

Introduction

Introduction

InfraRED is a tool for monitoring performance of a J2EE application and diagnosing performance problems. It collects metrics about various aspects of an application's performance and makes it available for quantitative analysis of the application.

InfraRED has the ability to monitor the complex architecture of J2EE application environments, provide detailed information for analysis and reporting, alert on performance related problems and guide you to determine the root cause of the problem. When you are trying to identify a performance issue that is causing your production application to not meet customer expectations or you are trying to proactively identify issues prior to deploying your application, InfraRED is essential to helping you save time, and ultimately ensure a better performing, more scalable Java application.

InfraRED uses Aspect Oriented Programming (AspectJ or AspectWerkz) to weave the performance monitoring code into the application.

Salient Features

- 👁 Fully non-intrusive. No coding required by developers.
- 👁 Layer-wise performance statistics summary (Web, Session, JDBC etc).
- 👁 API level detailed performance statistics.
- 👁 Call tree views (like in other profiler tools).
- 👁 JDBC API and SQL statistics.
- 👁 Last invocation statistics.
- 👁 Correlation of statistics across layers.
- 👁 Support for centralised gathering and presentation of performance data for one or more applications in a cluster.
- 👁 Remote or local collection of performance metrics.
- 👁 User-friendly web GUI.
- 👁 Simplified ant-based integration with application.
- 👁 Support for multiple application servers
 - 👁 Weblogic
 - 👁 Jboss
 - 👁 Tomcat
- 👁 Useful for applications with or without use of EJBs.
- 👁 Export summary of performance statistics into Excel spreadsheets.
- 👁 Very low overhead, Can be used in production environments.
- 👁 Ability to save and reload snapshots.

Why Use InfraRED?

Why Use InfraRED?

Why do you need InfraRED for your project? Here are some of the key differentiators of InfraRED that makes it attractive for you to use it.

End - To - End Statistics

InfraRED provides statistics about various aspects of an application's performance (Method timing, JDBC & SQL statistics, HTTP Response). It provides an end-to-end view to help in correlating all these metrics that are collected at different layers of the application. For example, you can use JDBC logging, p6spy or even Hibernate SQL logging to look at what SQL queries are executed, but InfraRED can give you a lot more information about those SQL queries and can also correlate these queries to the APIs in your application from which they were executed.

Minimal Overhead

InfraRED has been designed and tuned so as to have very little overhead and our goal has always been to make it usable in production environments. The overhead of running an application with InfraRED is less than 5%, for a typical enterprise application. Many applications do not have any performance monitoring tools in production systems and go through a lot of trouble to debug performance problems. InfraRED is designed to help you in such situations. Of course, most of its features are also very useful in performance labs, and even in QA and development environments.

Non - Intrusive

InfraRED is non-intrusive to the overall development of the application i.e. the developer need not edit any of his/her source files to monitor an application with InfraRED.

Flexible

InfraRED can be easily tuned to collect statistics at different levels of detail. The overhead incurred increases with the level of detail sought. On a production server InfraRED can be used to collect summary data with minimal overhead. On a developer system more detailed information can be collected to help debugging. The summary information would help isolate the problem to a certain area of code or a specific operation. More detailed analysis can be done on a developer machine to pinpoint the exact cause of the problem.

Focus on Persistence Layer

A large portion of performance related problems in a J2EE application arise in the persistence layer. This could be due to sub optimal SQL queries or improper use of the JDBC API. InfraRED has a special emphasis on the persistence layer. The metrics grouped under JDBC statistics is provided as a single page of vital information, top worst performing SQL queries, top most frequently executed SQL queries, prepare to execute ratio etc. It is also easy to find out the API from which the JDBC call was made.

Why Use InfraRED?

Why Use InfraRED?

Web UI

InfraRED provides an easy to use Web UI for analyzing the metrics. The first page shows the time spent in each layer of the application. On identifying the problematic layer, you can drill down further to find the problematic methods. You can further drill down and trace the execution of a method along with all the SQL calls that were executed.

Production tested and built from real experience

We have built InfraRED based on our practical experiences with building large-scale mission-critical enterprise applications for Fortune 500 companies.

Useful in all environments

Because of its easy setup and low overhead, you can use it in all environments. While its use in performance lab and production environments is clear, we usually use it even in DEV and QA environments. Early detection of performance problems is an important factor in building robust, well-performing systems. From that perspective, we have built features that make it easy for developers to identify problems on their own instead of waiting for a separate performance team to tell them where the problem is. One really cool developer feature is the 'Last Call' view that allows a developer to see the performance statistics (including all SQL queries) for the last few web requests in a single click. (There is no need to hunt and open log files, to search for the relevant data inside the log file or to ask DBAs to provide information about bad SQL queries.)

Cost and Licensing

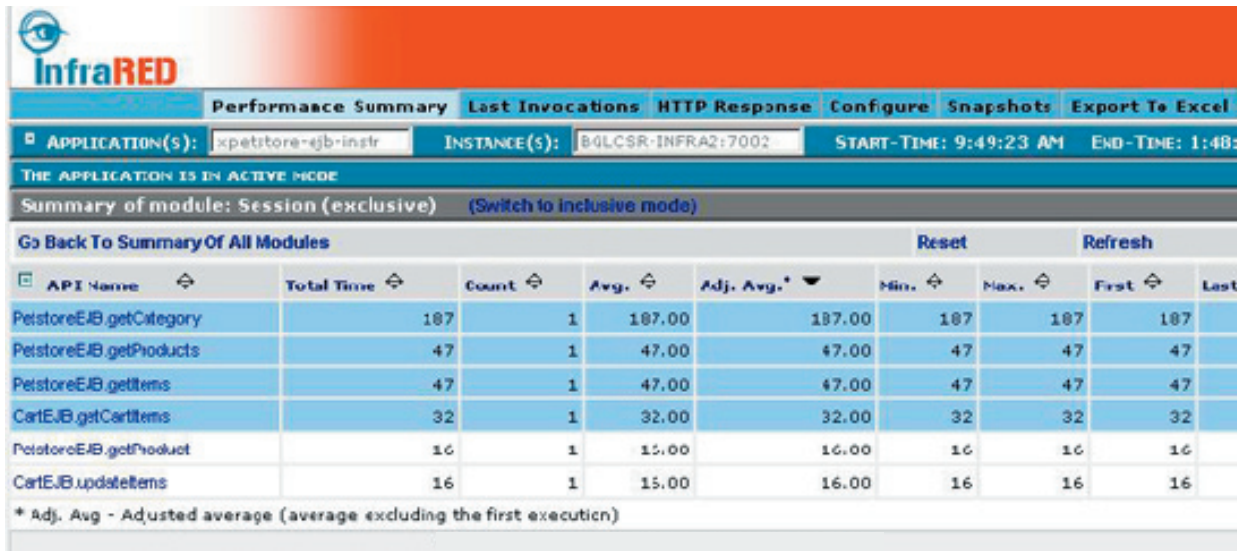
InfraRED is free and open-source



Screen Shots

Screen Shots

Minimum, Maximum, First, Last, Average API times for the various API in the layer are shown here. Each API can be clicked to view the call trace for that API.

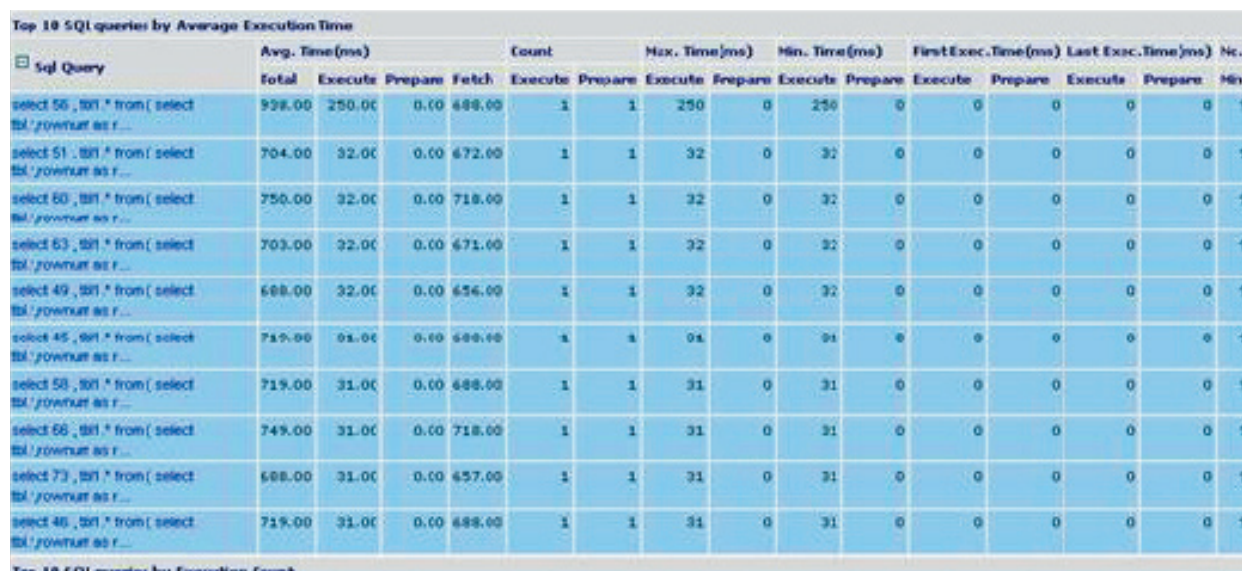


The screenshot shows the InfraRED Performance Summary interface. At the top, there are navigation tabs: Performance Summary (selected), Last Invocations, HTTP Response, Configure, Snapshots, and Export To Excel. Below the tabs, the application name is 'xpetstore-ajb-instr' and the instance is 'BGLCSR-INFRA2:7002'. The start time is 9:49:23 AM and the end time is 1:48:00 PM. A status bar indicates 'THE APPLICATION IS IN ACTIVE MODE'. Below this, there's a summary for the module 'Session (exclusive)' with a link to switch to inclusive mode. A table lists various API endpoints with their performance metrics. The table has columns for API Name, Total Time, Count, Avg., Adj. Avg., Min., Max., First, and Last. The first row, 'PetstoreEJB.getCategory', is highlighted in blue, indicating it is the most expensive query.

API Name	Total Time	Count	Avg.	Adj. Avg.*	Min.	Max.	First	Last
PetstoreEJB.getCategory	187	1	187.00	187.00	187	187	187	187
PetstoreEJB.getProducts	47	1	47.00	47.00	47	47	47	47
PetstoreEJB.getItems	47	1	47.00	47.00	47	47	47	47
CartEJB.getCartItems	32	1	32.00	32.00	32	32	32	32
PetstoreEJB.getProduct	16	1	16.00	16.00	16	16	16	16
CartEJB.updateItems	16	1	16.00	16.00	16	16	16	16

* Adj. Avg - Adjusted average (average excluding the first execution)

Top 10 most expensive queries and most frequent queries are shown for an operation. Queries taking more than the threshold value are highlighted in different color. APIs can be clicked to see call trace.



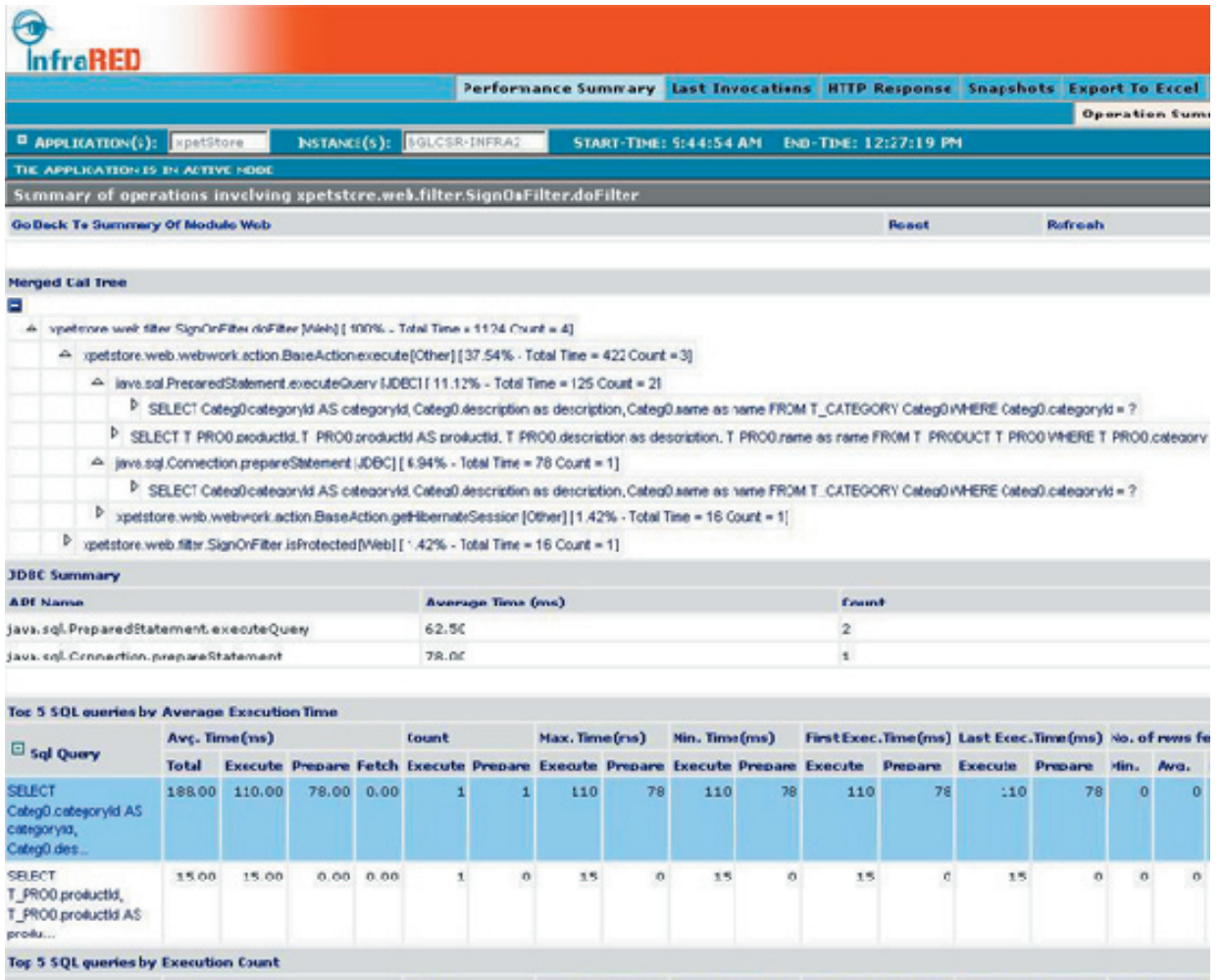
The screenshot shows a table titled 'Top 10 SQL queries by Average Execution Time'. The table has columns for SQL Query, Avg. Time (ms), Count, Max. Time (ms), Min. Time (ms), First Exec. Time (ms), Last Exec. Time (ms), and No. of Executions. The first row, 'select 56, t01 * from (select tbl_rownum as r...', is highlighted in blue, indicating it is the most expensive query. The table lists 10 queries with their respective execution times and counts.

Sql Query	Avg. Time (ms)			Count		Max. Time (ms)		Min. Time (ms)		First Exec. Time (ms)		Last Exec. Time (ms)		No. of Executions
	Total	Execute	Prepare	Fetch	Execute	Prepare	Execute	Prepare	Execute	Prepare	Execute	Prepare		
select 56, t01 * from (select tbl_rownum as r...	938.00	250.00	0.00	488.00	1	1	250	0	250	0	0	0	0	0
select 51, t01 * from (select tbl_rownum as r...	704.00	32.00	0.00	472.00	1	1	32	0	32	0	0	0	0	0
select 50, t01 * from (select tbl_rownum as r...	750.00	32.00	0.00	718.00	1	1	32	0	32	0	0	0	0	0
select 53, t01 * from (select tbl_rownum as r...	703.00	32.00	0.00	671.00	1	1	32	0	32	0	0	0	0	0
select 49, t01 * from (select tbl_rownum as r...	688.00	32.00	0.00	656.00	1	1	32	0	32	0	0	0	0	0
select 45, t01 * from (select tbl_rownum as r...	749.00	31.00	0.00	689.00	1	1	31	0	31	0	0	0	0	0
select 58, t01 * from (select tbl_rownum as r...	719.00	31.00	0.00	688.00	1	1	31	0	31	0	0	0	0	0
select 56, t01 * from (select tbl_rownum as r...	749.00	31.00	0.00	718.00	1	1	31	0	31	0	0	0	0	0
select 73, t01 * from (select tbl_rownum as r...	688.00	31.00	0.00	657.00	1	1	31	0	31	0	0	0	0	0
select 46, t01 * from (select tbl_rownum as r...	719.00	31.00	0.00	688.00	1	1	31	0	31	0	0	0	0	0

Screen Shots

Screen Shots

Merged call tree for an API can be seen. All the queries executed as part of the operation are also visible.



The screenshot displays the InfraRED Performance Summary interface. At the top, it shows the application name 'xpetstore' and instance 'TGLCSR-INFRA2'. The merged call tree shows the following structure:

- xpetstore.web.filter.SignOnFilter.doFilter [Web] [100% - Total Time = 1174 Count = 4]
 - xpetstore.web.webwork.action.BaseAction.execute [Other] [37.54% - Total Time = 422 Count = 3]
 - java.sql.PreparedStatement.executeQuery [JDBC] [11.12% - Total Time = 125 Count = 2]
 - SELECT Categ0.categoryId AS categoryId, Categ0.description as description, Categ0.name as name FROM T_CATEGORY Categ0 WHERE Categ0.categoryId = ?
 - SELECT T_PRO0.productId, T_PRO0.productId AS productId, T_PRO0.description as description, T_PRO0.name as name FROM T_PRODUCT T_PRO0 WHERE T_PRO0.categoryId = ?
 - java.sql.Connection.prepareStatement [JDBC] [6.94% - Total Time = 78 Count = 1]
 - SELECT Categ0.categoryId AS categoryId, Categ0.description as description, Categ0.name as name FROM T_CATEGORY Categ0 WHERE Categ0.categoryId = ?
 - xpetstore.web.webwork.action.BaseAction.getHibernateSession [Other] [1.42% - Total Time = 16 Count = 1]
 - xpetstore.web.filter.SignOnFilter.isProtected [Web] [1.42% - Total Time = 16 Count = 1]
 - SELECT ...

The JDBC Summary table is as follows:

ADF Name	Average Time (ms)	Count
java.sql.PreparedStatement.executeQuery	62.50	2
java.sql.Connection.prepareStatement	78.00	1

The Top 5 SQL queries by Average Execution Time table is as follows:

Sql Query	Avg. Time (ms)				Count		Max. Time (ms)		Min. Time (ms)		First Exec. Time (ms)		Last Exec. Time (ms)		No. of rows fetched	
	Total	Execute	Prepare	Fetch	Execute	Prepare	Execute	Prepare	Execute	Prepare	Execute	Prepare	Execute	Prepare	Min.	Avg.
SELECT Categ0.categoryId AS categoryId, Categ0.description as description, Categ0.name as name FROM T_CATEGORY Categ0 WHERE Categ0.categoryId = ?	188.00	110.00	78.00	0.00	1	1	110	78	110	78	110	78	110	78	0	0
SELECT T_PRO0.productId, T_PRO0.productId AS productId, T_PRO0.description as description, T_PRO0.name as name FROM T_PRODUCT T_PRO0 WHERE T_PRO0.categoryId = ?	15.00	15.00	0.00	0.00	1	0	15	0	15	0	15	0	15	0	0	0