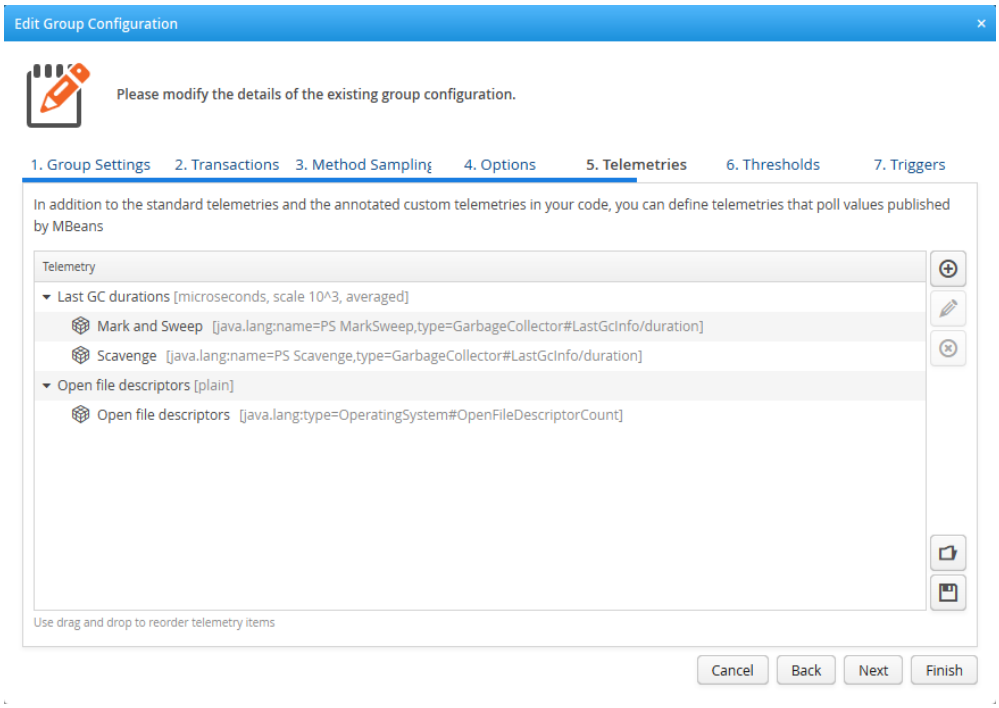
As in the VMs view, there is a "Configure" action that takes you to the list of all available telemetries. The telemetries are grouped into categories. The lower list shows the currently configured sparklines.



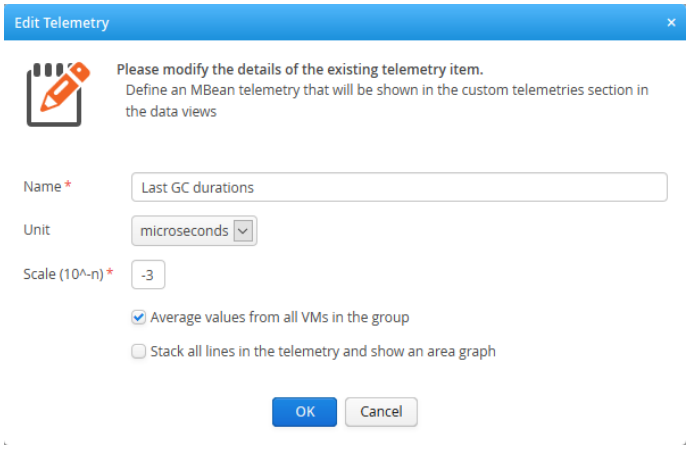
By clicking on a sparkline in the dashboard, the "VMs" view or the telemetry overview, the full telemetry view is activated with a time interval that most closely corresponds to the interval that was shown in the selected sparkline.

MBean telemetries

Many application servers and frameworks publish MBeans with values that are interesting for monitoring purposes. All numeric values that are published by an MBean can be polled by the perfino agent and become part of a telemetry. To this end, it is not necessary that a JMX server has actually been opened to the outside. It is enough if an MBean was registered with any MBean server internally.



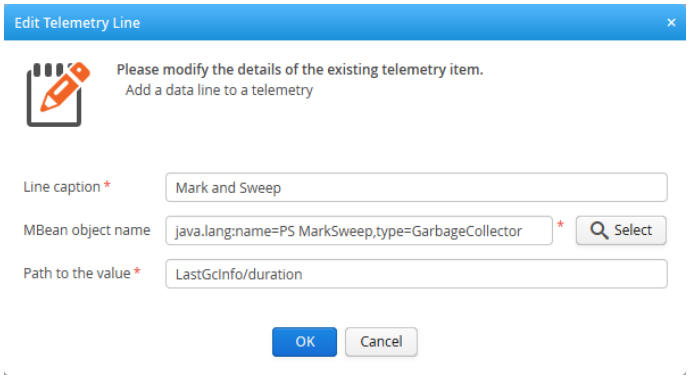
In the "Telemetries" step in the recording settings, you can add MBean telemetries and their telemetry lines. The telemetry defines the name of the telemetry, the unit, and its overall behavior while the telemetry lines define the actual data.



If the telemetry lines are parts of a total value, you can stack them into an area graph. If the monitored values from different VMs should be averaged, choose the "Average values from all VMs in the group" otherwise the values will be summed. Summing makes sense for business measurements such as "number of logged in users" or the use of a shared resource such as "number of database connections".

The configurable units are base units and unit prefixes will be added as required. For example, if you select the "bytes" unit, large values in the telemetry will be shown as "kB", "MB" and "GB" automatically. Sometimes a scale factor has to be applied to get from the recorded value to the selected unit. You can add that scale factor as a negative power of 10, i.e. to multiply by 0.01, the scale factor is 2.

Each MBean line is defined by an **MBean object name** and an **MBean value path**. You can most easily obtain these settings by clicking on the **[Select]** button in the MBean line configuration dialog. An MBean attribute browser [p. 81] is shown that allows you to select a numeric value from one of the monitored VMs. The configured line names are shown in the legend of the telemetry.



perfino will not create the platform MBean server if it does not exist, so if you configure a telemetry from the platform MBean server, you must call

```
ManagementFactory.getPlatformMBeanServer();
```

in your application at startup to be sure that the telemetry will work after a restart of the monitored VM.

Devops telemetries

To monitor any scalar value in your application, you can add a static method that returns that number anywhere in your code. Then, annotate the method with the `@Telemetry` annotation. You have to make sure that the containing class is actually loaded, otherwise perfino will not detect the annotated method.

You can display the custom telemetry by going to the "VM Data Views" and locating the telemetry under the "Custom telemetries" node.

Once an annotation telemetry was detected by perfino, it will always be shown in this list, regardless of whether the annotated method is currently available in a connected VM. When you retire such a telemetry, go to the general settings, and click on **[Configure Hidden Devops Telemetries]**. Here you you can hide selected telemetries. Note that the telemetries are matched by name and not by the annotated method.

For more information, please see the Javadoc of the `com.perfino.annotation.Telemetry` class in the `api/doc` directory of your perfino installation.

Time zones

All displayed times are expressed in the time zone of the server. If you are in a different time zone, there will be an offset. perfino detects this condition by inspecting the time zone provided by the request headers from your browser. In that case, the current time with an explicit time zone is shown in the header.

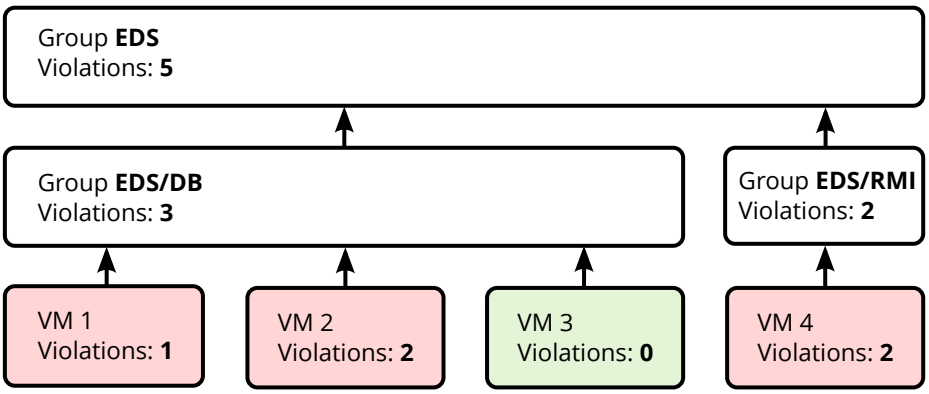
Thresholds

Thresholds detect anomalous conditions for telemetries. Threshold violations are not directly coupled to alerts or other actions, they just **increase an associated counter**. Thresholds can be configured for single VMs or for VM groups. This is different from triggers [p. 60] that always operate on a VM group level rather than for single VMs.

For **single VM thresholds**, the telemetry value of each VM is checked and for each offending VM a threshold violation is created. For example, you may have an upper bound on the used heap and each VM that uses more than that gets its own threshold violation.

For **VM group thresholds**, the telemetry value of the VM group is checked and only one threshold violation is created no matter how many VMs violate the threshold individually. Imagine a database that serves 10 VMs in a VM group. If that database becomes very slow and you have defined a threshold for the average JDBC execution time, all 10 VMs will report a threshold violation. In the end, you just want one alert and not 10. In addition, the averaged group value is smoother and fewer spurious threshold violations will occur than for single VM thresholds.

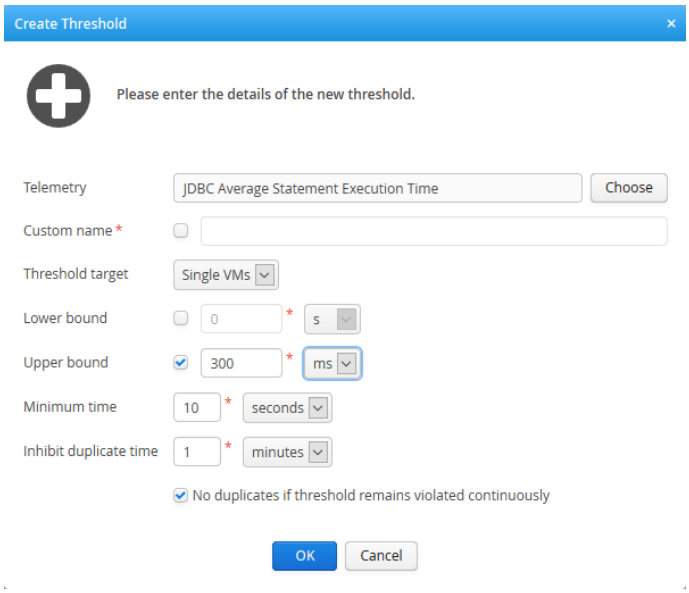
All telemetries that show frequency data are summed for VM groups. If you have 10 VMs that perform 10 JDBC statements per second, the VM group will show 100 JDBC statements per second. If your acceptable range is defined for the total values rather than the loads on the single VMs, then you have to configure your threshold as a VM group threshold.



The counter is maintained on a per-VM level for single VM thresholds as well as on a VM group level. When a threshold violation occurs, it bubbles up through the parent hierarchy, increasing all the associated counters by one. At each VM group in that hierarchy, you can define triggers that react to the corresponding number of threshold violations.

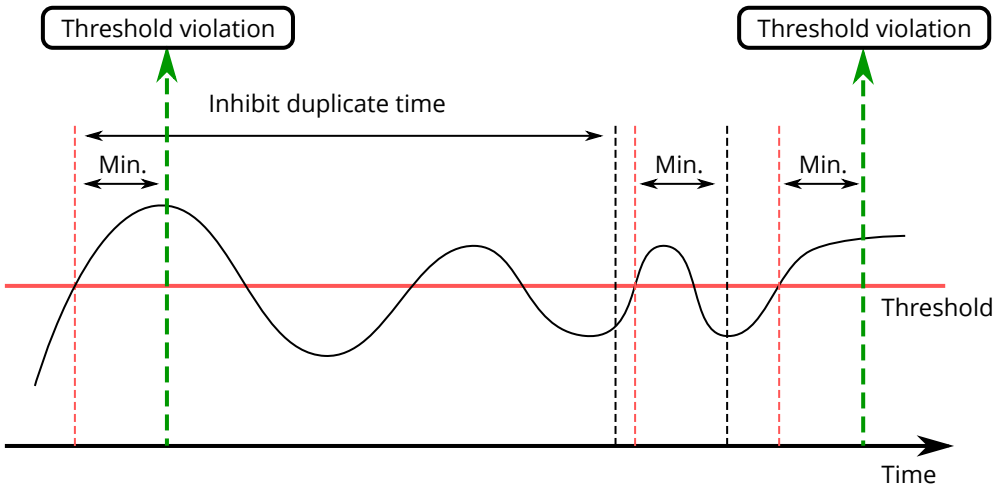
Configuration

You can define thresholds for each telemetry in the recording settings of a VM group. A threshold definition has an associated telemetry and optional **lower and upper bounds**. The available units depend on the selection of the telemetry.



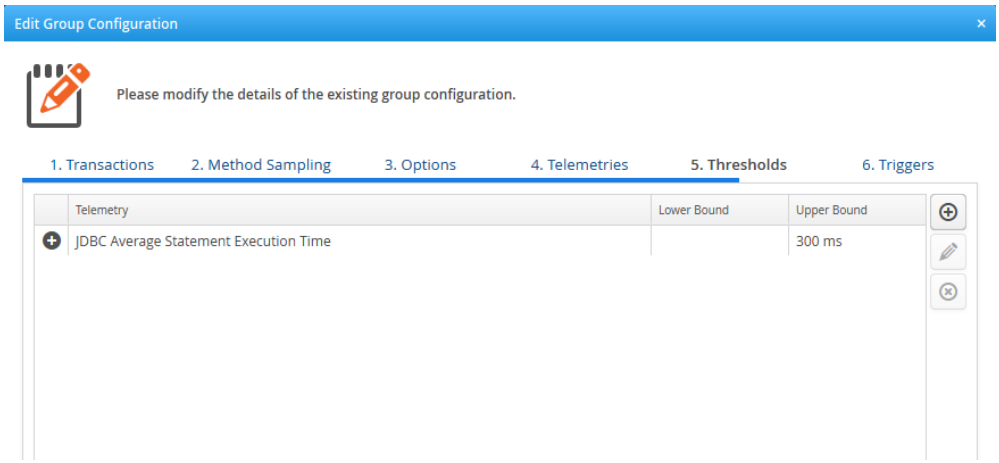
In some cases, you need multiple threshold definitions for the same telemetry, for example to designate different severities. In that case, you have to give the thresholds **different custom names**, like "High average JDBC execution time" and "Very high average JDBC execution time". These definitions can be used as the basis for triggers with different escalation strategies.

To avoid spurious firing of threshold violations, a minimum time can be configured for which the bounds have to be exceeded before a threshold violation is detected. After a threshold has fired, there is an inhibition time, during which the threshold is muted and cannot fire again. This serves to prevent frequent firing in the case of oscillating conditions.



If the threshold is continuously violated, perfino will not fire any more threshold violations during that time. There is a check box to disable this constraint. If disabled, new threshold violations will be fired at a constant rate with a period of the configured inhibition time.

Like for transactions and method sampling filters, you can save and load sets of thresholds. This makes it possible to copy and paste threshold definitions between VM group configurations.



Threshold violation data

While threshold violations are mainly used to generate alerts, it can be useful to inspect the actual data to see where the threshold violations are coming from. In the VM data views, the "Threshold violations" view shows a list of threshold violation types. Each threshold violation type contains a cumulated group hierarchy tree that shows which VMs or VM groups are responsible for the total count. The nodes in the back traces are the VM groups, the leaf nodes are the single VMs or the VM groups where the thresholds are defined.

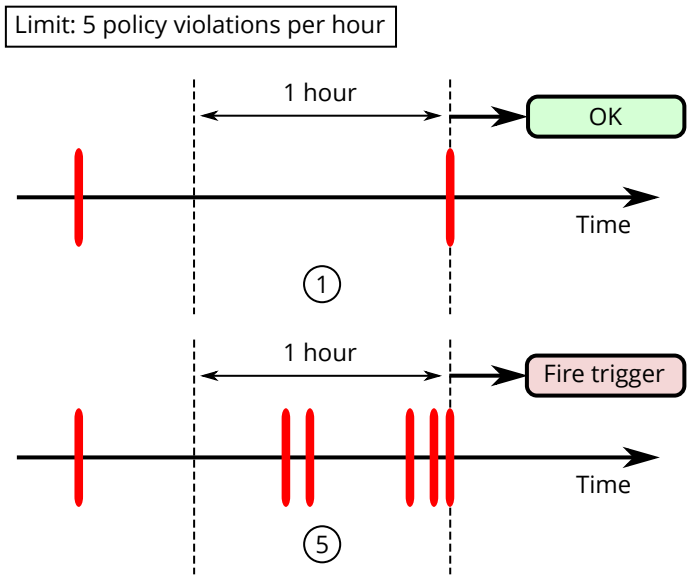
Triggers

While policies [p. 32] and thresholds [p. 57] detect anomalous conditions and display them in the dashboard and the VM data views, they cannot take any actions by themselves. With triggers, you can react to policy and threshold violations and execute a list of configurable actions.

Mechanism

Triggers operate on a different level than policies and thresholds. The latter are configuration options that are applied to single VMs or VM groups. Triggers, on the other hand are not directly coupled to single policy or threshold violations. Rather, they **react to sequences of such events** that originate from all monitored VMs in a VM group.

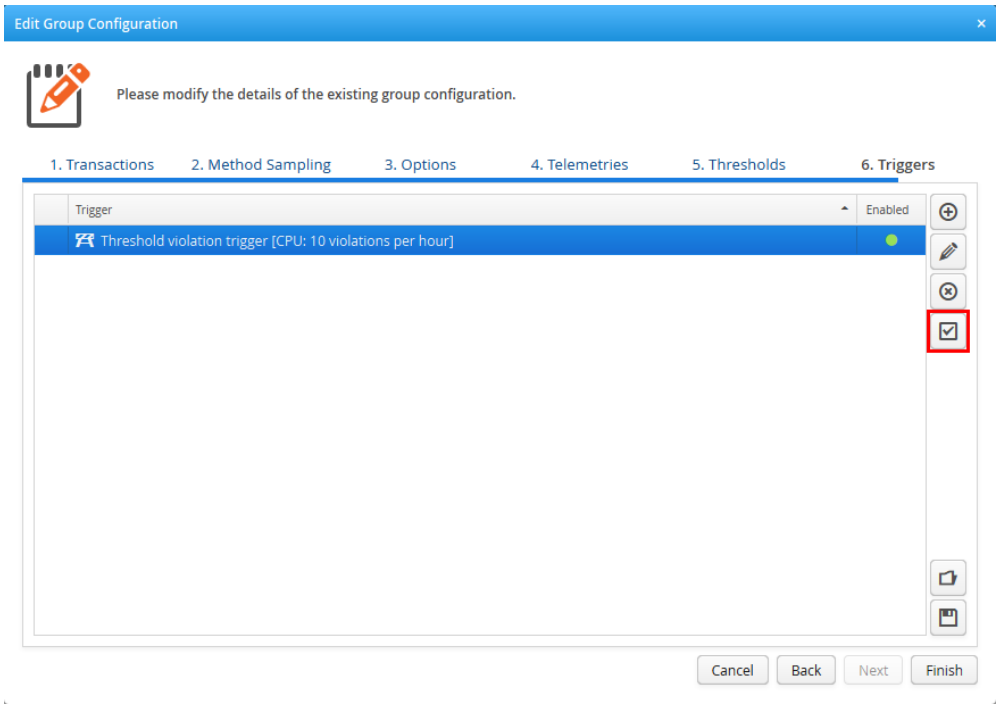
This mechanism is intended to give you greater flexibility for deciding what constitutes a condition that requires a particular action. For example, you might expect up to 1 slow URL invocation when a cache is rebuilt. However, if there are 5 slow URL invocations per hour or more, then something is wrong. The definition of what is acceptable and what is not, strongly depends on the type and the implementation of your application.



Configuration

In the recording settings, you can edit triggers for each VM group. Triggers operate on all recursively contained VMs. It is possible to define different triggers for a VM group and an ancestor VM group, both sets of triggers are handled separately.

In some cases, triggers are fired too often. For that case, perfino allows you to disable a trigger until you have time to figure out how to change the underlying configuration.



Like for transaction definitions and other entities in perfino, triggers can be saved to and loaded from trigger sets. This enables you to copy and paste trigger definitions as starting points to multiple VM groups.

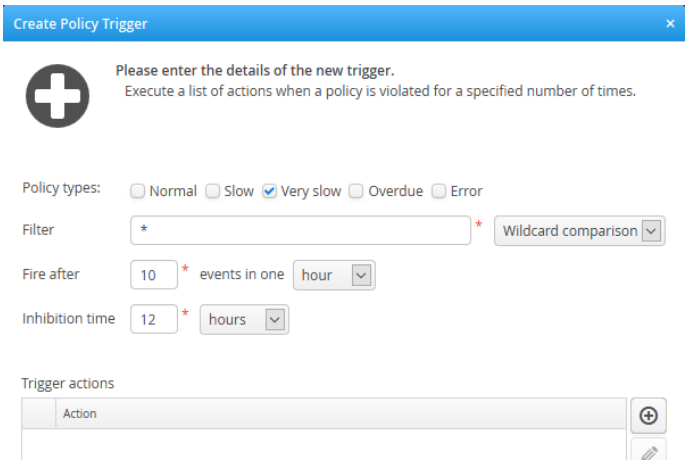
Trigger types

There are three types of triggers:

- **Policy triggers**

  Policy triggers are fired when the policy violations of a set of transactions exceed a defined number during a specified period of time. The filter text field takes a transaction name pattern, either a wildcard expression or a regular expression. Also, policy triggers are configured for particular policy violation types.

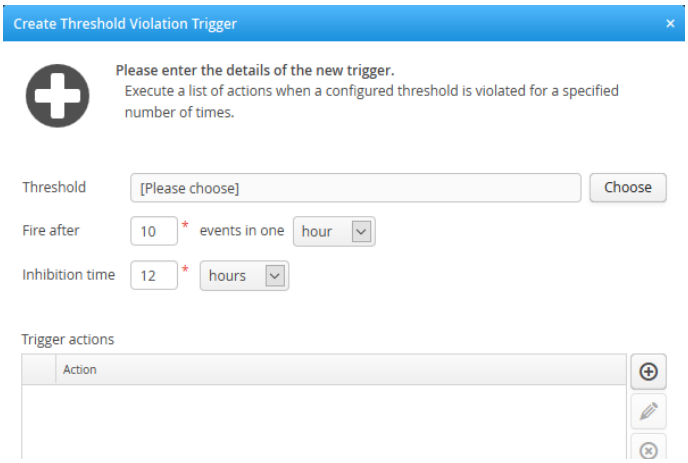
  You can have multiple policy triggers, each matching different transaction names. Unlike for transaction definitions, there is no name matching for policy triggers where only the first matching entry is used. If you add more than one trigger for the same transaction name, you will probably want to set different event rates, otherwise both triggers will be fired at the same time.



The trigger condition is decoupled from the actual condition of the policy violation. Various transactions can define different times after which a transaction is characterized as "very slow", the policy trigger then counts these events.

- **Threshold triggers**

  Threshold triggers are fired when the rate of threshold violations for a selected telemetry exceeds a configured value. This requires that you have configured at least one threshold [p. 57] for the same VM group.



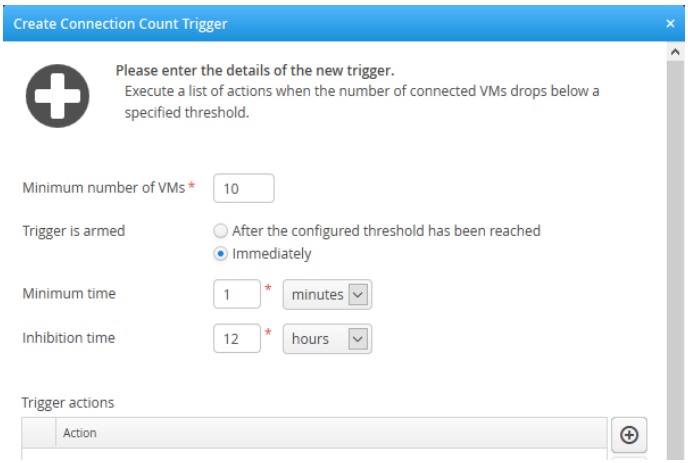
Like for the policy trigger, the trigger condition does not define an actual threshold. Thresholds can be defined differently in different descendant VM groups and the threshold trigger then counts threshold violations.

As an example, imagine you have two groups of machines, powerful machines and legacy machines. On a powerful machine, the number of threads may not exceed 1000, on a legacy machine that threshold is just 500. You would create VM groups named "Powerful" and "Legacy" and define the corresponding thresholds in the recording settings of each group as well as a default threshold in the "All VMs" group. Then, in the "All VMs" group, you would define a threshold trigger for the thread count telemetry. That trigger would handle both VM groups at the same time.

- **Connection trigger**

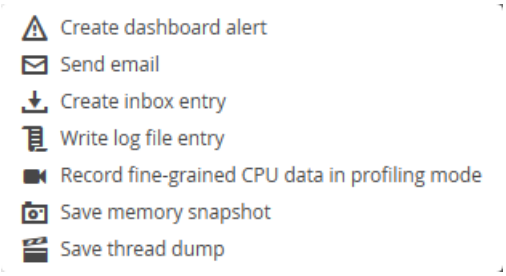
  The number of connected VMs is a scalar value that originates in the perfino collector and not in the monitored VMs. The connection trigger is intended to take action in the case that too

few VMs are running. The configured minimum number is not reached immediately when the perfino server is started, so you have the option of arming the trigger only after the minimum number has been reached for the first time, or after a fixed amount of time has passed.



Trigger actions

Each trigger can have an arbitrary list of actions.



The types of actions that can be added to a trigger can be ordered into two categories:

- **Notification actions**

  Notifications can be created for consumption inside perfino. Apart from the alerts mechanism [p. 65] , you can write to the log file or to the inbox. Alerts are visible in the dashboard and are saved as historical data. The inbox is maintained for each user and has an "unread status" on a per-user basis. Externally, you can send emails. For the latter, the SMTP configuration in the general settings has to be valid.

- **Data collection actions**

  As a reaction to anomalous conditions, you can decide to record more data for a detailed analysis. These data recording options are more intrusive than the regular perfino recording.

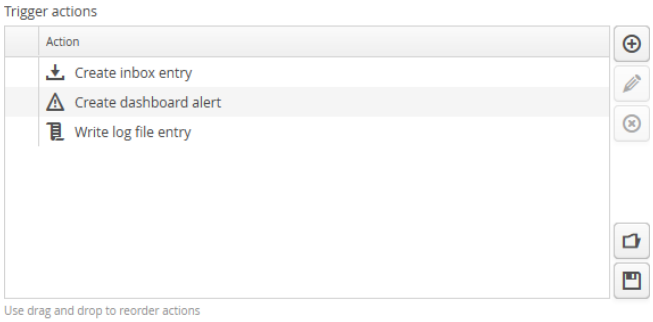
  For example. saving an **HPROF heap snapshot** should not be done on a regular basis, but if memory is low, it will help you to find a memory leak. HPROF snapshots are written directly by the JVM and do not require a native JVMTI profiling agent to be loaded. As such, this is a low-risk activity, although the VM is halted until the snapshot is saved. JProfiler and other Java profilers can open HPROF snapshots.

Thread dumps are a basic low-overhead way to inspect what is currently happening in a JVM. A policy trigger with event type "overdue" can save a thread dump, so you can immediately see in which method a transaction is hanging.

Recording data in profiling mode is an escalation in CPU data gathering that requires a native JVMTI agent to be loaded. The native perfino agent operates in a restricted mode that is optimized for minimum overhead and minimum risk. More information on this topic is available in the chapter oncross-over to profiling [p. 89] .

The data collection actions are also available in the "VMs view" when clicking on the "actions" link next to a connected VM.

The list of actions is executed in order. If one action fails, perfino jumps to next action and does not terminate the execution of the trigger actions.

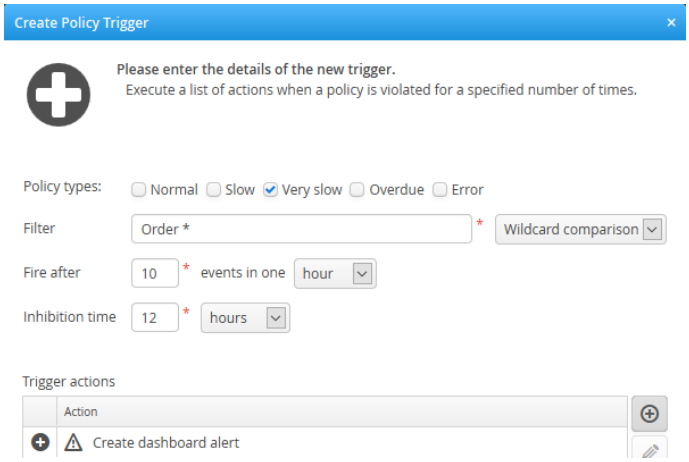


Alerts

Alerts communicate anomalous conditions to you and other users of the perfino UI server. By default, perfino does not create any alerts since the conditions that are noteworthy on such a level are highly individual.

Configuration

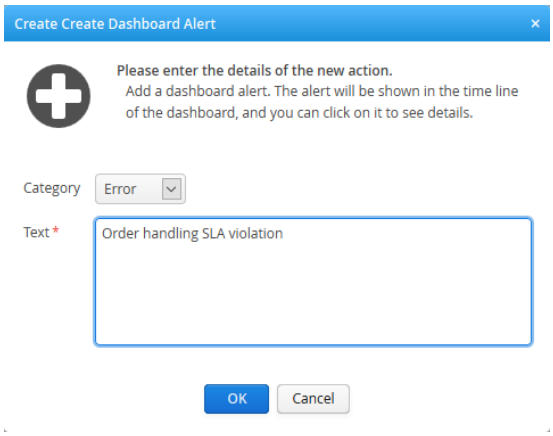
To create alerts, you have to add a trigger [p. 60] that fires for a specified policy or threshold violation. In the list of actions, add a **"Create dashboard alert" action**.



When configuring the "Create dashboard alert" action you can specify a category. This selection determines the color of the alert bars in the dashboard:

- **Error**
  - Red alert bars
- **Warning**
  - Orange alert bars
- **Info**
  - Green alert bars

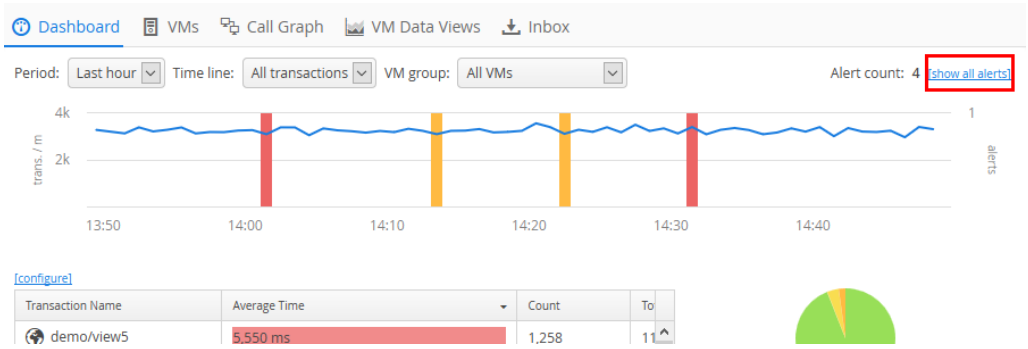
With the configured text message, you will be able to identify the origin of the alert in the data views.



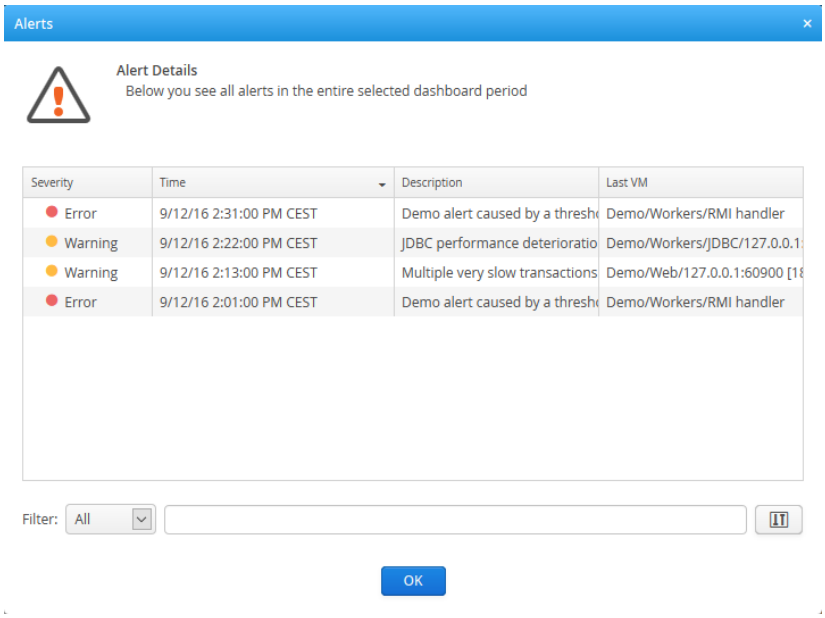
Data

Alerts are displayed in the dashboard. They are drawn as **alert bars** over the transaction timeline. In this way, you can quickly correlate an alert with an increase in the general activity of your application.

If there is more than one alert in the same time slot, just one alert bar is shown with a height proportional to the number of contained alerts. If those alerts are from different categories, the associated colors are shown stacked on top of each other. The tool tip shows the number of alerts in each category.



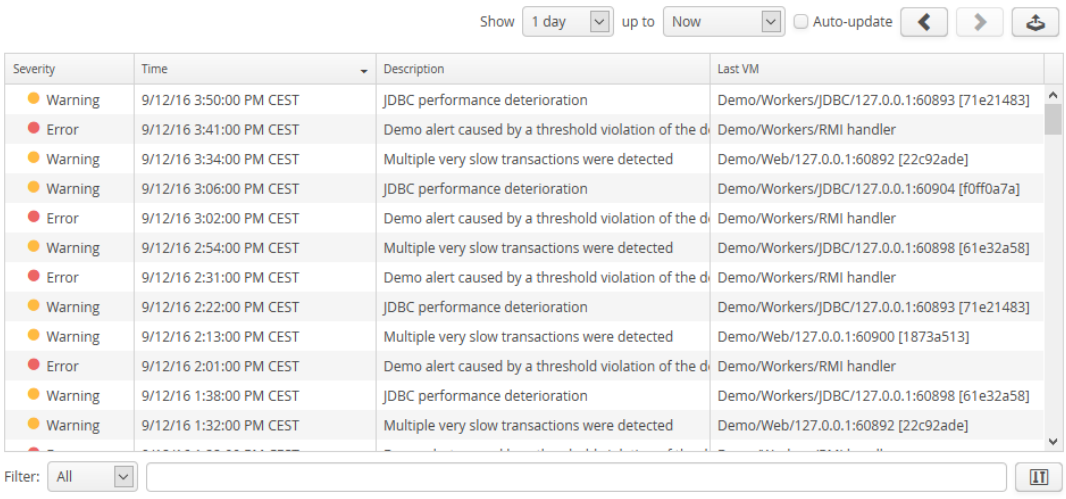
Clicking on the alert bar brings up the **alert detail dialog** where you can see exact times, categories and the text messages that were configured in the trigger action. To view all alerts in the time range that is currently displayed by the transaction time line, click on the "show all alerts" link in the top-right corner.



The "Last VM" column in the detail dialog shows the VM that was responsible for triggering the alert. Note that triggers always react to policy and threshold violations of entire VM groups, so multiple VMs may contribute to the condition that creates a single alert.

The dashboard only shows alerts from the selected period and from VMs that are recursively contained in the selected VM group. By changing the period or the VM group selection, you can limit the displayed alerts to your focus of interest.

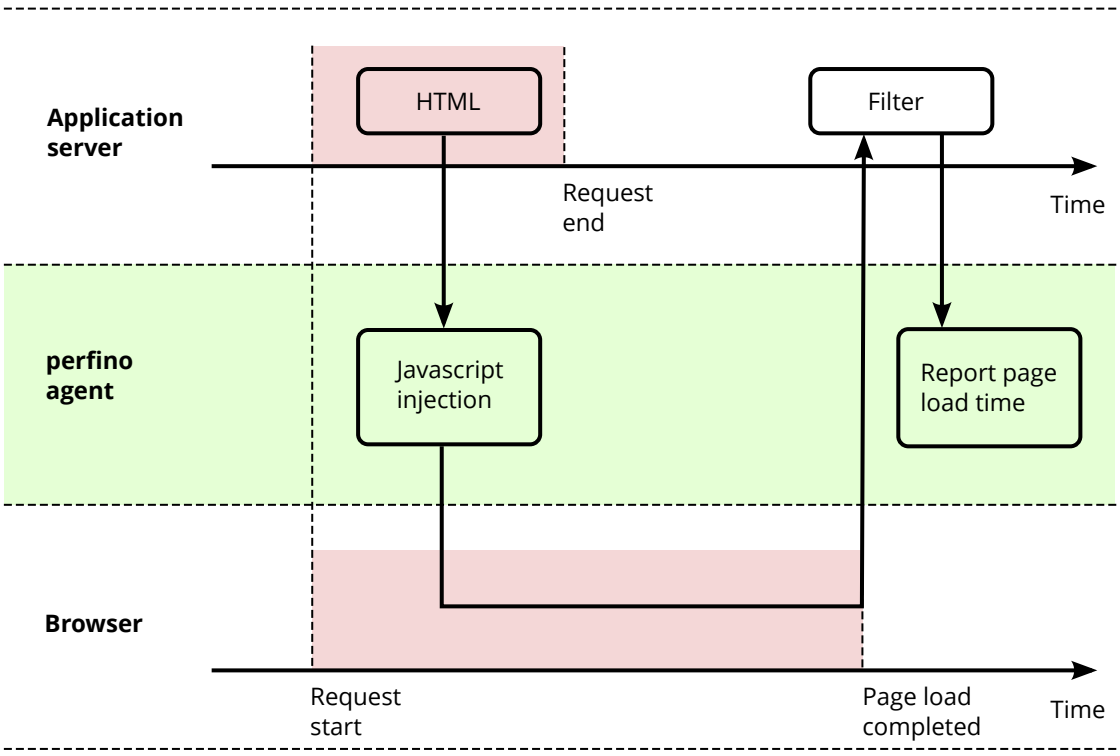
To analyze alerts in a historical context, go to the alerts view in the "VM data views". Here you can see alerts from previous days, weeks or months. If the "Auto-update" check box is selected, alerts are displayed as soon as they are generated by perfino. Also, you can inspect alerts from a single VM only, which is not possible in the dashboard.



End User Experience Monitoring

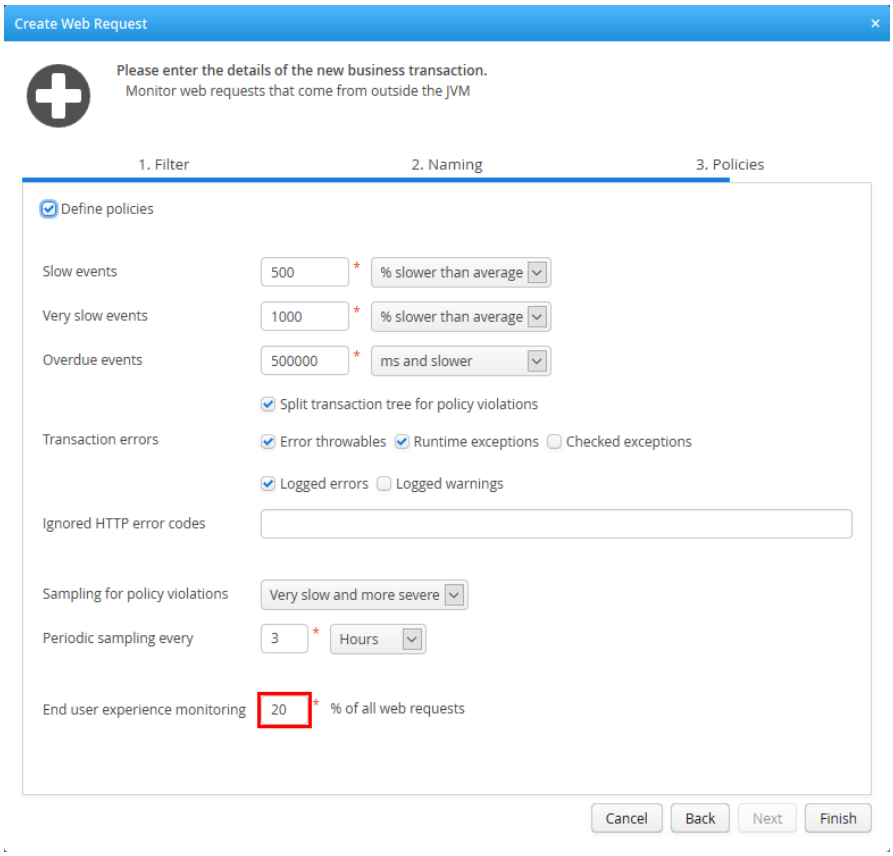
For web requests, the server transaction time is shorter than the page load time in the browser. Secondary requests that are triggered by the page can lead to an unacceptable end user experience even though your server monitoring indicates that everything is fine.

perfino includes a servlet filter that injects a Javascript snippet into HTML pages and reports the page load time back to the server. In addition, a transaction ID is sent back so that perfino can correlate execution times of web transactions and the associated page load times. The snippet is very small and does not require any external libraries. The overhead is also extremely small, since the snippet only runs after the page has loaded and the Web Performance API is used to get the timing directly from the browser.

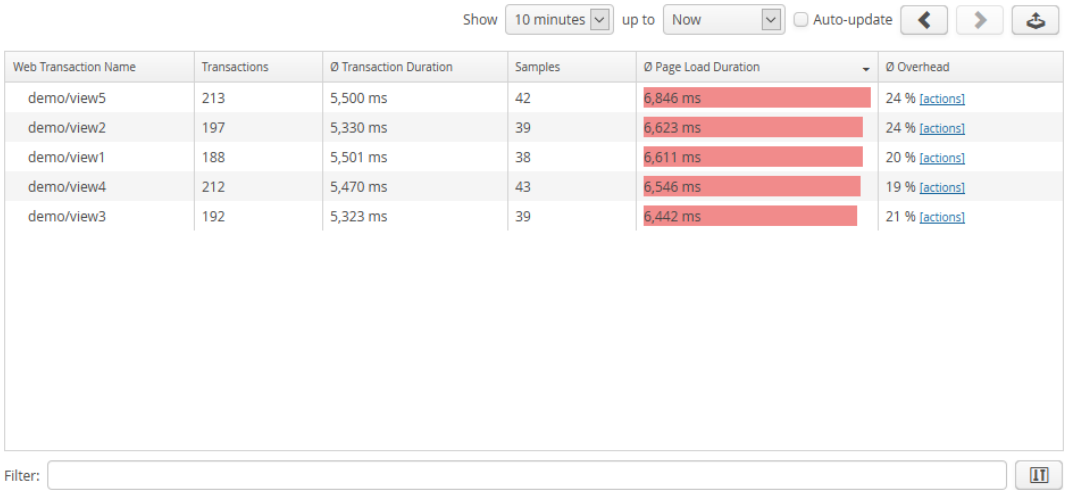


There are no cross origin request problems with this scheme, because the browser sends the data back to the application server where the page was loaded from. A servlet filter provided by perfino reads the timing data and reports it to the perfino monitoring agent. From there it is transmitted to the perfino collector.

Typically, you do not need page load times from all HTTP requests that create HTML pages, so you just sample a particular percentage. This percentage value can be configured in the policies configuration of web transactions. To disable this feature, set the percentage value to 0%.



In the VM data views, the "Transactions" view category holds the "End user monitoring" view. It is structured like the hot spots view with different time ranges and navigation buttons to show previous intervals. The table lists all web transactions for which end user monitoring data has been received by the perfino collector.



From the ratio of the page load duration and the transaction execution time, an **overhead** column is calculated. The "Samples" column shows you how many measurements were taken, while the "Transactions" column contains the total number of transactions. Their ratio should correspond to the sampling percentage above. It can be slightly lower due to incomplete page loads and

browsers that do not support the Web Performance API. To get better statistics, switch the time range to a longer interval.

Using the servlet filter

The servlet filter that injects the monitoring Javascript snippet into HTML pages is contained in the perfino API that is located in the JAR file `api/perfino_api.jar`. See the javadoc overview for how to download this JAR file with Maven, Ivy or Gradle.

To enable end user experience monitoring, you have to add a filter definition to your `web.xml` file:

```
<filter>
  <filter-name>euem</filter-name>
  <filter-class>com.perfino.filter.EndUserFilter</filter-class>
</filter>
```

In addition, you need at least one filter mapping that passes your HTML pages through this filter:

```
<filter-mapping>
   <filter-name>euem</filter-name>
   <url-pattern>/*</url-pattern>
</filter-mapping>
```

Only HTML files with a `head` tag are processed, other files are left unmodified. HTML pages that are not generated as part of a business transaction are not injected.

If you have filters that compress your pages, you have to make sure to insert the `EndUserFilter` after any such filters, so that it can detect and modify the HTML content.

The Javascript snippet is loaded from and reports back to URLs below `[context path]/__perfino`. You have to make sure that URLs matching `/__perfino/**` are passed to the filter. In the above example configuration that would be the case. If the filter mapping only handles selected URLs, the following additional filter mapping has to be added as the first mapping:

```
<filter-mapping>
   <filter-name>euem</filter-name>
   <url-pattern>/__perfino/*</url-pattern>
</filter-mapping>
```

Cached pages

If you use a filter to cache certain HTML pages, you might want to disable end user experience monitoring for those pages by adjusting the URL pattern for the `EndUserFilter`. If you really want to monitor cached pages, make sure that the sampling percentage for end user experience monitoring is set to 100%, otherwise not all cached pages will include the Javascript snippet.

Since the load time of the page will not be correlated to the transaction time on the server in that case, the average overhead may become negative. In that case, the perfino UI shows `[Cached]` instead of the negative percentage number. In the time lines for the overheads of a single transaction, negative overhead percentages due to caching are set to zero.

For cached pages, a dynamic change of the transaction naming configuration will not propagate to the browser. perfino uses a hashing technique to detect pages that were generated with a different configuration and ignores the reported load times.

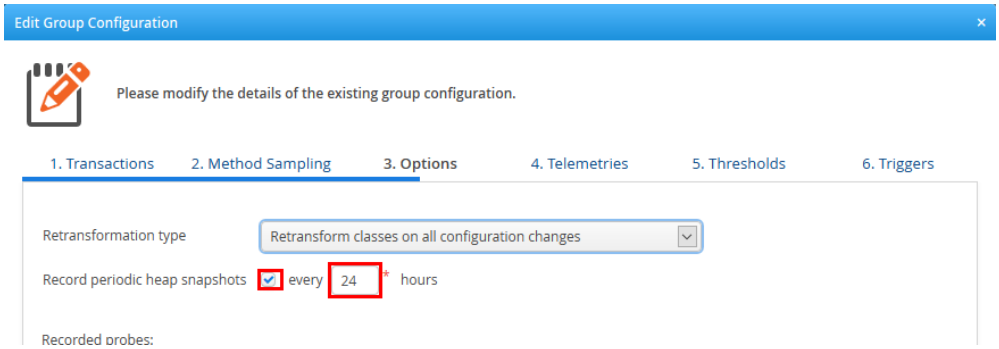
Memory

Memory analysis in perfino is always on a per-VM level. Two kinds of memory snapshots can be taken and analyzed: Low-overhead memory snapshots and HPROF snapshots.

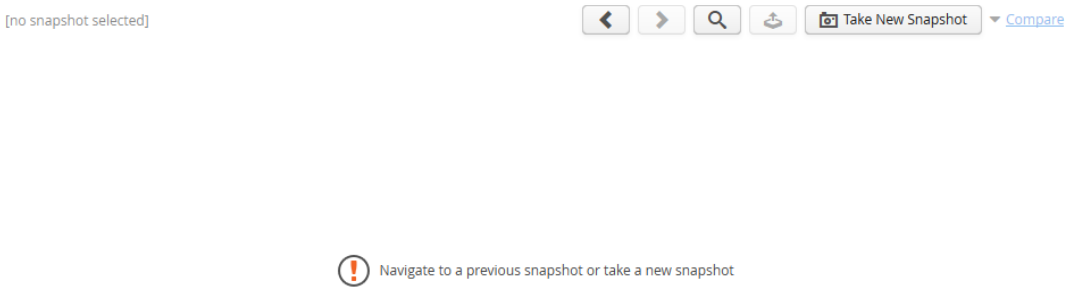
Memory snapshots

Low overhead memory monitoring is only possible at a "shallow" level. The JVM knows about loaded classes and instance counts and exposes this information. There is no kind of reference information available, which is important for solving memory leaks. Nevertheless, classes and instance counts give you a good initial overview of the memory consumption inside a JVM. For historical comparisons [p. 77], plotting the instance count of a particular class against time often shows important trends.

To get a basis for historical comparisons, memory snapshots are recorded once per day. This default can be changed on the "Recording" step of the VM group configuration. You can change the period for memory snapshot recording or disable it altogether.

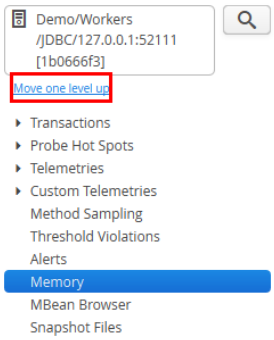


When you open the "Memory" view in the "VM data views", you are required to **select a single VM or a VM pool** with the VM selector in the top-left corner. After you make this selection, the navigation controls in the top right corner are enabled and you can move to the most recent snapshot with the ❮ Previous button.



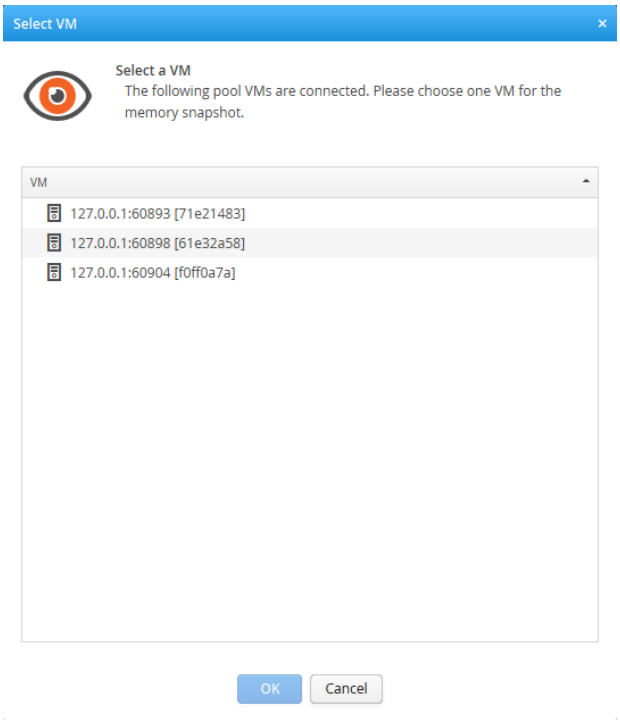
If you select a **VM pool**, you must 🔍 select a specific snapshot. In this way, you also narrow your focus to a particular VM from the VM pool and the VM selection will be set to that VM for all VM

data views. To move back to the entire pool, simply click on the "Move one level up" link below the VM selector.

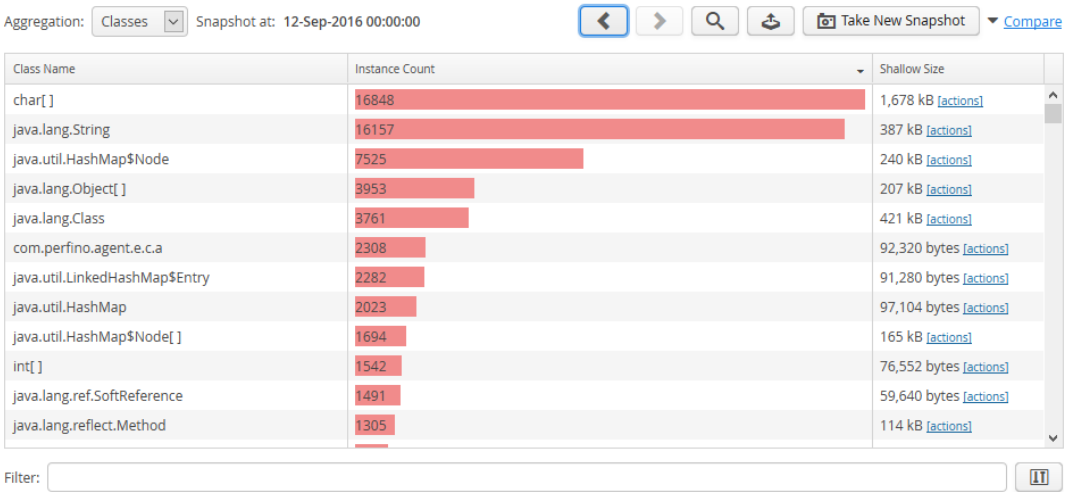


The record button takes a new snapshot and displays it immediately. Since periodic snapshots are taken quite rarely, this strategy may be necessary if you want to analyze a current situation.

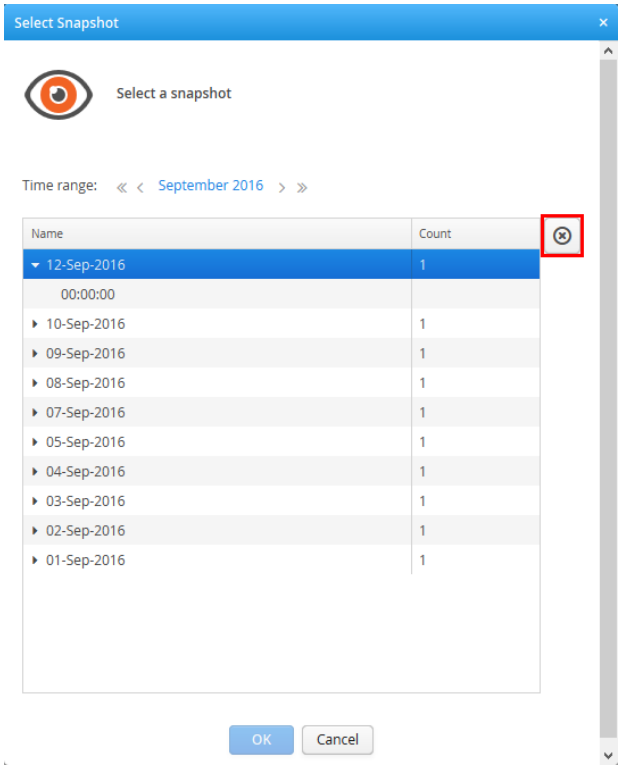
If you have selected a **VM pool**, perfino asks you for which VM the snapshot should be taken. Just as when selecting an existing snapshot for a VM pool, the VM selection is then changed to that particular VM.



A memory snapshot is a list of loaded classes together with their instance counts and shallow sizes. "Shallow" size means that only the direct size of the object is counted. This includes pointer sizes for referenced objects, but not the referenced objects themselves. Any other size calculations like deep size or retained size would require information about references which are not available with this low-overhead technique.



The 🔍 snapshot selector shows you all memory snapshots that are stored in perfino for the current VM selection. This is also the place to delete snapshots.



HPROF snapshots

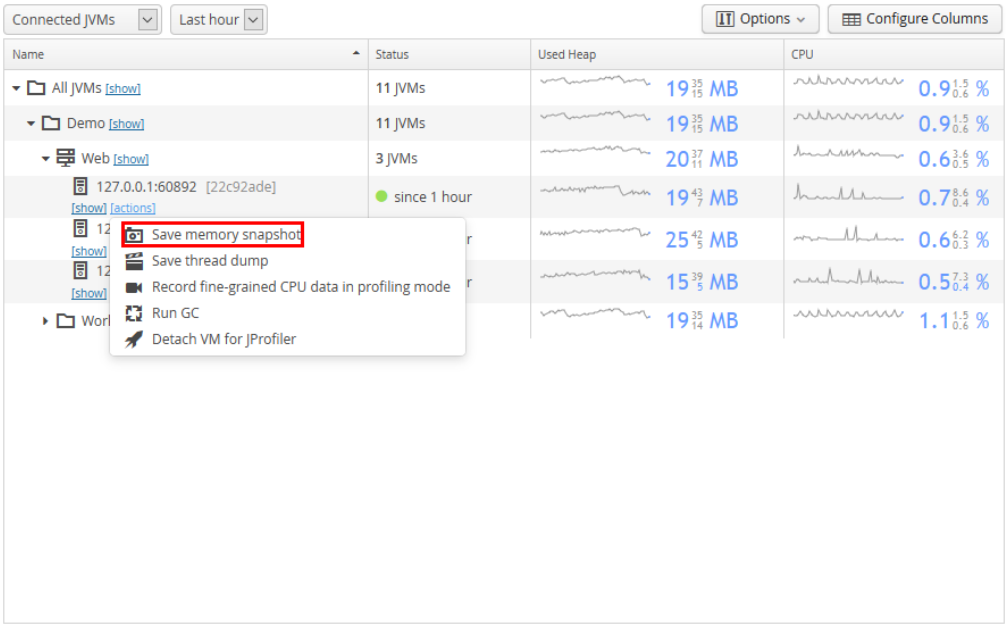
Advanced memory analysis that can give answers to questions like

- What is the cause for the memory leak in my application?
- My application needs a lot of memory, how can I reduce memory consumption?
- Where does this class loader leak come from?

require a full heap dump.

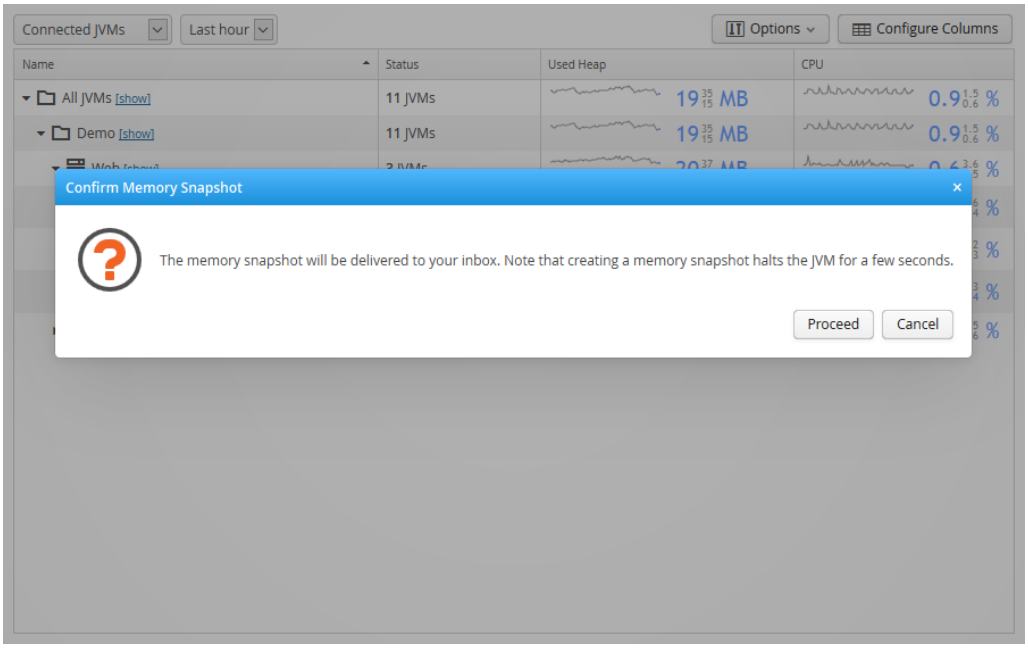
The JVM can save memory snapshots directly in a format called HPROF without the help of native JVMTI profiling agents. Saving HPROF snapshots is a low-risk operation, but during the time that such a snapshot is saved, the JVM is slowed down or temporarily halted.

You can save HPROF snapshots with triggers [p. 60] or manually in the VMs view, by clicking on the "actions" link next to a VM and selecting "Save a memory snapshot".



The snapshot will be delivered to your inbox when it is ready. This can take a few seconds. If the snapshot is large and the network connection between the perfino collector and the monitored VM has low bandwidth, it can also take several minutes. From the inbox, you can download the snapshot and load it with JProfiler or other Java profilers.

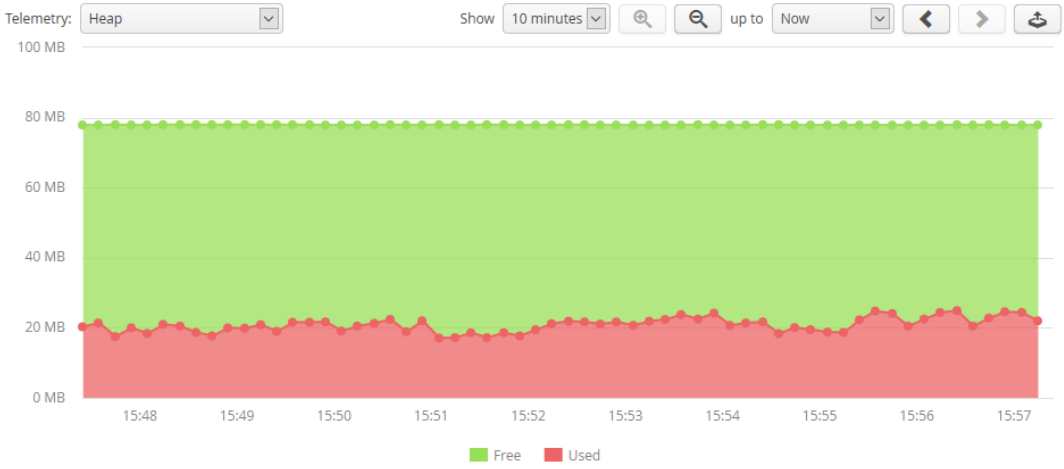
Note that if a sampling operation started with the "Record fine-grained CPU data in profiling mode" is currently in progress, the heap dump will only be taken after that operation has completed.



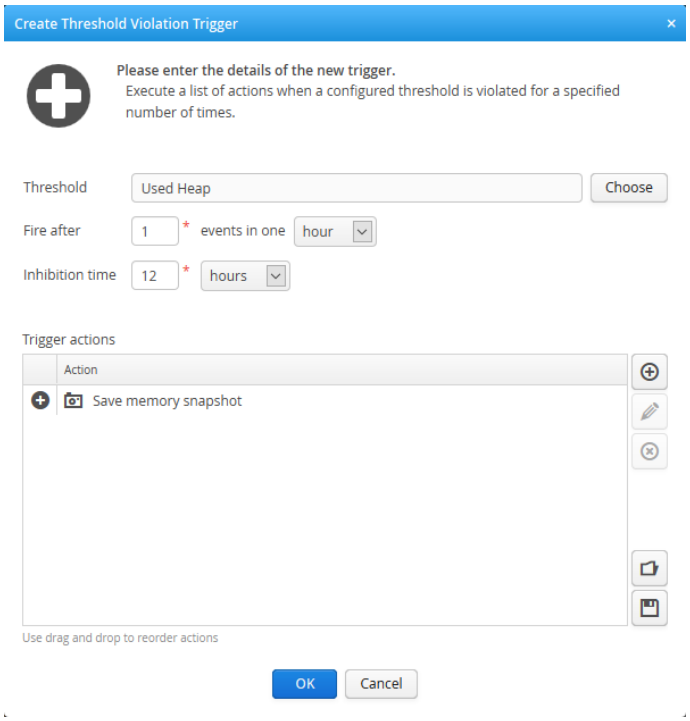
Snapshots are stored indefinitely until you delete them from the "Snapshots" view. The inbox item just notifies you about a new snapshot, deleting the inbox item will not delete the snapshot itself. In the "Snapshots" view you can select the VM group or VM to show only snapshots that have been taken for the selected VMs.

Heap telemetry

Information about free heap and used heap is always available with a fine time resolution. As such, they are an ideal basis for trigger conditions that save HPROF snapshots in case of low memory.



To do that, define a **threshold** for the "Free heap" telemetry with a specified lower bound in the recording settings. In the trigger settings, add a **threshold trigger** for that threshold and add a "Save a memory snapshot" action to the trigger.



perfino only takes a memory snapshot for the last VM where a threshold violation was detected, not for all such VMs.

Historical Comparisons

perfino stores data for long period of times, in some highly aggregated forms even indefinitely. Apart from telemetries where the main content of the view already is a historical comparison by itself, the VM data views show data from a certain time interval and let you move back and forth to adjacent intervals or select arbitrary intervals in the past.

In addition to looking at this historical data to analyze it by itself, many VM data views offer facilities to make historical comparisons. There are two kinds of comparisons: Comparing all the content from two selected points in time and comparing one scalar value for many points in time.

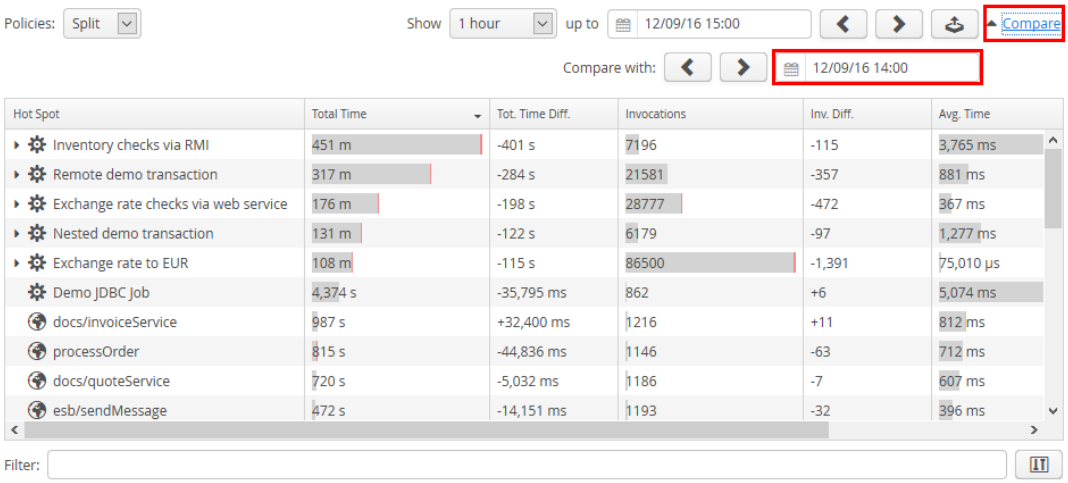
Content comparisons

Content comparisons are available in the views that show transactions as well as the probe hot spots views. To start a comparison, click on the "Compare" link in the top right corner. This immediately shows you a comparison between the following intervals:

- **Second interval**

  The second interval is the interval you were looking at before clicking on the "Compare" link. You can adjust the second interval with the same navigation controls as before.
- **First interval**

  The first interval is set to the interval just preceding the second interval. New navigation buttons are shown that let you adjust the first interval as required. With the date and time chooser you can jump back to arbitrary points in time.



Transaction data is aggregated, so as you move the first interval back in time, you will hit the limit of stored data. At that point you have to switch to a larger resolution for which data is retained longer. The available display intervals with their retention times are:

Display interval	Retention single VMs	Retention VM groups	Retention pooled VMs
1 minute	27 hours	27 hours	12 hours
10 minutes	27 hours	27 hours	12 hours
1 hour	30 days	10 days	10 days
1 day	unlimited	120 days	60 days

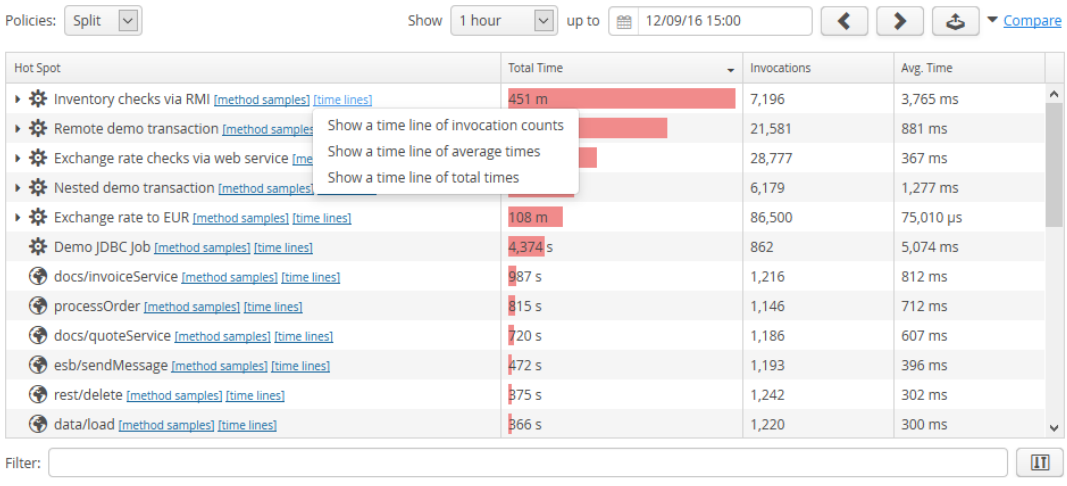
The retention times for VM groups are lower in order to reduce storage requirements. Analyzing single pooled VMs is only useful for fixing a problem and not suitable for long-term analysis, so the retention times for for pooled VM data are even lower.

In a comparison view, all measurements that are shown in the regular view get an additional column that shows the difference between the second and the first interval. In addition, the measurement columns are shown with a **difference bar** in the background:

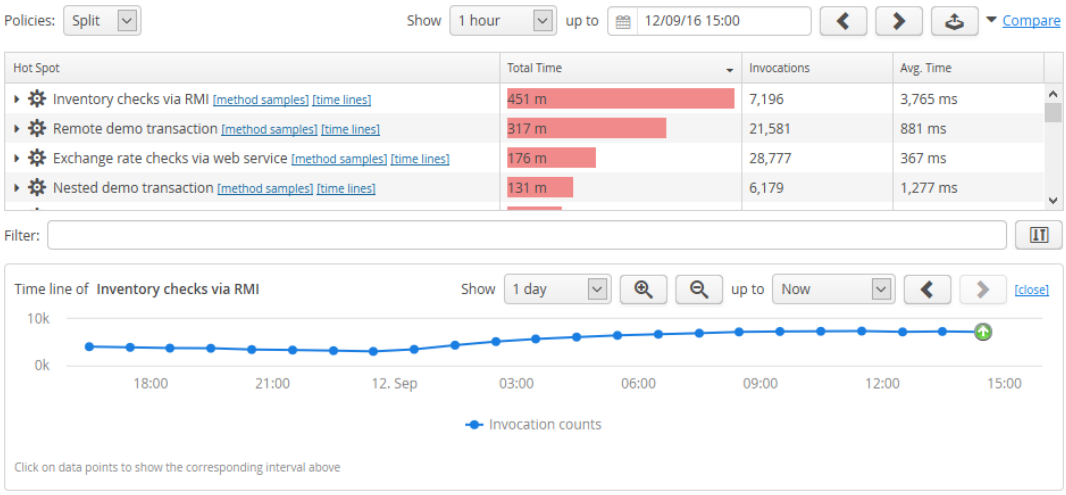
- If the value has **increased**, the total length of the bar corresponds to the value in the second interval, with the increase shown in **green**.
- If the value has **decreased**, the total length of the bar corresponds to the value in the first interval, with the decrease shown in **red**.
- The **unchanged fraction** is shown in **gray**.

Timelines

If you click on the "timelines" link next to an element in any of the transaction views, you can choose which column should be plotted over time.



The timeline will replace the default transaction telemetry in the split panel below the view. Clicking on the **[close]** button will restore the default state. Unlike the default transaction timeline that shows all transactions split into policy violation lines, these timelines are calculated from the transaction trees, and each point in such a timeline corresponds to one specific transaction interval. Instead of a highlighted time range, the currently selected transaction interval is shown as a special marker on the corresponding data point.

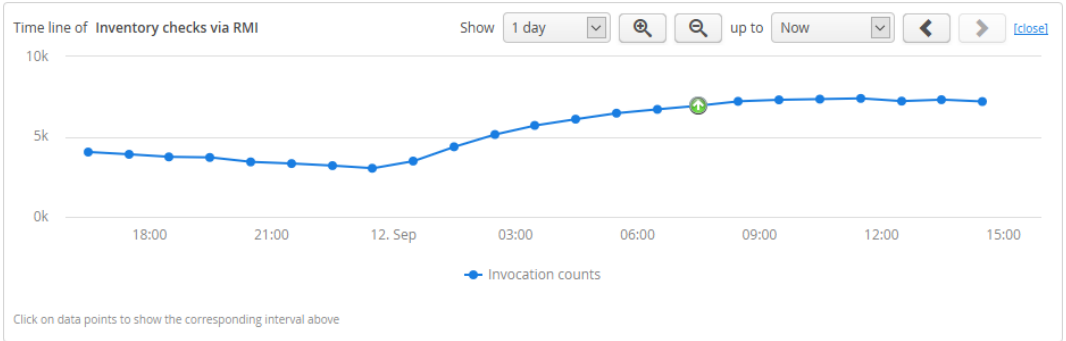


Time line data is not available for the one minute resolution, so the time line links are not shown for that display interval. For other display intervals, the data points in the time line initially match the display interval in the data view, so that the numbers are comparable. The timeline has navigation buttons itself which take you to different intervals of the entire timeline, just like for telemetry views.

The following display intervals in the data view and the time line are compatible with respect to the measured intervals:

Data view	Time line
10 minutes	3/6/12 hours
1 hour	1/3/6 days
1 day	12/30/60/180 days

To get the chronological context for the data displayed above, look for the up-arrow icon in the timeline. You can navigate to different points in time by clicking on them.



If you change the display interval of the time line and then click on a data point in the time line, the resolution of the data view will be adjusted to be compatible with the time line. Then, the current time can be marked.

If you select a new timeline in the data view, the old timeline will be replaced. Any change in the parameters of the data view will automatically close the timeline.

MBean Browser

Introduction

Many application servers and frameworks such as Apache Camel use JMX to expose a number of MBeans for configuration and monitoring purposes. The JVM itself also publishes a number of platform MXBeans that present interesting information around the low-level operations in the JVM.

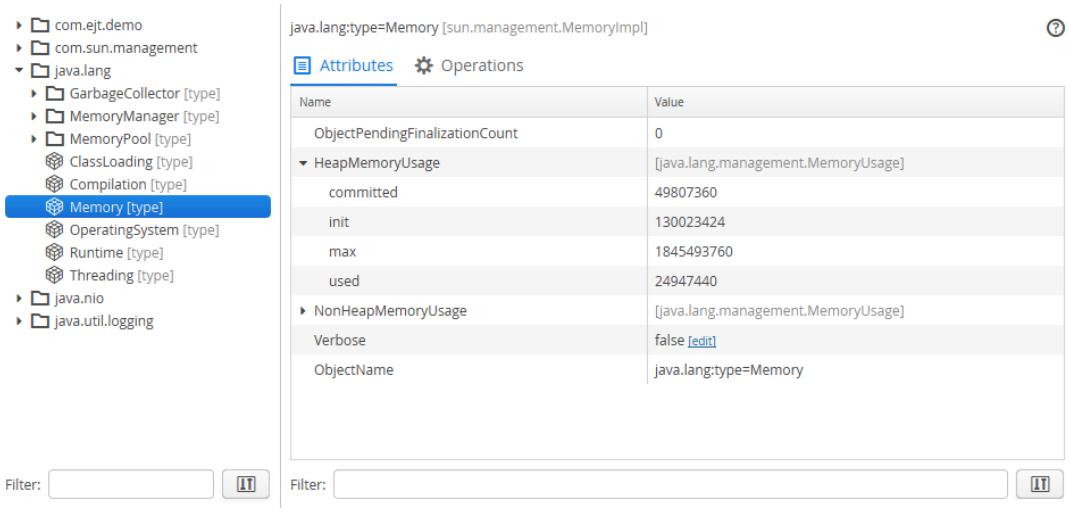
perfino includes an MBean browser that shows all registered MBeans in a selected VM. The remote management level of JMX for accessing MBean servers is not required, because the perfino agent is already running in-process and has access to all registered MBean servers.

perfino supports the type system of **Open MBeans**. Besides defining a number of simple types, Open MBeans can define complex data types that do not involve custom classes. Also, arrays and tables are available as data structures. With **MXBeans**, JMX offers an easy way to create Open MBeans automatically from Java classes. For example, the MBeans provided by the JVM are MXBeans.

The "MBean browser" view in the "VM Data Views" shows all registered MBeans in one selected JVM. If you have selected a VM group, you have to switch to a single VM first. If you have selected a VM pool, the context area will show an image button that will present a list of pool VMs.

Attributes

At the top level of the tree table showing the MBean content, you see the MBean attributes.



The following data structures are shown as nested rows:

- **Arrays**

  Elements of primitive arrays and object arrays are shown in nested rows with the index as the key name.
- **Composite data**

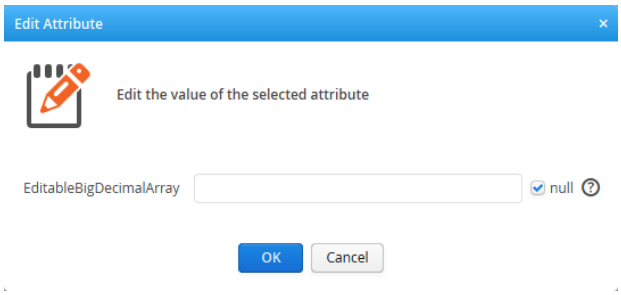
  All items in a composite data type are shown as nested rows. Each item can be an arbitrary type, so nesting can continue to an arbitrary depth.
- **Tabular data**

  Most frequently you will encounter tabular data in MXBeans where instances of `java.util.Map` are mapped to a tabular data type with one key column and one value

column. If the type of the key is a simple type, the map is shown "inline", and each key-value pair is shown as a nested row. If the key has a complex type, a level of "map entry" elements with nested key and value entries is inserted. This is also the case for the general tabular type with composite keys and multiple values.

Optionally, MBean attributes can be editable in which case an **[edit]** link will be displayed next to their value. MBean attributes can only be edited by a user with at least the "profiler" access level. Composite and tabular types cannot be edited in the MBean browser, but arrays or simple types are editable.

If a value is nullable, such as an array, the editor has a check box to choose the null state.



Array elements are separated by semicolons. One trailing semicolon can be ignored, so `1` and `1;` are equivalent. A missing value before a semicolon will be treated as a null value for object arrays. For string arrays, you can create empty elements with double quotes ("") and elements that contain semicolons by quoting the entire element. Double quotes in string elements must be doubled. For example, entering a string array value of

```
"Test";"";;"embedded "" quote";"A;B";;
```

creates the string array

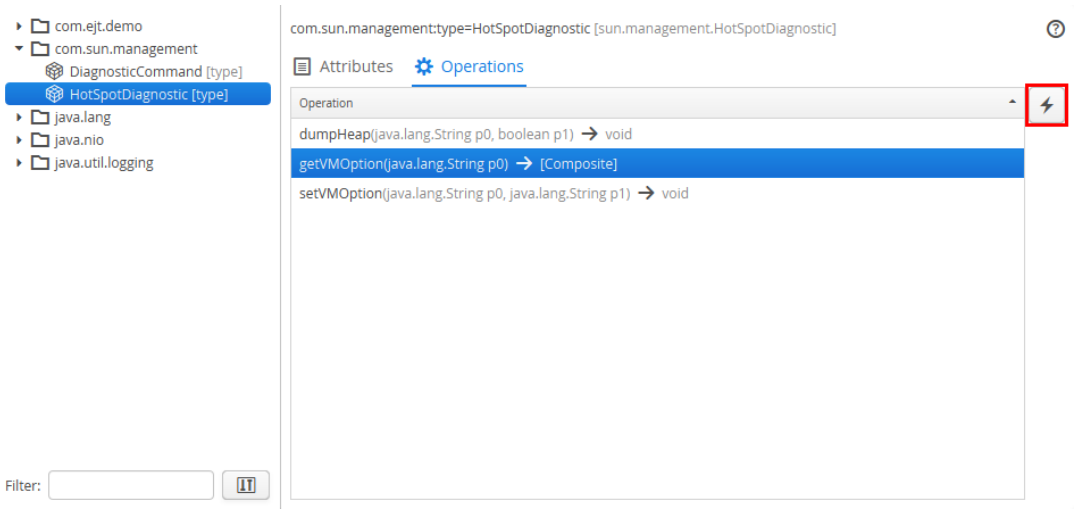
```
new String[] {"Test", "", null, "embedded \" quote", "A;B", null}
```

perfino can create **telemetries** from numeric MBean attribute values. When you define an MBean telemetry line [p. 50] in the recording settings, the **[Select]** button will bring up an MBean attribute browser with a VM selector on top. After you select a VM, you can choose an MBean attribute. Contrary to the MBean browser, the attribute rows are selectable in this case.

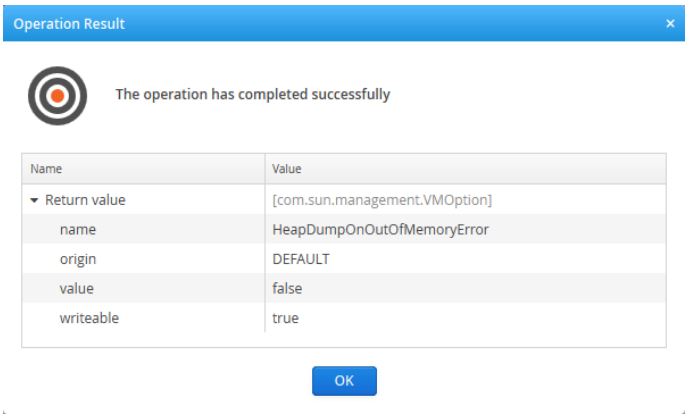
A telemetry can also track nested values in composite data or tabular data with simple keys and single values. When you chose the nested row, a value path is built where path components are separated by forward slashes.

Operations

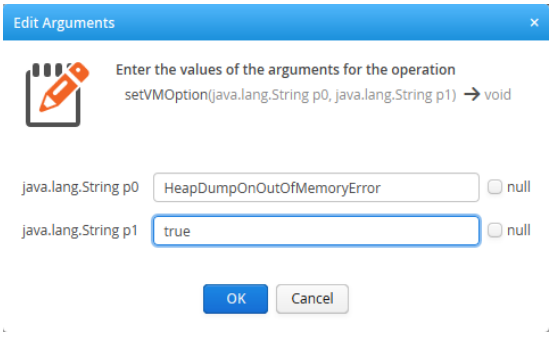
In addition to inspecting and modifying MBean attributes, you can invoke MBean operations and check their return values. MBean operations are methods on the MBean interface that are not setters or getters. To be able to invoke MBean operations, a user has to have at least the "profiler" access level.



The return value of an operation may have a composite, tabular or array type, so a new window with a content similar to the MBean attribute tree table is shown. For a simple return type, there is only one row named "Return value". For other types, the "Return value" is the root element into which the result is added.



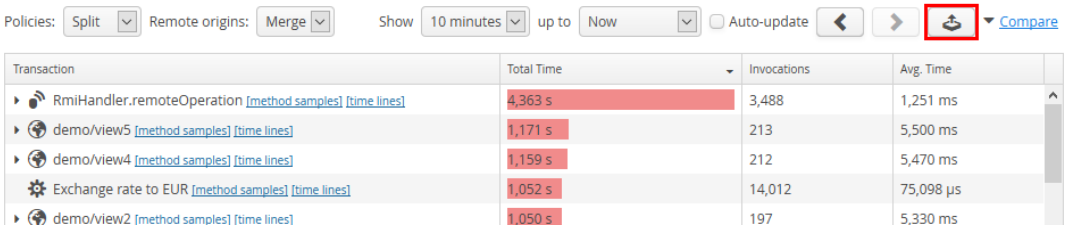
MBean operations can have one or more arguments. When you enter them, the same rules and restrictions apply as when editing an MBean attribute.



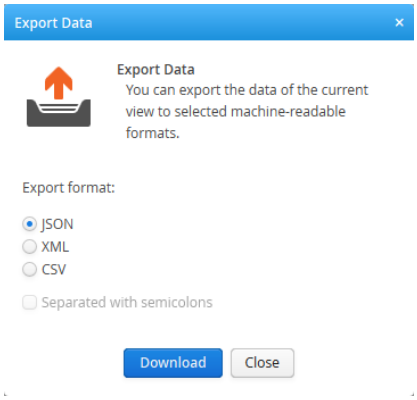
REST Export API

Introduction

In the perfino UI, the data views have an export button that allows you to extract the displayed data in a machine-readable format for further processing.



Depending on the type of data, the supported formats are JSON, XML and CSV.



To automate external data analysis, as well as to hook up perfino to other monitoring systems, this manual export is impractical. In that case, you can use the REST export API instead.

By default, the REST API service is not made available. You can enable it by setting the "restApi" property [p. 96] in `perfino.properties` to a non-zero value. The protocol on the configured port is the same as that of the web server, i.e. either HTTPS or HTTP.

Once enabled you can make HTTP(S) calls to the configured port to retrieve recorded data. Just like in the web UI, you can request data for single VMs, or ask for cumulated data for a particular VM group.

Using the API

Access to the API is protected with basic HTTP authentication. This means that the password will only be encrypted when using HTTPS as the protocol. Since the API does not make any modifications, any configured user regardless of the access mode can export data with the REST API.

The returned format depends on the "Accept" header of the HTTP request. The following mime types are supported:

- **text/plain**

  The output will be plain text. If multiple columns are available, CSV data is written. For hierarchical data, only the top-level will be exported. The separator is a comma by default,

but you can change this with the `csvSeparator` URL query parameter. The `winLineBreak` URL query parameter changes the default line feed from LF to CR+LF.

- **application/json**

  The output will be in JSON format. Use `application/json;charset=UTF-8` to ensure that the output is in UTF-8 encoding regardless of other accept headers.

- **application/xml**

  The output will be in XML format.

A call to the REST API consists of one or more URL segments, followed by a list of query parameters. For example, if the configured API port is 8500, a call to

```
https://localhost:8500/groups
```

lists all groups that are configured in perfino. The URL

```
https://localhost:8500/transactions/callTree?group=Demo%2FWeb&interval=10min
```

retrieves the call tree data for the VM group "Demo/Web" for the last 10 minutes. Note the URL-encoded forward slash in the group name.

All start and end times can be specified in milliseconds between the current time and midnight, January 1, 1970 UTC or in one of the following formats:

Format	Example	Description

`yyyy-MM-dd'T'HH:mm:ss.SSS`
`yyyy-MM-dd'T'HH:mm:ss`
`yyyy-MM-dd`
	2016-03-02T22:40:00.000
	2016-03-02T22:40:00
	2016-03-02
		Date and time in the local time of the server. All shortened versions will be equal to providing zeros.

`yyyy-MM-dd'T'HH:mm:ss.SSS'Z'`
`yyyy-MM-dd'T'HH:mm:ss'Z'`
`yyyy-MM-dd'Z'`
	2016-03-02T22:40:00.000Z
	2016-03-02T22:40:00Z
	2016-03-02Z
		Date and time in UTC. All shortened versions will be equal to providing zeros.

API documentation

The following URLs are available:

- **/groups**

  Returns a list of all VM groups. The group hierarchy separator is a forward slash. All nodes in the tree table of the recording options are returned in a breadth-first manner. In XML and JSON, the "pool" attribute shows if the group is a VM pool or not.

- **/vms**

  Returns a list of VMs. The names include the hierarchical group path as returned by the `/groups` URL. Individual pool VMs are not returned.

Query Parameter	Description
group	A specific group the VMs should be listed from. If unspecified, all VMs are returned.

Query Parameter	Description
connected	If set to `true`, only currently connected VMs are returned.

- **/telemetries**

  Returns a list of all available telemetry types, to be used in the URL below.

- **/telemetries/{telemetryType}**

  Returns the specified telemetry data. The values of `{telemetryType}` must be one of the values that is returned by the `/telemetries` URL above.

Query Parameter	Description
interval	A telemetry interval. Possible values are `10min`, `3h`, `3d`, `30d`
startTime endTime	You can specify a start or an end time in addition to the interval. For time formats, please see above. If left out, the current time will be used as the end time.
vm group	You can specify a vm name or a group name. If left out, all VMs will be used.
pretty	If set to `true`, JSON and XML output will be pretty printed
csvSeparator	If text/plain is requested, you can specify a custom separator char
winLineBreak	If set to `true` and if text/plain is requested, CR+LF line breaks will be written instead of LF line breaks.

- **/transactions/{dataType}**

  Returns the specified transaction data. Possible values of `{dataType}` are `callTree`, `hotSpots`, `overdue` and `endUser`.

Query Parameter	Description
interval	A transaction interval. Possible values are `1min`, `10min`, `1h`, `1d`
startTime endTime	You can specify a start or an end time in addition to the interval. For time formats, please see above. If left out, the current time will be used as the end time.
vm group	You can specify a vm name or a group name. If left out, all VMs will be used.
mergePolicies	If set to `true`, policies will be merged for the data types `callTree` and `hotSpots`.
removeOrigins	If set to `false`, origins will be shown for data type `callTree`.

Query Parameter	Description
pretty	If set to `true`, JSON and XML output will be pretty printed
csvSeparator	If text/plain is requested, you can specify a custom separator char
winLineBreak	If set to `true` and if text/plain is requested, CR+LF line breaks will be written instead of LF line breaks.

- **/probeHotSpots**

  Returns a list of all available probe types to be used in the URL below.
- **/probeHotSpots/{probeType}**

  Returns the specified probe hotspot data. The values of `{probeType}` must be one of the values that is returned by the `/probeHotSpots` URL above.

Query Parameter	Description
interval	A transaction interval. Possible values are `1min`, `10min`, `1h`, `1d`
startTime endTime	You can specify a start or an end time in addition to the interval. For time formats, please see above. If left out, the current time will be used as the end time.
vm group	You can specify a vm name or a group name. If left out, all VMs will be used.
mergePolicies	If set to `true`, policies will be merged.
pretty	If set to `true`, JSON and XML output will be pretty printed
csvSeparator	If text/plain is requested, you can specify a custom separator char
winLineBreak	If set to `true` and if text/plain is requested, CR+LF line breaks will be written instead of LF line breaks.

- **/alerts**

  Returns the specified list of alerts.

Query Parameter	Description
startTime endTime	You can specify a start and an end time. If one is left out one day will be exported. If both are left out the current time will be used as the end time.
group	You can specify a group name. If left out, all VMs will be used.
pretty	If set to `true`, JSON and XML output will be pretty printed
csvSeparator	If text/plain is requested, you can specify a custom separator char

Query Parameter	Description
winLineBreak	If set to `true` and if text/plain is requested, CR+LF line breaks will be written instead of LF line breaks.

- **/violations**

  Returns the specified threshold violation data.

Query Parameter	Description
startTime endTime	You can specify a start and an end time. If one is left out one day will be exported. If both are left out the current time will be used as the end time.
vm group	You can specify a vm name or a group name. If left out, all VMs will be used.
pretty	If set to `true`, JSON and XML output will be pretty printed
csvSeparator	If text/plain is requested, you can specify a custom separator char
winLineBreak	If set to `true` and if text/plain is requested, CR+LF line breaks will be written instead of LF line breaks.

- **/triggerBackup**

  Triggers a backup of the database. The backup files are written to the `backup` directory inside the perfino data directory. You need to authenticate with an admin user in order to be able to use this URL.

  The return value of this call is the absolute directory path of the backup directory.

  To restore such a backup, stop the perfino server, replace the contents of the "db" directory with the contents of the "backup" directory and start the perfino server again.

  If the REST API is impractical to use, or if you have not activated it for your perfino installation, you can also create a file named `trigger_backup` in the perfino data directory. After the backup has been completed successfully, the file will be deleted and the backup will be performed.

Cross-over To Profiling

There are several reasons why you should not have a profiler running in production at all times. For one, there is the overhead that may be unacceptably high depending on the profiling settings. Profilers are geared towards maximizing the extraction of useful information with no explicit guarantees as for the incurred overhead. Also, the use of the native profiling interface of the JVM (JVMTI) is something that is an additional risk in a production environment. Depending on whether you use non-standard garbage collectors or other JVM tuning options, there may be stability concerns, since the JVM is not tested as extensively with JVMTI as it is without.

However, the best defense against performance problems is defence in depth. Sometimes there are situations that require more information than what can be obtained from the low overhead monitoring and sampling techniques that are available in perfino.

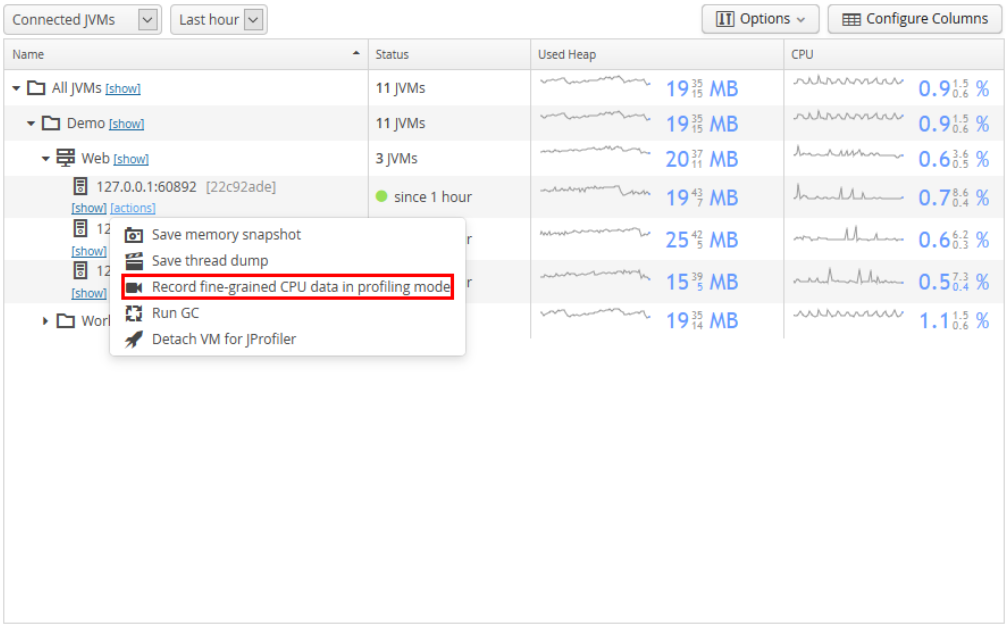
Recording CPU snapshots

For these cases, perfino offers a **full sampling mode that loads a native JVMTI library** that is optimized for in-production use. The result of this operation is a JProfiler snapshot file that can be downloaded from perfino and opened in JProfiler - similarly to the HPROF memory snapshot files [p. 71] that are saved directly by the JVM.

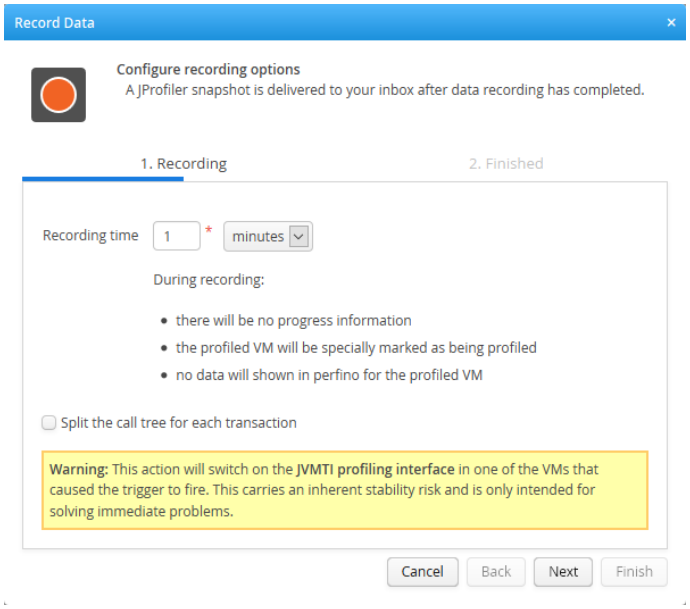
This profiling mode is only available if the correct native libraries are present in the `lib/[platform]` directory next to the `perfino.jar` file that was used in the `-javaagent` VM parameter. If you download the agent [p. 10] from the perfino UI, native libraries for all supported platforms are already included. If you copy the agent yourself from the perfino installation directory, copy the `agent/lib` directory along with it.

Once the JVMTI has been turned on, it cannot be turned off again. As long as the JVM is running, it will remain in this state. Typically, the overhead of the JVMTI without any data recording is less than 1%, though.

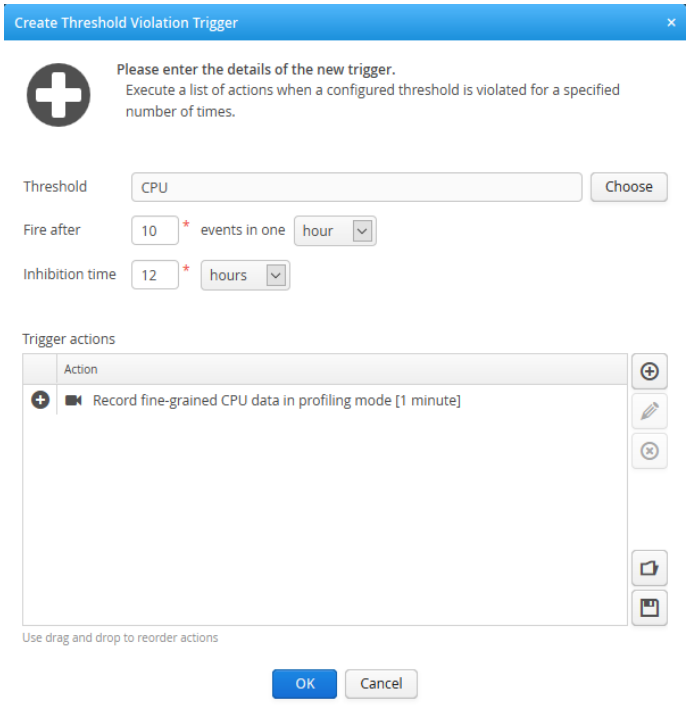
To take a CPU snapshot of a particular VM, go to the VMs view, click on the "actions" link next to the VM and select "Record fine-grained CPU data in profiling mode".



The duration of CPU recording can be configured. perfino monitoring is not impacted by CPU sampling. Also, you can specify if the call tree should be split for each transaction or not. This is a top-level split and the sampling call tree is appended to the perfino transaction tree (see below). Whether this split is beneficial or not depends on your transaction definitions and what kind of problem you want to find. The overhead for sampling with transaction splitting is higher than without.



To take snapshots automatically if the CPU load is too high, you can set up a threshold for the CPU telemetry and configure a trigger that includes the "Record fine-grained CPU data in profiling mode" trigger action.



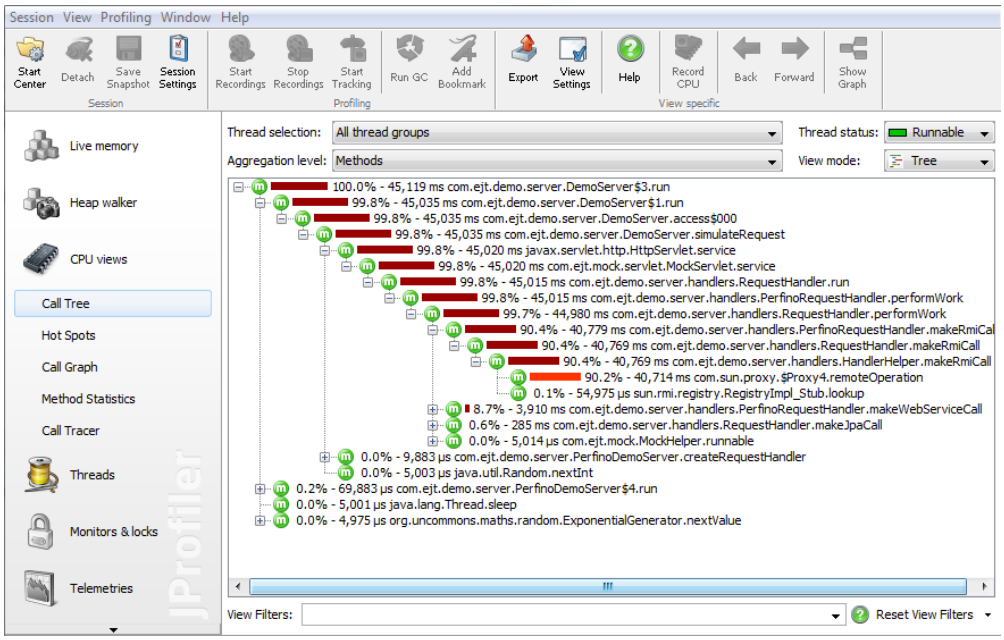
Note that only the last offending VM will be profiled, not all of them.

If you use a custom library for network I/O, you can add selected methods to the net I/O thread state [p. 114] , so that you get useful hot spots for your analysis.

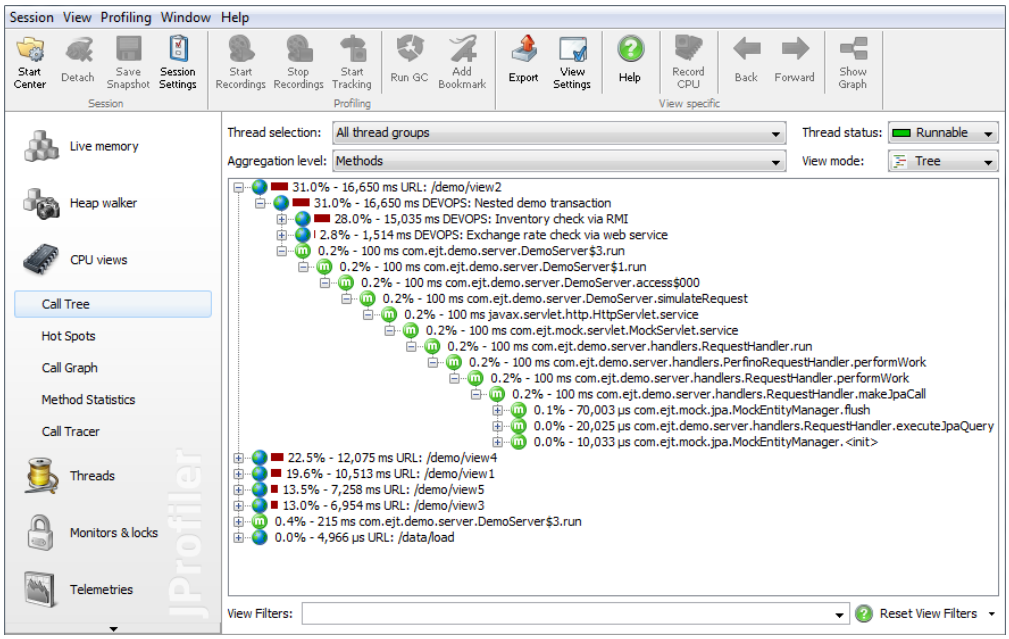
Viewing CPU snapshots in JProfiler

CPU snapshots are delivered to your perfino inbox from where you can download them. The file extension is ".jps" which stands for "JProfiler snapshot". If you have JProfiler installed, you can double-click on the snapshot file to open it, or choose `Session->Open Snapshot` from JProfiler's main menu.

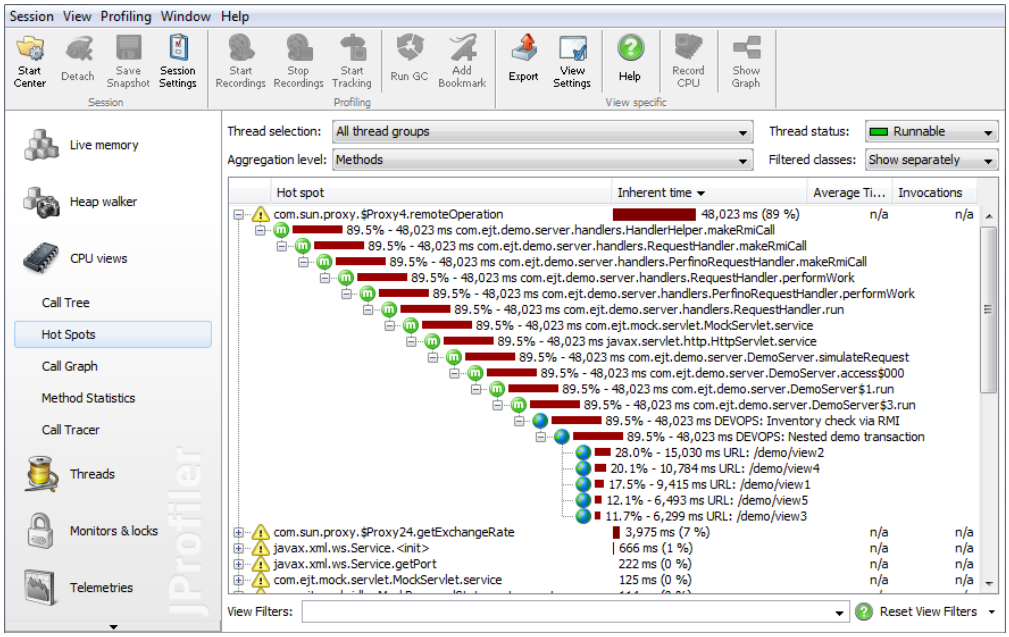
CPU snapshots taken by perfino only contain data for the call tree, hot spots and call graph views as well as the thread history and thread monitor views. Other views are disabled when you open a perfino CPU snapshot.



If transaction splitting was enabled for CPU recording, the perfino call tree is shown at the top and the sampling data is attached as appropriate.

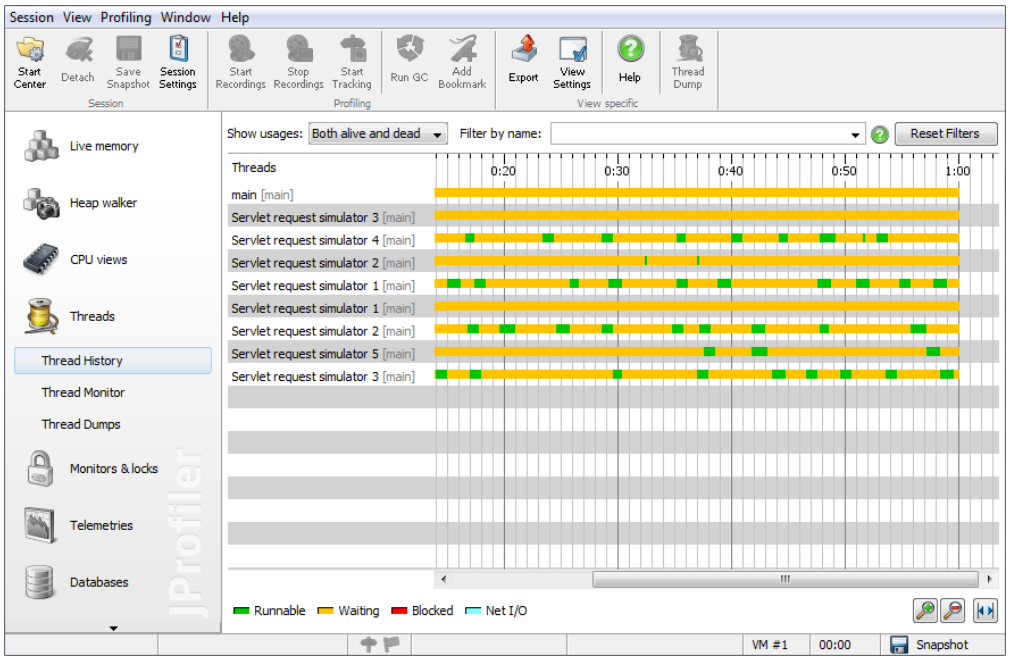


The back traces in the hot spots view also show the perfino transactions in this case.



The attribution of sampling data to transactions is not exact, but only an estimation based on the analysis of subsequent sampling call stacks. For very short running transactions (< 5ms), sampling data may be attributed to another transaction or even be placed outside any transaction.

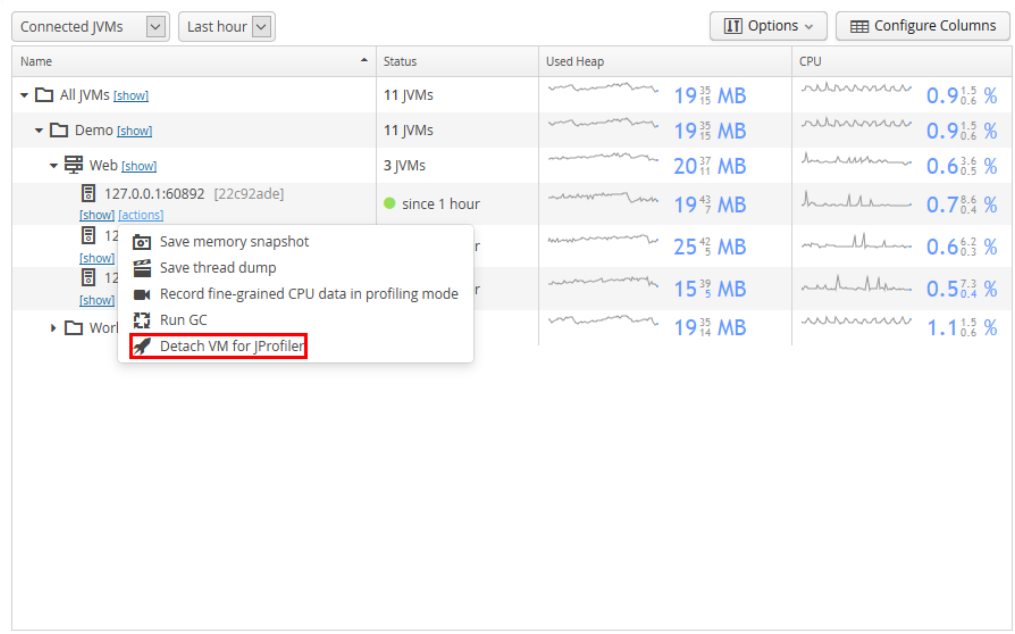
In the thread history view, you can check when threads were created or terminated and what kind of activity they performed over their recorded lifetime. Only green areas indicate times when the thread was actually eligible to execute methods. Other thread statuses designate idle time while waiting or blocking on monitors or reading or writing to network sockets.



Full profiling

Sometimes performance or memory problems are so tricky that you need the entire arsenal of recording techniques that a profiler has to offer. For example, the capabilities of the heap walker for live heap snapshots can be instrumental for solving a memory leak or the CPU and probe profiling features may be required for understanding a performance problem. Questions involving threading issues, monitors and locks can only be solved in JProfiler and not in perfino.

perfino helps you to cross over to full profiling with the minimum amount of intrusion into your production environment. If a JProfiler installation can be found, perfino can load its agent into a monitored VM and prepare it for a connection from the JProfiler GUI. Use the "Detach VM for JProfiler" action in the VMs view to initiate this process. The VM will be detached from perfino and has to be restarted after profiling to reconnect to perfino.



To take advantage of this integration, you have to download the JProfiler archive and extract it in one of the following directories:

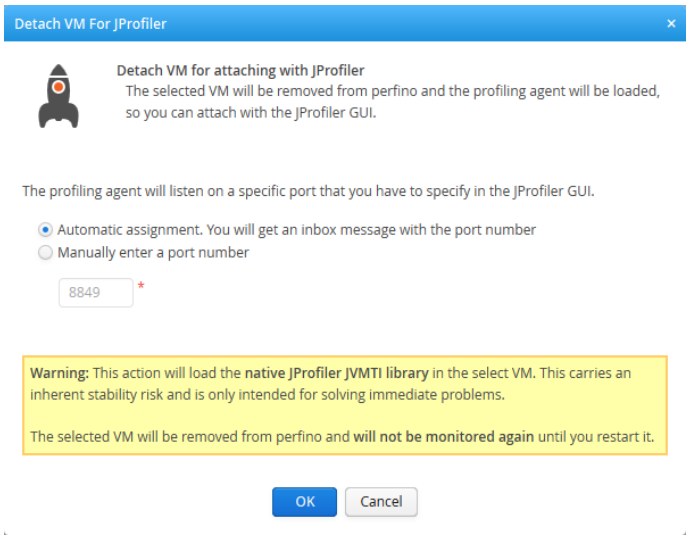
- The directory where the `perfino.jar` file has been installed
- The directory `$HOME/.perfino` on Linux/Unix or `%USERPROFILE%\.perfino` on Windows, for the user who is running the monitored VM.

The JProfiler archives from the download page contain a single top-level directory named `jprofiler[major version]`. The perfino agent sequentially looks for such directories in the above locations and uses the directory with the highest version number. Any version of JProfiler starting from 8.0.6 can be used for this integration.

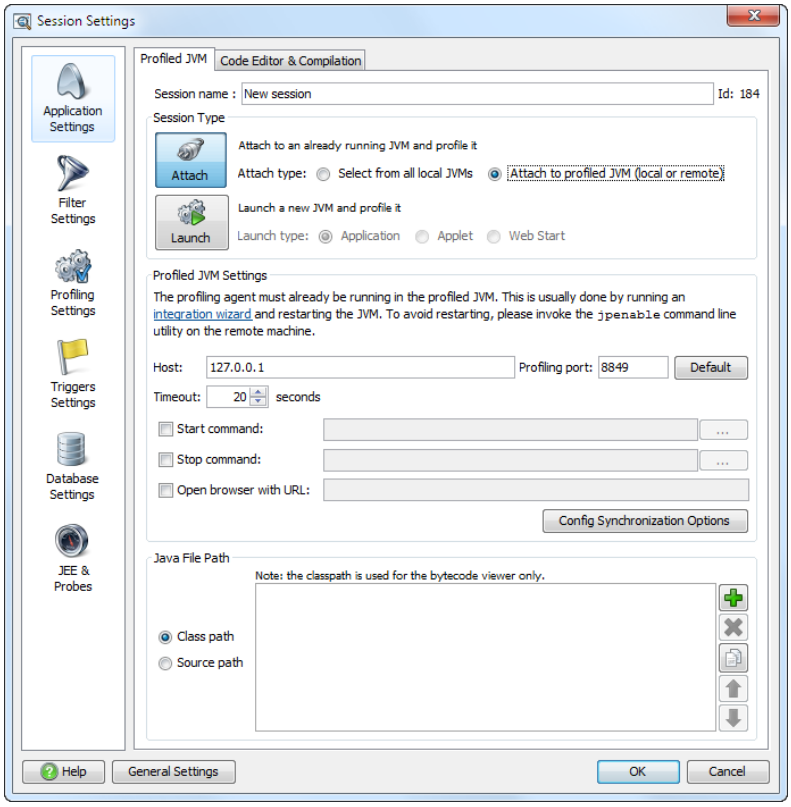
If none of the standard directories above are suitable or if you already have a JProfiler installation elsewhere, you can append the option `,jprofiler=[JProfiler installation directory]` to the VM parameter for monitoring [p. 10] . If the appropriate native library file can be found in that JProfiler installation, it will be used preferentially, otherwise the perfino agent will continue to look in the standard directories.

If no JProfiler installation can be found, specific instructions for integrating JProfiler are given. You can download and extract JProfiler on the machine where the monitored VM is running and execute the "Detach VM for JProfiler" action again, there is no need to restart the monitored VM.

If JProfiler was found, a confirmation dialog is shown, where you get the chance to configure the profiling port to which the JProfiler GUI will attach. You can either enter the desired port number yourself or let perfino find a free port for you. In the latter case, an inbox message will be sent to you with the actual port number.



In JProfiler, create a new session of type "Attach to profiled JVM (local or remote)" and enter the IP address or host name where the VM is running as well as the port on which the profiling agent is listening.



perfino will stop monitoring the VM when you prepare it for profiling. To monitor it again, you have to restart the JVM.

A Configuration

A.1 Server Configuration

perfino configuration options that cannot be changed in the perfino UI are contained in the text file `perfino.properties` in the perfino installation directory. You can either edit that text file in a text editor or use the `configure` executable in the perfino installation directory. The latter presents an organized view of all properties, saves the file even if elevated privileges are required and can restart the perfino server to apply your changes.

The properties themselves are documented with comments in `perfino.properties`. Here, a couple of scenarios are discussed where it is necessary to adjust the default parameters.

Data directory

The **dataDirectory** property points to the directory where all variable data is located. The following subdirectories are created by perfino:

- **db**

  Contains the embedded H2 database.
- **log**

  Contains all log files. By default, log files are rotated. The rotation settings can be changed in the `log4j.properties` file in the perfino installation directory. Different settings can be applied to the three different log files, "server", "connection" and "event". All log files can be viewed in the perfino UI.
- **snapshots**

  Memory snapshots and profiling snapshots are saved in this directory. These snapshots can be downloaded or deleted in the "Snapshots" view in perfino.
- **ssl**

  This directory contains the file pair `agent.ks` / `server.ks` for authentication and encryption as well as the SSL certificate for the web server.

If you run two perfino servers on the same machine, they have to have different data directories. By default the installer always suggests the same location for the data directory, so in the case of multiple installations you have to adjust it in the installer or after the installation in the perfino.properties file.

Web server

perfino comes with a built-in web server that listens on port 8020 by default. You can adjust that port with the **httpPort** property and switch to HTTPS by setting **useHttps** to `true`.

When you use HTTPS, perfino will generate a self-signed certificate `ssl/self_signed.ks` in the perfino data directory. Browsers will display warning messages with this certificate. If you have a certificate that is signed by a recognized certificate authority, you can copy it in PKCS12 format to `ssl/web.pkcs12`. If the certificate file has a different name, you can specify the **keystoreName** property.

If the certificate is protected with a password, you can specify it in the **keystorePassword** property. While the password cannot be encrypted, it can at least be obfuscated with the command line tool `perfino_obfuscate`:

```
perfino_obfuscate [password]
```

If you put perfino behind a reverse proxy, you have to set the **reverseProxy** property to `true`. The web server will then analyze the proxy headers to create correct URLs. If this should not work due to a problem with the reverse proxy, set the **reverseProxyHost** to the host name of the proxy.

If you have infrastructure that can check the health of a web server by making an HTTP GET request, you can set the **healthCheckPort** to a non-zero value in order to create such an HTTP port in perfino. Any HTTP request to that port will return a document with HTML mime type and the text "Alive". For example, Amazon Web Services provides a health check service that is used by Route 53 to determine if an IP address can be routed to or not.

The REST API service [p. 84] is enabled by setting the **apiPort** property to a non-zero value. You cannot set it to the same port as the web server. The REST API port uses the same protocol (HTTP/HTTPS) as configured for the web server.

Communication with monitored VMs

Monitored VMs create a TCP connection on the port that is configured with the **vmPort** property.

By default, the communication between monitored VMs and the perfino collector is unencrypted and unauthenticated. This means that every VM can connect to the perfino server and the perfino agent has no way of knowing if the perfino server on the other side can be trusted. This can be acceptable in certain local area networks but it is not suitable for connecting over WANs or even over the internet.

To enable authentication and encryption set the property **vmUseSsl** to `true`. In that case, the file pair `ssl/agent.ks` and `ssl/server.ks` will be created. Now, the server will only allow connections from agents who possess the `agent.ks` file and agents will only connect to servers who have the `server.ks` file. In addition the communication protocol will be encrypted.

For more information on this topic, see the chapter on monitoring JVMs [p. 10] .

Remote perfino UI

Running the perfino UI on a different server than the collector can have two purposes. First, it allows you to split the server load of the UI and the collector to separate machines which is a good idea if you have many users. Second, some network topologies require that the collector runs in one and the perfino UI in another network. For example, if the collector runs in an internal network that is protected by a dual firewall, and the perfino UI should be available to the outside, the perfino UI has to run on a machine in the DMZ.

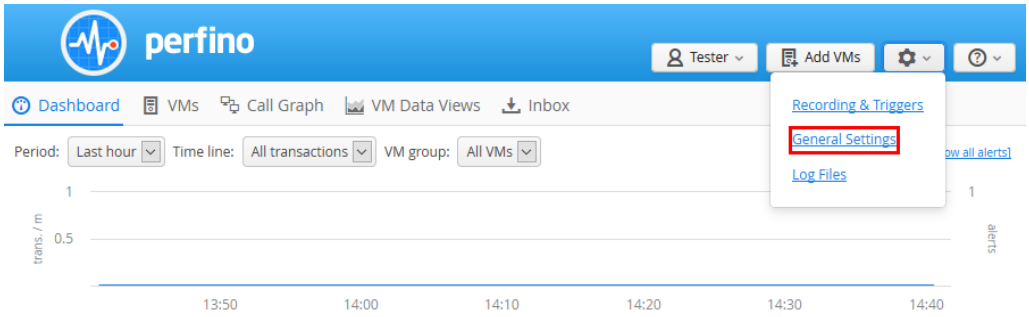
The first change you have to make is to set the **startRemoteServer** property to `true`. Then, the perfino server will listen for remote perfino UIs on the port configured with the **remoteServerPort** property.

The perfino UI can be deployed as a WAR file to a servlet container like Tomcat or Jetty. The WAR file has to be generated with the `deploy` tool in the `deploy` directory of your perfino installation. It will create the file `perfino-standalone.war` in the same directory.

In the application server, define the JNDI property **perfino/server** with a value of `"server name"[:port]` where "server name" is the name or IP address of the server where the perfino collector server is running. The port is optional and defaults to 1099 unless you have configured the **remoteServerPort** property differently. Then, deploy the generated WAR file into your application server. The perfino web application will make an RMI connection to the configured data collection server automatically.

A.2 Server Administration

Many server options are contained in the file `perfino.properties` in the installation directory and can only be changed when the server is restarted [p. 96] . The server administration settings that can be changed in the perfino UI are contained in the "General Settings". These settings include user configuration, license keys and data consolidation options.



Access control

There are three different access levels in perfino: admin, profiler and viewer. With these access levels you can implement a safe usage policy for the perfino UI in a larger organization.

- **Admin**

  When you install perfino, you have to create an admin user. Only an admin user can open the "General Settings" and make modifications to it.

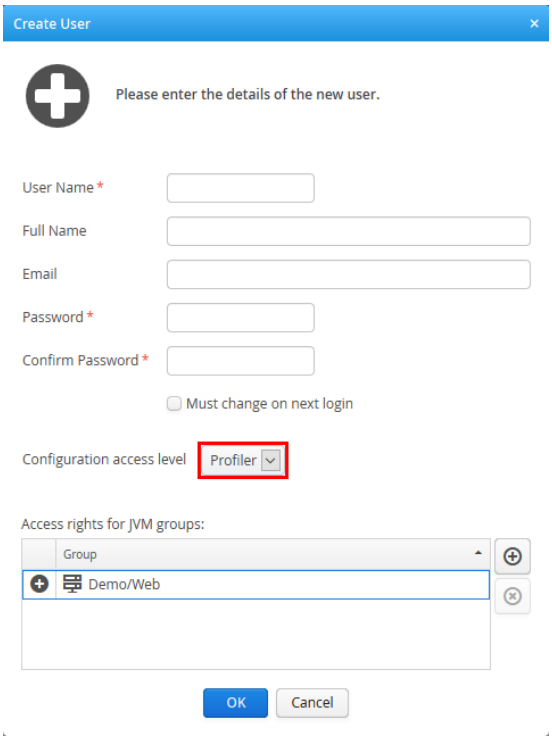
- **Profiler**

  With the "Profiler" access level, you can assign full control for a particular VM group, including the ability to put the JVM in profiling mode.

  First, you create the VM group in the recording settings, if it does not already exist. When creating the user, set its access level to "Profiler" and add the VM group to the list of access rights, removing the entry for "All VM groups".

  This user will now be able to change the recording settings for this VM group and all VM groups that are contained within it. In addition, heap dumps and CPU snapshots in profiling mode can be taken for the associated VMs.

  Viewing rights are not restricted in perfino.



- **Viewer**

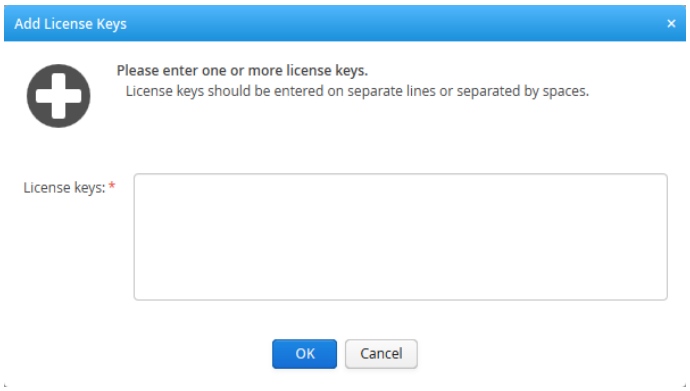
  To give somebody access to monitoring data without the ability to change any monitoring settings, use this access level.

License keys

As an administrator, you can add or remove license keys from perfino.

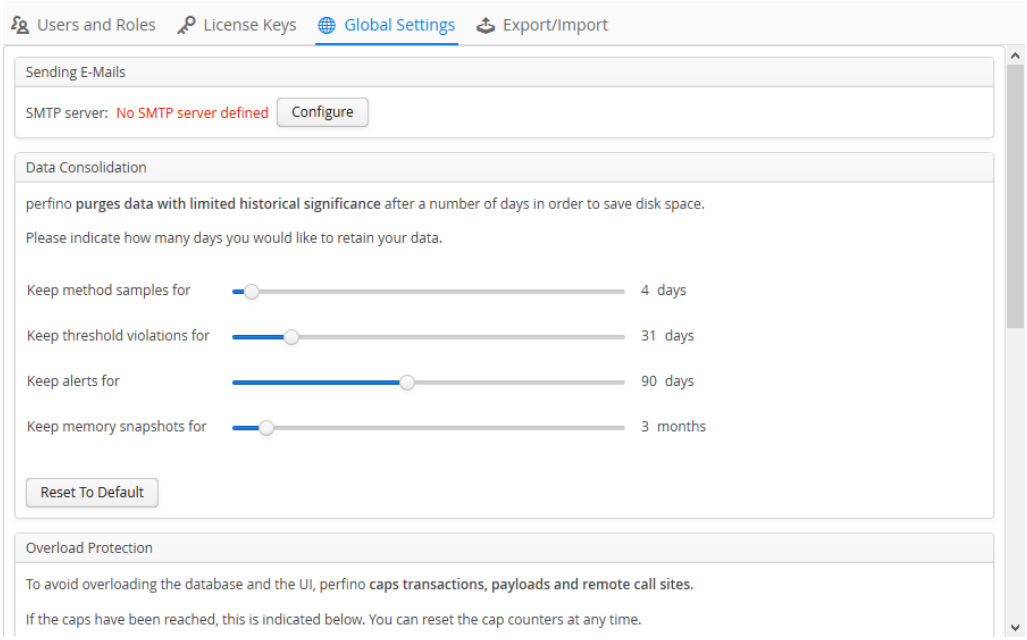
If more VMs are monitored than licenses are available, then some VMs will be marked as "unlicensed" and no monitoring data will be available for those VMs. When you add a new license key, perfino will start monitoring additional VMs right away, without the need for a restart.

The license key entry dialog can take multiple license keys, with each license key on a separate line. One of the license keys in perfino has to be a product key, otherwise no VMs can be monitored. License keys for additional VMs increase the maximum number of monitored VMs.



Global settings

In the global settings, you can configure an SMTP server that is used for sending emails. Emails are only sent by "Send email" trigger actions [p. 60] . If no SMTP server is defined, these actions do not have any effect.



The data consolidation options determine how long certain types of data are kept in the database.

Transaction data [p. 77] and telemetries [p. 50] are consolidated automatically, the highest aggregation levels remain in the database indefinitely. Due to the large interval size, the linear increase in required storage space is small.

Detailed data like method sampling takes a lot of space and is of decreasing interest the farther back you go. After some time, it can be deleted. The higher the retention times are set, the more disk space will be used by the database.

With the "frequency unit for telemetries" option you can make the numbers in telemetries easier to interpret. It should be set to a value that fits with the typical throughput in your applications. If your monitored VMs handle many transactions per second, you can set it to "per second", if the transaction frequency is more on the order of minutes, the setting "per hour" will be appropriate.

The global settings also show the currently installed version of perfino and let you configure automatic update checking. If selected, perfino will check for updates within in the current major version once a day. If an update is detected, an inbox message will be sent to all admin users. The inbox message contains the new version and hyperlinks for change log and download. An update notification will only be shown once even if perfino is restarted. When another version is released, a new update notification will be sent, even if you have not updated perfino in the meantime.

Whether you have enabled or disabled automatic update checking, the manual **[Check For Updates Now]** button performs the same check live and shows you its results in a dialog right away.

A.3 Import And Export

There are several kinds of motivations for exporting and importing your perfino configuration:

- **Backup**

  Backing up your perfino server configuration from time to time is a good idea. The configuration contains a lot of knowledge about your business transactions and their expected behavior. It is a valuable piece of data that could be stored in a version control system.

- **Staging**

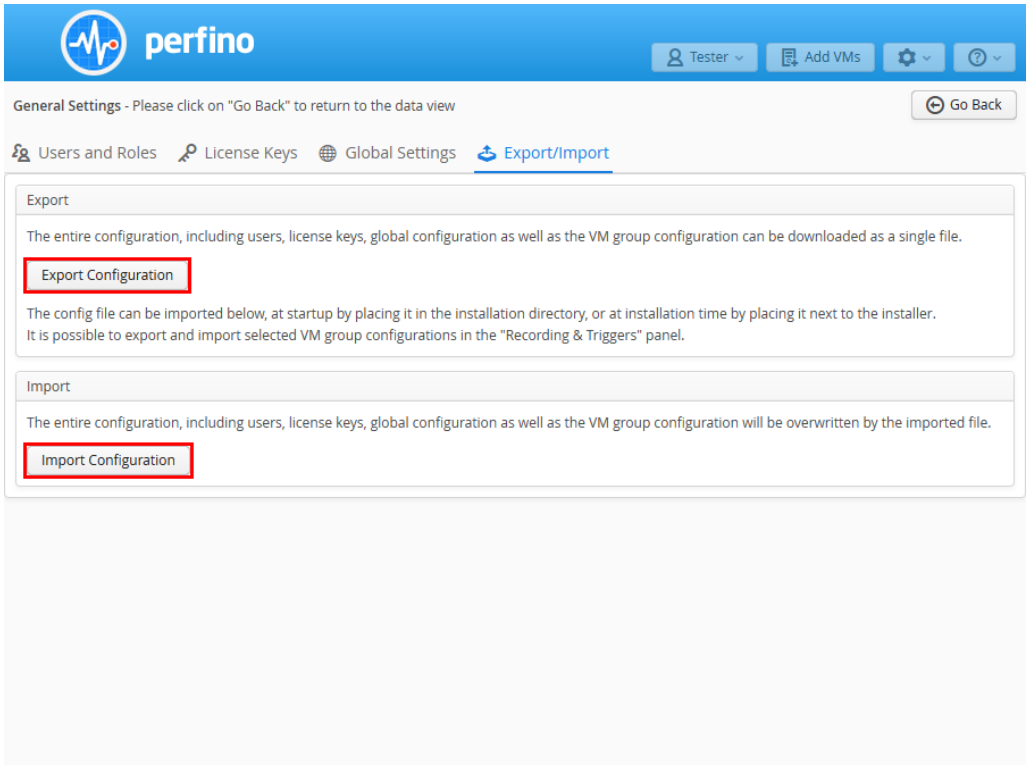
  If you have a staging environment where you try out changes in the perfino configuration before deploying them to a production server, you need a way to transport the configuration from one perfino installation to another.

- **Unattended deployment**

  If you deploy your perfino server to the cloud, you probably use the unattended installation mode [p. 103] of perfino and install it on a server instance where no perfino server has been installed yet. In that case you have to put the exported configuration next to the installer.

Server configuration

If you are an administrator, you can go to the general settings and select the "Export/Import" tab. The entire configuration is exported to an XML file. When you import that file, the entire server configuration is replaced. The change does not happen until you click on **[Apply Settings]**. It is not possible to delete the current user that way.

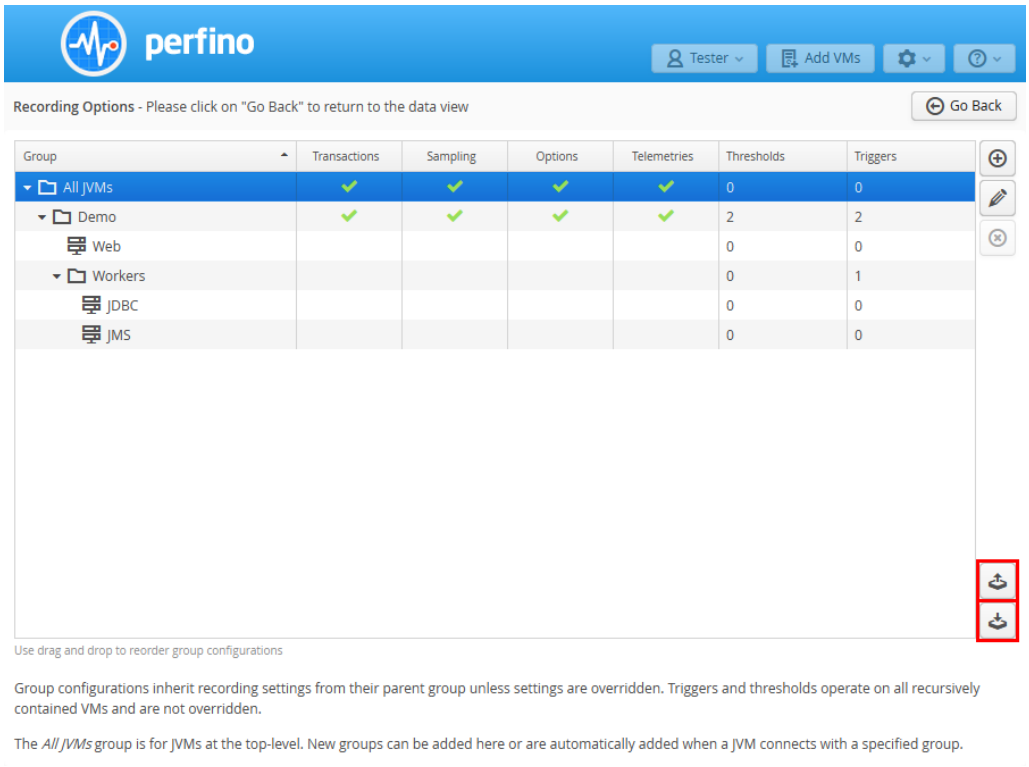


The XML file format is always backwards compatible, a more recent perfino server can read the exported file of an older version.

Naming that file `perfino_server_config.xml` and copying it to the perfino data directory overwrites the entire configuration at startup. This mechanism is used by the unattended installation [p. 103] . After the import, the file is deleted automatically.

Recording configuration

The exported XML file with the entire configuration contains license keys and user data. This may not be what you want for staging or also for backup purposes. In the recording settings, you can also export selected VM group configurations. Here, you can select "All VMs" or a particular VM group and click on the export button on the right.



This export is not only available for administrators, but also for users with "profiler" access rights.

Naming this file `perfino_recording_config.xml` and copying it to the perfino data directory overwrites the entire recording configuration at startup. The file is deleted automatically after the import has been completed.

A.4 Unattended Installations

Unattended mode and response files

In case you want to fully automate the installation of perfino, you can pass the argument `-q` to the installer. This makes the installer run in **unattended mode**. To set the installation directory, pass the argument `-dir [installation directory]`:

```
./perfino_unix.sh -q -dir /opt/perfino
```

To apply user input from a previous GUI or console installation, locate the response file `.install4j/response.varfile` in the installation directory and pass it to the installer with the argument `-varfile response.varfile`. The response file is a plain text file and the contained property definitions can be changed in a text editor. Properties related to the `perfino.properties` file will not be picked up from the response file. To modify them, you have to use the server configuration mechanism detailed below.

Automatic server configuration

In a cloud environment, you might want to recycle an instance or set up scripts that initialize a new instance with a completely configured perfino installation. To help you with that task, the perfino installer tries to read a number of **optional configuration files** with special names from the same directory.

Each of those files is explained in the following sections.

Server configuration

To adjust values in `perfino.properties`, you simply copy a `perfino.properties` file from the data directory of a configured installation into the same directory as the installer.

The installer will use the data in that file for the initial values. You can trim the contents of that file to the properties that deviate from the defaults, the installer will supply the default values for all other options. The basis for the structure of the actual `perfino.properties` file is the default template in the installer, so it does not matter if you delete comments or change the order of properties. Extra properties that are not present in the default template are merged in at the end.

Monitoring configuration

The monitoring configuration includes everything you can adjust in the perfino UI. Open the general settings in a configured installation, select the "Export/Import" tab and click on "Export

configuration". This will save the entire server configuration to a file [p. 101] . If you rename that file to `perfino_server_config.xml` and put it into the same directory as the installer, the installer will automatically apply this configuration in the new installation.

Agent configuration

The monitoring configuration in the previous section includes the configuration for the agent. However, there is one case where you might want to supply a separate agent configuration: When the perfino agent connects to a perfino server for the first time, it receives its configuration from the server and, as a consequence, some classes have to be reinstrumented for monitoring.

If your policy is to avoid all class retransformations, you can specify that on the "Options" step of the VM group configuration. In that case, any configuration change will only be applied when the monitored VM is restarted. To avoid the need for a restart in an unattended deployment, the configuration for the agent can be imported in advance.

First, you have to open the recording settings and export the VM group configurations [p. 101] of interest. Then, rename the exported file to `perfino_recording_config.xml` and place it next to the installer. The installer will perform the import for agents that are running on the local machine. For other machines, you have to perform this import yourself by calling

```
java -jar perfino.jar import perfino_recording_config.xml
```

The agent extracts its config from that file and writes it in binary form to the directory

`$HOME/.perfino/config`.

This directory is read by all agents on the local machine.

Note that these steps are only necessary if you want to avoid class retransformations. Otherwise all configuration changes are applied on the fly.

Agent and server keystores

You can encrypt and authenticate [p. 96] the communication between monitored VMs and the perfino server. The files `agent.ks` and `server.ks` constitute a key pair that enables both encryption as well as mutual authentication.

For an unattended deployment, you will probably already be using a particular key pair with your monitored VMs. In a configured perfino installation you can find these files in the `ssl` directory below the data directory. Placing them next to the installer ensures that they are copied to the same location in the new installation and that the server does not generate a new key pair.

Server SSL certificate

It is recommended to use SSL [p. 96] to encrypt the communication between the perfino UI server and browsers. If you enable SSL during the installation, a self-signed SSL certificate is generated and saved to `ssl/web.pkcs12` in the perfino data directory. You can replace that file with a certificate that is signed by a well-known certificate authority.

If you have such a certificate, you can put it next to the installer with the name `web.pkcs12`. No self-signed certificate will be generated in that case.

A.5 Automatic Update Of The Perfino Agent

When you set up a JVM for monitoring, you copy the perfino agent files to the machine where the monitored JVM is running. When you update the perfino server, the question arises how the agent files are updated.

While the server is shut down during an update, so that all of its files can be replaced, the monitored JVM cannot be terminated just for updating a monitoring agent. This is why perfino performs an automatic deployment of agent updates whenever the server installation is updated.

Server Update

You can check for updates within the same major series in the global settings [p. 98] and update notifications are sent to you as inbox messages if automatic update checking is enabled.

When you update the perfino server, all monitoring agents are disconnected from the collector. However, the agents continue to record data and will transmit it to the collector when it becomes available again. After a disconnection, the agent will periodically try to reconnect to the perfino collector with diminishing frequency. Data is only discarded if its quantity exceeds limits that are considered unsafe with respect to memory overhead.

After a server update, the perfino agent may have changed with respect to the older version. In that case, the VMs view will show an update icon next to the VM name. The perfino server will continue to work with agents of all previous versions, but new functionality may not be available for JVMs that are being monitored with an outdated agent.

When the monitored JVM is restarted at some point in the future, the new agent will be used automatically. There is no need for you to transfer new agent files to remote machines.

Agent Update Mechanism

When connected to a JVM that is monitored with an outdated perfino agent, the perfino server transfers the new agent files to the remote machine. Because the original agent files are in use and may be write protected, they cannot be overwritten. New agents are stored in the `$HOME/.perfino/agent2` directory. In that directory there are subdirectories for each monitored VM that in turn contain directories with the transferred agents.

When a monitored JVM is started, it loads the `perfino.jar` Java agent that you have specified [p. 10] in the `-javaagent` VM parameter. That Java agent bootstraps the actual implementation of the monitoring agent by looking into `$HOME/.perfino/agent2` and selecting the most recent agent files for the monitored JVM. If no agents have been transferred, the implementation in the `lib` directory of the extracted agent archive is used.

While the `perfino.jar` file with the bootstrapping code is never updated, it performs a limited function at startup that does not impact the monitoring functionality itself. Even if the JAR file changes in a newer release, it does not mean that you have to replace it on any remote machines.

A.6 Overload Protection

If too much information is generated that cannot be cumulated by the perfino collector, the system would be overloaded - the perfino collector would not be able to keep up with its consolidation efforts, the database would grow to eat up all disk space and the UI would become sluggish or unusable.

In a correct configuration, a limited amount of distinct information is generated. For the case that the configuration is not optimal, perfino provides an overload protection mechanism that prevents a breakdown by capping various recording types and warning you with inbox messages that your configuration needs to be adjusted.

When an overload cap is reached, the recorded information is not discarded, but no new names are generated and all further information is shown cumulated in a "capped description" node. While total numbers remain correct, insight into recording details is restricted until you fix your configuration and reset the cap counters.

Types of overloads

The problematic data collection types include

- **Transaction names**

  For example, if each distinct URL creates a transaction with the full URL path as its name and you have a lot of different static resources, then too many transaction names are generated. You should discard the static resource calls because they do not map to high-level business transactions.

- **Payloads**

  For example, if you have configured to resolve non-prepared JDBC statements separately, and the SQL statements contain non-numeric embedded IDs that perfino cannot extract and replace with ID markers, there will be too many distinct statements after a certain amount of time.

  Another possibility is that your prepared statements contain parameters directly in the statement body and not as bound variables. Such usage of prepared statements is non-optimal and does not work with perfino.

- **Remote call sites**

  perfino can track inter-VM calls for several technologies, like web services, EJB and RMI. For each such remote call site, perfino has to split its transaction trees and maintain associated information. In VM pools where many VMs call each other, this can lead to excessive overhead if the number of calls grows like O(n^2) with the number of monitored VMs.

Configuring overload protection

Each overload type is configurable separately. Open the general settings, select the "Global Settings" tab and change one of the maximum numbers in the "Overload protection" section.

Next to each overload type, a colored label informs you whether the particular cap has been reached yet or not. For example, if the transaction name cap has been reached, a typical strategy is to change the configuration so that less transaction names are created. After that, you want perfino to start counting again from zero. Click the **[Reset all cap counters to zero]** button in order to reset all counters.

B Advanced Topics

B.1 Annotation Transactions

Many frameworks use annotations for designating important classes and entry points that could be turned into transactions in perfino. With the "Annotated invocations" transaction type you can let perfino do this work for you.

Method and class annotations

An annotation transaction definition takes the fully qualified class name of the annotation that you are interested in. Annotations can be either used on **classes or on methods**. Before proceeding with the configuration, you have to tell perfino for which target the selected annotation is used.

For an annotated method, each method invocation becomes a transaction in perfino. Annotations on classes create transactions for all calls into **public instance methods**.

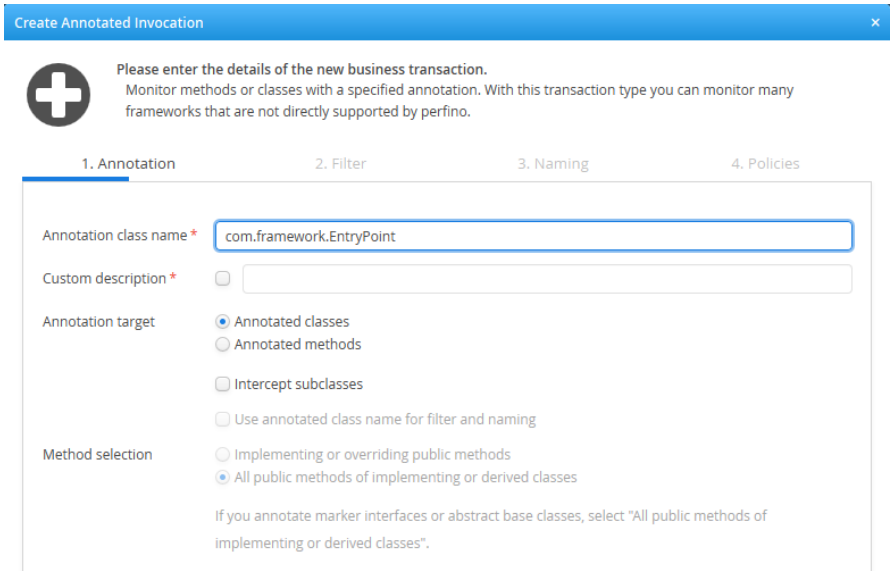
The second choice is whether derived classes should be considered as well. For methods, it can be important to capture the time in an overridden method. For example, if a framework creates an implementing proxy class and overrides the annotated method in order to add database transactions, you want to intercept the method in the proxy class, and not only the annotated method.

For annotated classes, the "Method selection" options determine which methods are selected to generate transactions:

- **Implementing methods only**

  If the annotation is placed on an interface that already defines all operations of interest with its methods, you should select this option.
- **All public methods**

  If the derived classes have their own methods of interest that do not implement or override the methods in the annotated class, use this option to create transactions from all public methods.

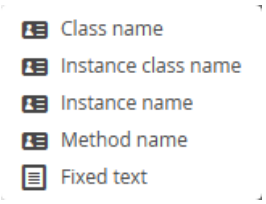


Naming

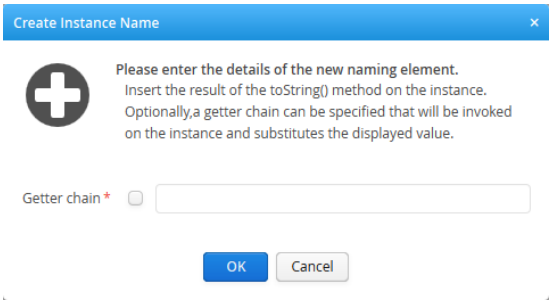
When you use annotations on classes with the setting for intercepting public methods in derived classes, the question remains what class name should be used in the transaction naming.

By default, a naming element of type "Class name" adds the name of the class where the intercepted method is defined. If you want to add the name of the annotated super class instead, select the "Use annotated class name for filter and naming" check box in the "Annotation" step of the wizard.

If you want to go the other way and add the class name of the actual instance on which the method was called, use the "Instance class name" naming element instead.



Even more specific, the "Instance name" naming element adds the `toString()` invocation on the object where the transaction method was called.



This naming element can also apply a getter chain to this object and append the `toString()` invocation on the result to the transaction name instead. For example, if the instance class has a `getVerbose()` method that returns the desired text, set the getter chain to `getVerbose()`. You can mix public field accesses and parameter-less invocations of public methods like this:

```
getParent().descriptor.getVerbose()
```

If you use the `getClass()` method to append a class name, there are two special fields that are provided by perfino to simulate the abbreviated and simple class name modes that are available for the "Class name" and "Instance class name" naming elements:

- With `getClass().simpleName`, the name of the class without its package is added. For example, `com.mycorp.MyClass` becomes `MyClass`.
- With `getClass().abbrevName`, the abbreviated package names are added. For example, `com.mycorp.MyClass` becomes `c.m.MyClass`.

Note that the "Instance name" naming element generates a far higher overhead than the "Instance class name" or "Class name" naming elements, since it always involves actual method calls.

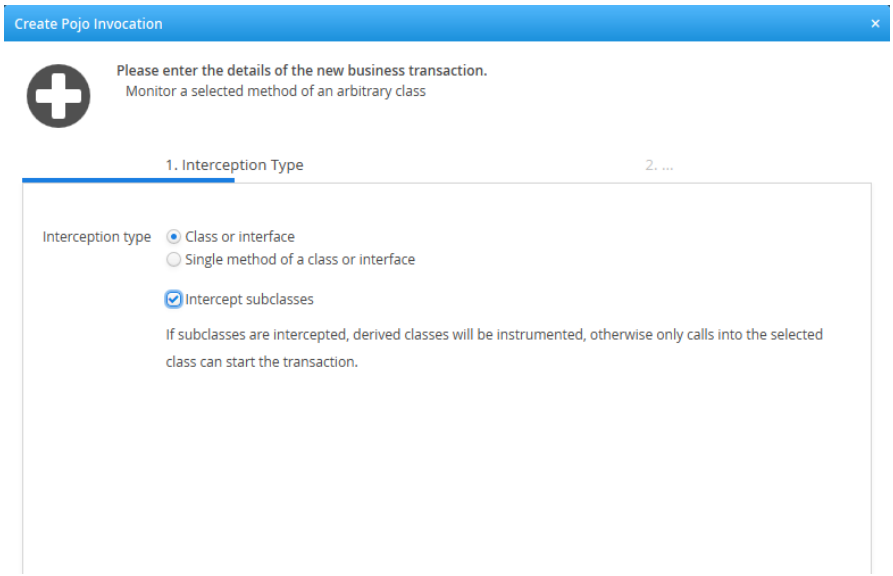
B.2 POJO Transactions

With POJO transactions you can take any method call in the JVM and turn it into a transaction. The functionality for POJO transactions mirrors that of the DevOps transactions [p. 112] which are specified directly in your code by using the perfino annotations.

POJO transactions are necessary if you cannot add annotations into your code or if the classes of interest are in an external library. Otherwise, DevOps transactions are recommended for easier maintainability.

Classes or methods

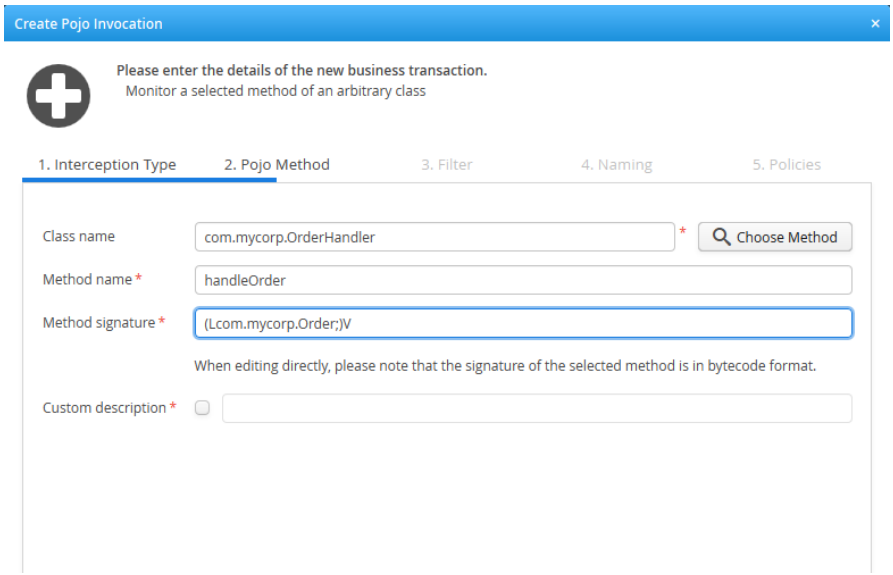
Similar to annotation transactions [p. 107] , the first question is whether you want to choose a single method or all public methods from a particular class.



POJO transactions have the same configuration options regarding inheritance as annotation transactions. You can switch on transactions from derived classes with the "intercept subclasses" check box.

For a single method transaction, this means that overridden methods will create transactions as well. If such a method makes a super call, that super call will create a separate transaction. If the names of both transactions are equal, the super call will not be recorded separately. However, with an "Instance class name" naming element it is easy to create two different names. In that case, you will see the super method as a nested transaction. If this is not what you want, you can set the "no nested transaction" option on the "Naming" step of the wizard to the value "That match this entry".

The method of interest is conveniently chosen with the method chooser. It can show you classes from all connected VMs or from JAR, WAR and EAR files that can be uploaded on the fly. You can also edit the method manually in which case you have to take care to enter the method signature in bytecode format.



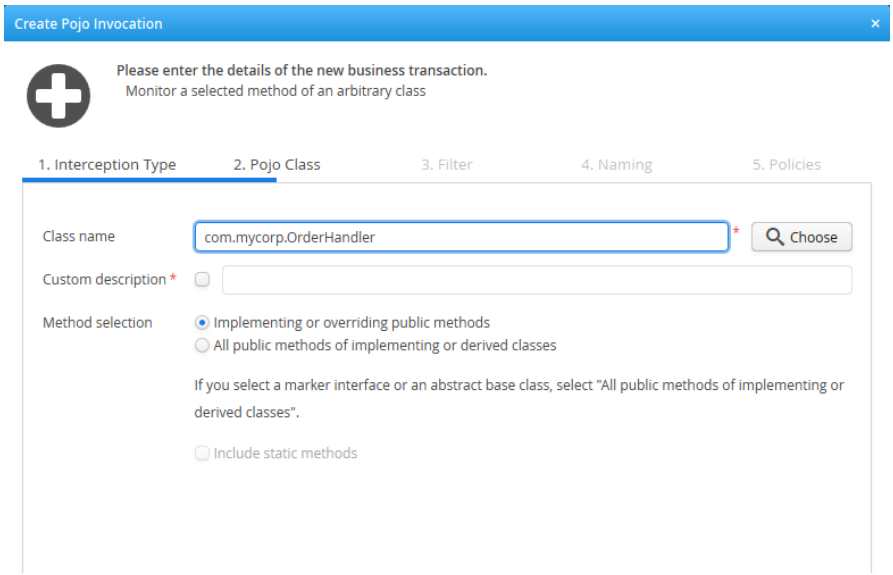
For classes, all public methods of the selected class will create transactions. When the inheritance option is turned on, you can choose between two different strategies for method selection in the derived classes.

- **Implementing methods only**

  This makes sense if you have selected an interface that defines all operations of interest with its methods. Additional public methods in implementing classes will be ignored.
- **All public methods**

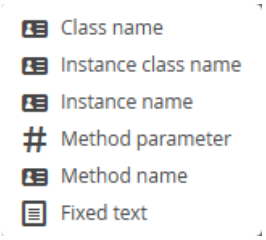
  The selected class may be a base class or a marker interface and the methods of interest are in the derived classes. In this case, all public methods in derived classes will create transactions.

As an additional way to collect transaction methods you have the option to select static methods. If the inheritance option is selected and all public methods are used, this will collect public static methods in derived classes as well.

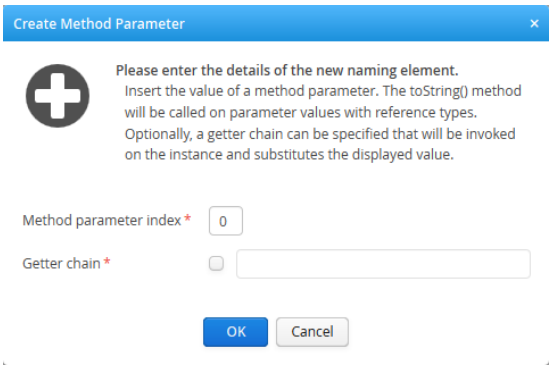


Naming

In addition to the naming elements that are available for annotation transactions, POJO method transactions have a "method parameter" naming element. Method parameter interception only makes sense if a particular method is selected, so it is not available for POJO class transactions.



You can select a parameter by its zero-based index. In addition, a getter chain can be applied to the parameter just like for the "Instance name" element that is described for annotation transactions [p. 107] .



B.3 DevOps Transactions

Maintaining POJO transactions in the perfino VM group configurations is an extra step in the development workflow and can get easily out of sync with the actual code. While POJO transactions are ideally suited for external classes, it is much more maintainable to directly annotate the methods and classes of interest in your own code.

The naming of this transaction type comes from the DevOps software development method that stresses the importance of increased collaboration between the development and the operations departments. Thinking about and implementing monitoring aspects at development time falls into this category.

The perfino annotations are located in the JAR file `api/perfino_api.jar`, see the javadoc overview for how to download this JAR file with Maven, Ivy or Gradle.

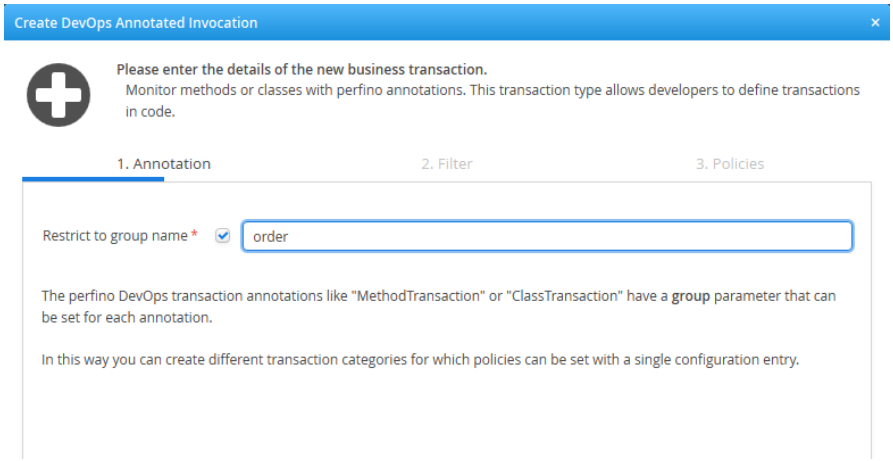
All annotations have a class retention policy. This means that they are present in the class file, but the JVM does not load them into memory. Code that queries runtime annotations on a particular method cannot be confused by the additional perfino annotations because they are only detected by perfino at the class loading stage and do not appear in the loaded class objects.

The usage of DevOps transactions is **described in detail in the Javadoc** that is present in the `api/doc` directory of the perfino installation.

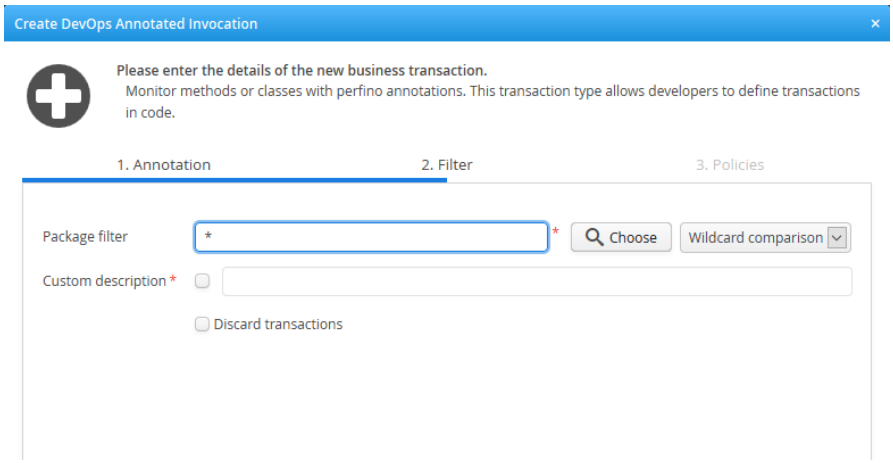
Policies

DevOps annotations only define the transaction naming, but not the policies for transactions. Policies are closer to the operations side and often need to be adjusted in production.

It is possible to select different DevOps annotations in the perfino configuration by way of the **group name**. Each DevOps annotations can have a "group" parameter that is set to the empty string by default.



If the configured groups are not sufficiently granular or if you want to single out a particular class, go to the next step to add a class filter. Here, it is also possible to discard all DevOps transactions that originate from the selected group and classes.



On the last step of the wizard, you define the policies for the selected DevOps transactions. As with other transaction types, it often makes sense to keep a catch-all transaction definition at the bottom and add more specific transaction definitions at the top.

B.4 Customizing Net I/O Thread States For CPU Recording

perfino offers you the possibility to record natively sampled CPU data [p. 89] with minimum overhead and stability risk for the entire VM. The resulting snapshot files can be viewed in JProfiler.

One pre-condition for useful sampling data is that all thread states where the thread is waiting are separate from the default "Runnable" thread state. If that is not the case, the top hotspots usually consist of methods that wait, block or perform network input and output (net I/O).

While waiting and blocking are comprehensively handled by the perfino sampling library, network I/O is often performed via native libraries that perfino does not know about. To mitigate this problem, perfino offers a mechanism to specify a list of additional methods that will be considered as net I/O.

To do that, you can specify the system property `-Dperfino.netioMethods=[path to text file]` in the Java invocation of the monitored VM. The referenced text file must be located on the machine where the monitored VM is running. Alternatively, perfino looks for the file `$HOME/.perfino/netio.txt` on Linux/Unix or `%USERPROFILE%\.perfino\netio.txt` on Windows.

In the `netio.txt` file, add your net I/O method definitions, one definition per line. A method definition can have one of three forms:

- **Class wildcard**

  Add a class name with a trailing asterisk, like

  ```
  com.mycorp.MyClass.*
  ```

  If you reference an inner class, the inner class separators must be written as dollar signs:

  ```
  com.mycorp.MyClass$InnerClass.*
  ```

  In this case, all methods of the selected class will be recorded in the net I/O state.

- **Signature wildcard**

  Add a class name and a method name, separated by a dot, like

  ```
  com.mycorp.MyClass.myMethod
  ```

  In this case, all methods of the selected class with the specified name but with an arbitrary signature will be recorded in the net I/O state.

- **Specific method**

  Like "Signature wildcard", but with the signature in bytecode format appended at the end. For example,

  ```
  java.net.AbstractPlainSocketImpl.doConnect(Ljava/net/InetAddress;II)V
  ```

  This is an actual net I/O method that is handled by default, the signature is for a method that returns void and takes a `java.net.InetAddress` and two `int` parameters. This is the same format that is used by JNI type signatures.

  The single specified method will be recorded in the net I/O state as a result of this definition.