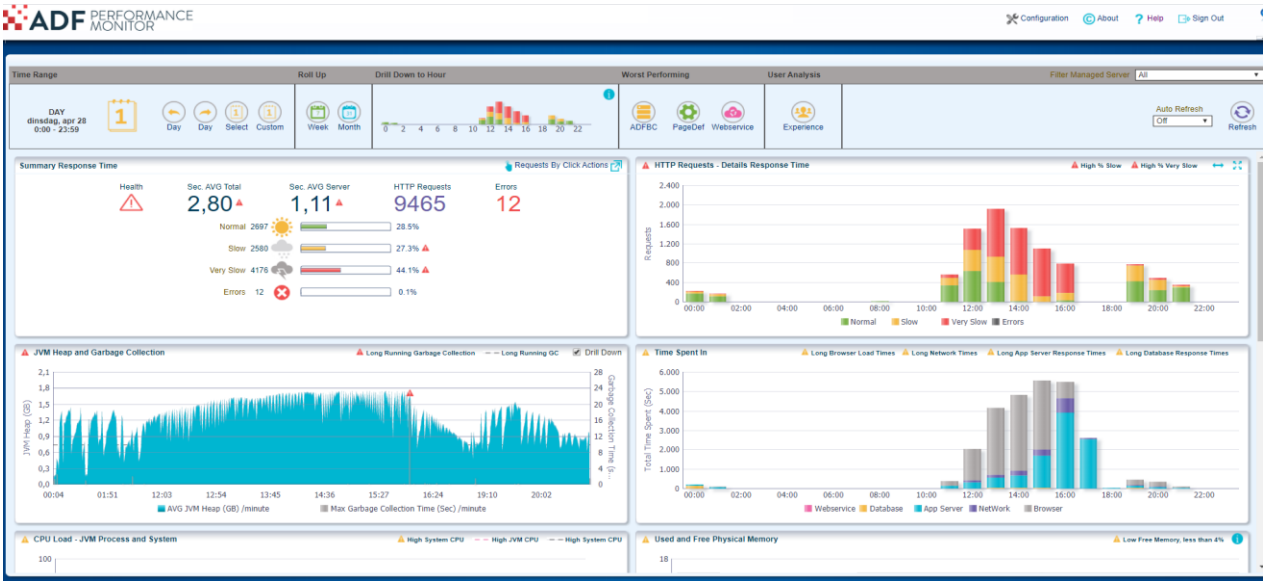
AMIS | CONCLUSION

ADF PERFORMANCE MONITOR

White Paper

Measuring, Analyzing, Tuning, and Controlling the Performance of Oracle ADF Applications



Author

Frank Houweling

Function

Principal Performance Consultant

ADF PERFORMANCE MONITOR

Contents

Executive Overview **3**
Introduction **3**
Oracle ADF Applications and Performance 3
ADF Performance Monitor 4
Features **5**
Summary Response Times and Health KPI Warnings 6
Details HTTP Response Times 7
Layer Occupying HTTP-Request Processing Time 7
JVM Performance 8
JVM CPU Load and System CPU Load 9
JVM CPU Time Consumption 10
Used vs. Free System Memory 10
AVG Response Time and 50th, 90th, 95th Percentiles 11
Active End-Users and HTTP Sessions 11
Linux Load Averages 12
Errors/Faults 13
HTTP-Request and HTTP-Response Network Time 14
Browser Load Time 15
ADF Click Actions 16
User ID Filter 18
Thread Waited and Blocked Time 19
Thread CPU Time of Requests 21
ADF Call Stacks 22
Troubleshoot Slow SQL Queries 23
ADF BC Memory Analyzer 24
Warnings and Suggested Solutions 25
End-User Metrics 26
Worst Performing Executions in ADF BC and ADF Model Layer 27
Support for EJB/JPA (EclipseLink) 28
Support for Soap/REST 28
Export to Excel or CSV File 28
Product Architecture **29**
How It Works 29
1. Metrics Collection 29
2. Metrics Logging 29
3. Dashboard Reporting Application 29
Monitored Events 29
Overhead 29
Clear Error Messages 29
Configuration **30**
Prerequisites 30
Configuration 30
Custom Instrumentation 30
Always Turn On/Off 30
Installation Guide 30
Training 30
Website 30
Use in JDeveloper During Development 31
Conclusion 32

Resources **32**
Disclaimer **32**
Appendix A: Comparison with Other Tools **33**

Executive Overview

The ADF Performance Monitor enables enterprises to maximize the value of ADF applications by identifying, diagnosing, and fixing the blind spots and recurring problems that frustrate users. For business-critical applications, it is critical to monitor, diagnose, and resolve application problems before they affect revenue. The ADF Performance Monitor is specifically designed to measure, analyze, tune, and control the performance of Oracle ADF applications, delivering enterprises insight into real end-user experiences. During the lifecycle of an ADF application, the product's QA teams and administrators can detect, analyze, and resolve common and uncommon response-time and resource-usage issues. This document provides more information about the architecture, features, and implementation of the ADF Performance Monitor. Further information is available at adfpm.com.

Introduction

Oracle ADF Applications and Performance

Good performance is the key to the success of a web application, and Oracle ADF applications are no exception to this rule. However, identifying and diagnosing bottlenecks in Oracle ADF applications can be time-intensive, costly, and challenging. Although ADF is a powerful, advanced, and highly configurable framework enabling high performance and scalability, choosing incorrect performance configuration parameter settings can lead to pitfalls. This can be caused by the default values not being the optimal values. Frequently, even experienced ADF developers cannot pinpoint why an ADF application is running slowly. In this case, behind-the-scenes information is extremely useful for a better understanding of the ADF application.

ADF Performance Monitor

The ADF Performance Monitor detects and identifies the causes of performance problems in a production, test, and development environment. The tool comprises a library that attaches to the ADF application, a database schema, and a separate dashboard-based reporting application. The library collects runtime performance metrics. During development, issues are reported in the JDeveloper's console log; in test and production environments, metrics are saved in a database schema and issues are reported in real time in the dashboard application (Figure 1).

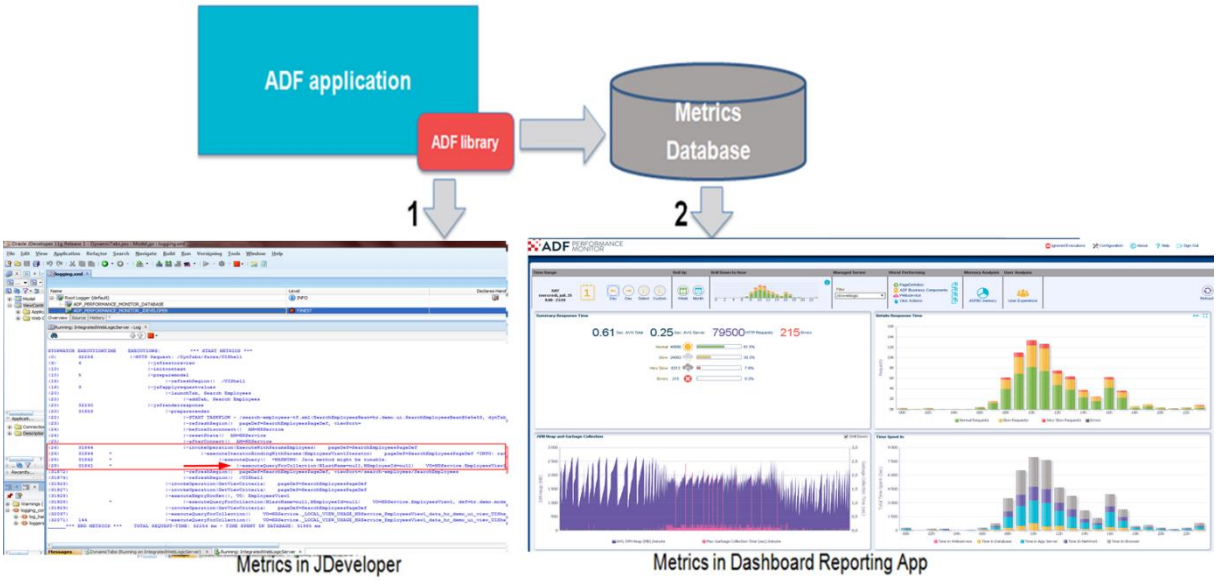


Figure 1: Performance metrics are reported in JDeveloper during development (1); in test and production environments, metrics are saved in a database schema, and issues are reported in a dashboard reporting application (2).

With the ADF Performance Monitor development, QA and operation teams can:

- receive real-time and historic (today, yesterday, last week/month) performance overviews;
- access insight into real end-user experience;
- gain knowledge of the layer occupying HTTP-request processing time - the dashboard shows time spent in the database, webservice, application server (appserver), and network/browser load time;
- receive insight into processes inside the ADF application and in the ADF framework (i.e., which methods, operations, and queries are executed, when and how often);
- detect bottlenecks, inefficiencies, and typical bad practices throughout the life cycle of an ADF application, enabling better application-architecture design decisions;
- increase the ADF application's scalability (optimal infrastructure, hardware, and license utilization);
- filter user IDs, user sessions, and business transactions through WebLogic managed server (e.g., to troubleshoot problems);
- gain insight into the usage of ADF Business Components memory; and
- receive insight into errors/exceptions, including their type, severity, ADF call stack, and stack traces (both technical and functional), and subsequently troubleshoot them quickly.

In *Appendix A: Comparison with Other* **,** a comparison is made with other performance monitoring tools in the ADF landscape.

Features

The dashboard (Figure 2) provides a 24/7 overview of performance within a selected time range.

Figure 2: Performance overview in a selected time range (month, week, day, hour, 5 minutes).

It displays real-time critical information about the ADF application's performance, providing specific feedback to common queries: Are response times within or outside SLA boundaries? What is the error rate? What is the health of the JVM? Is immediate action required?

In the menu, a time range can be selected (Figure 3) from 5 minutes to 1 month. Any time range can be drilled down to or rolled up to (5-minute, hour, day, week, month), or you can navigate to the next or previous equivalent time range.

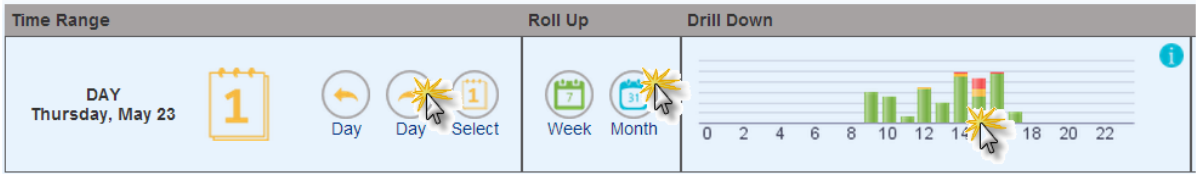


Figure 3: Drill down, roll up, or navigate to the next or previous time range.

Summary Response Times and Health KPI Warnings

The top-left box provides a summary of HTTP-request response times in the experience of real users, using the following categories:

1. “normal” (well within the SLA range);
2. “slow” (somewhat outside the SLA range);
3. “very slow” (well outside SLA boundaries—this is considered seriously problematic); and
4. errors/faults.

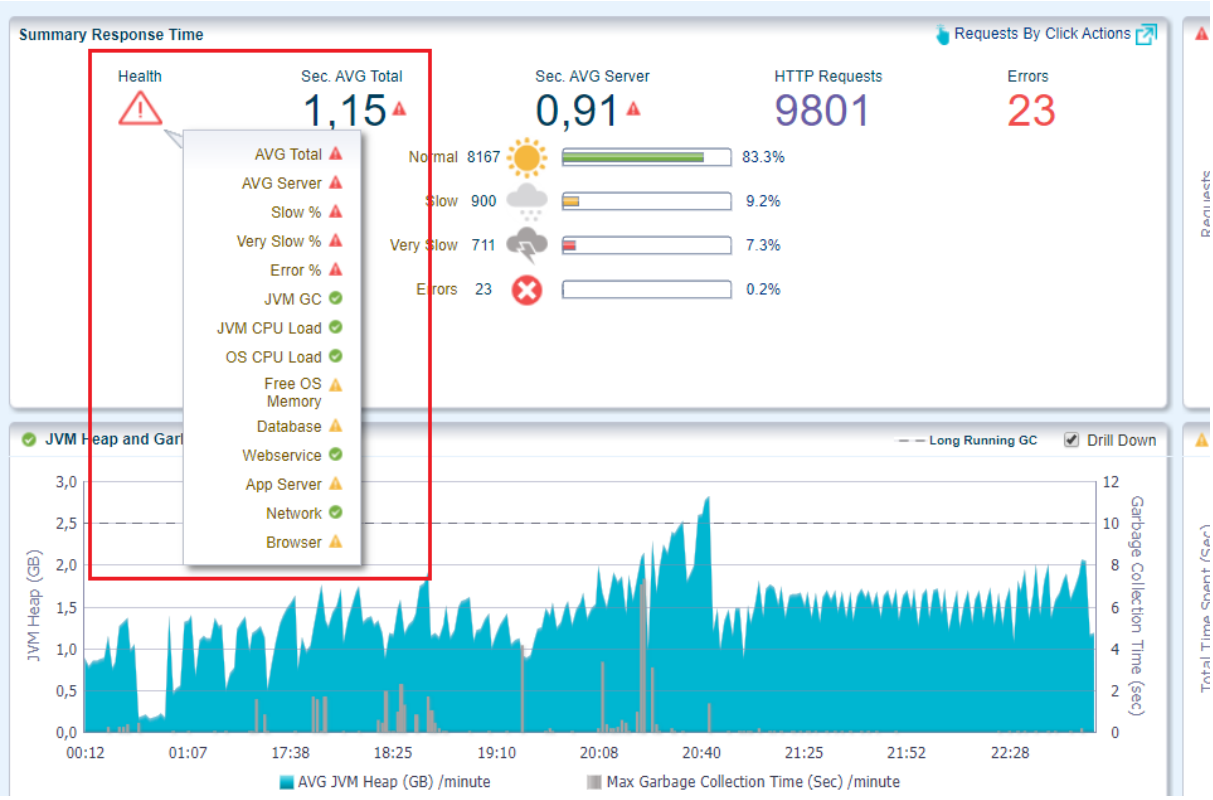


Figure 4: SLA and Health warnings: the AVG total time, AVG server, % Slow, and % Very Slow are greater than the configured threshold.

The ADF Performance Monitor features automatic SLA and health KPI warnings for AVG Total Process Time, AVG Server Process Time, Error %, Very Slow Requests %, JVM garbage collection, JVM and system CPU load, OS memory, database, webservice, appserver, network, and browser.

Depending on the performance targets or SLA, teams can configure how the monitor should interpret HTTP-request response times, SLA and health KPI warnings.

Details HTTP Response Times

HTTP response times can be analyzed over the selected time range. This makes visible when the load is high (and how high), how the response times are distributed over the categories (normal, slow, very slow and error) and when there are performance problems (more red and yellow colored parts of the bars). When there is more red (and yellow) colored parts in the bar chart (top right) this should be a trigger to drill down to this hour for further analysis to find the root cause.

Figure 5: The number of HTTP Request divided over a day. Each bar represents an hour. For each hour the number of normal, slow, very slow requests and errors are shown.

Layer Occupying HTTP-Request Processing Time

This chart shows the layer occupying HTTP-request processing time; i.e., whether more time is spent with the database, webservice, appserver, network, or browser.

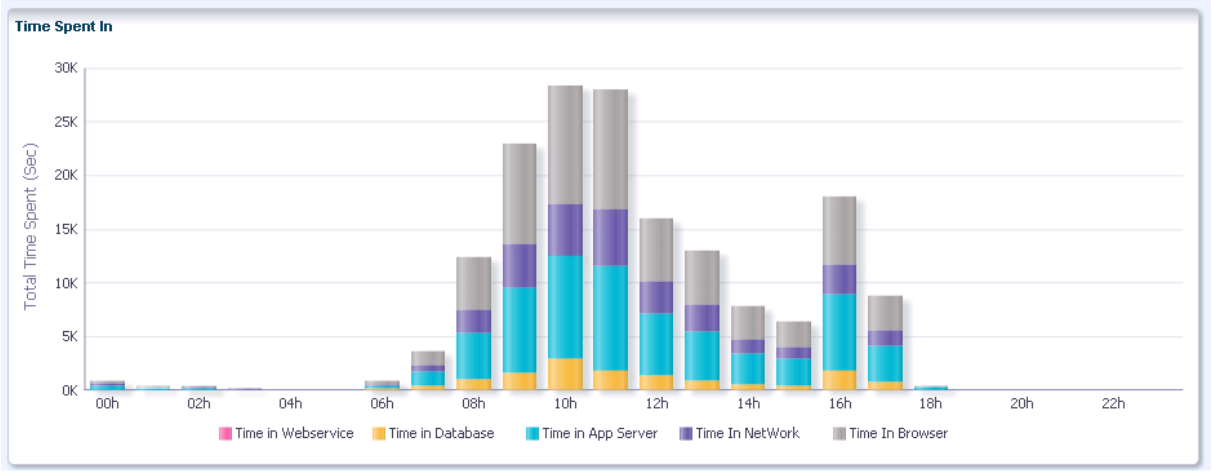


Figure 6: The total execution time by layer.

JVM Performance

A healthy JVM is critical to performance management. The ADF Performance Monitor shows real-time or historic heap usage and garbage collection times (Figure 7). If garbage collections are running over a configurable threshold (for example, 20 seconds), a warning sign appears. This is often an indication of a problem, such as a freeze of all current requests because the JVM cannot clear enough memory.

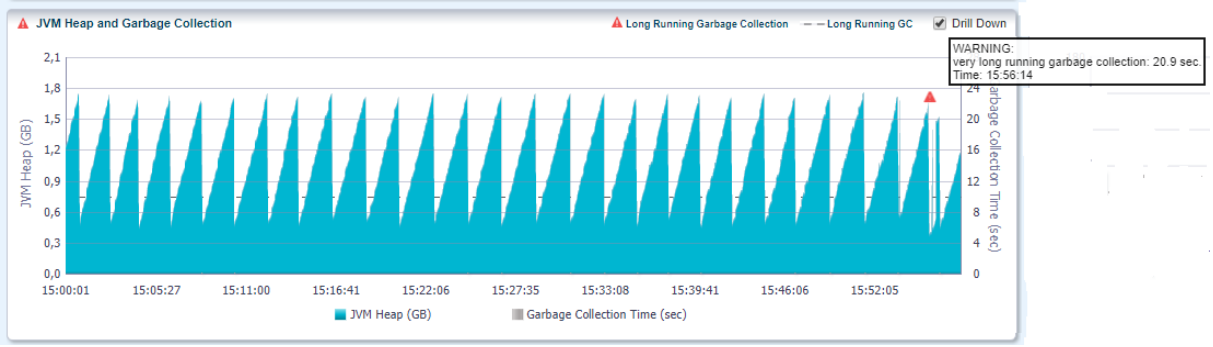


Figure 7: Example of exceptionally long-running garbage collections (20.9 sec).

JVM CPU Load and System CPU Load

All critical IT infrastructure devices contain a central processing unit (CPU) that ultimately dictates the performance, stability, and performance capacity of the device. CPU performance latency occurs when one or more processes consume most of the processor time. Threads that are ready to be executed must wait in a queue for the processor to become available. Trying to overcome a processor bottleneck by throwing hardware at the problem (e.g., more CPU, more memory, faster disks, more network connections) will often not help. Monitoring CPU usage helps you to analyze spikes in CPU load and identify overactive CPU usage. Depending on the CPU performance behavior, you can:

- upgrade the CPU or add multiple processors;
- cut load;
- find underlying performance bottlenecks caused by (ADF) program code;
- avoid exorbitant costs arising due to unnecessary upgrades;
- identify unnecessary background processes running; and
- find out the resource utilization of a process and its impact on the system as a whole

The monitor shows the CPU load percentage of the JVM and the total CPU load percentage of the whole underlying operating system. A high CPU usage rate and memory usage may indicate a poorly tuned or designed application.

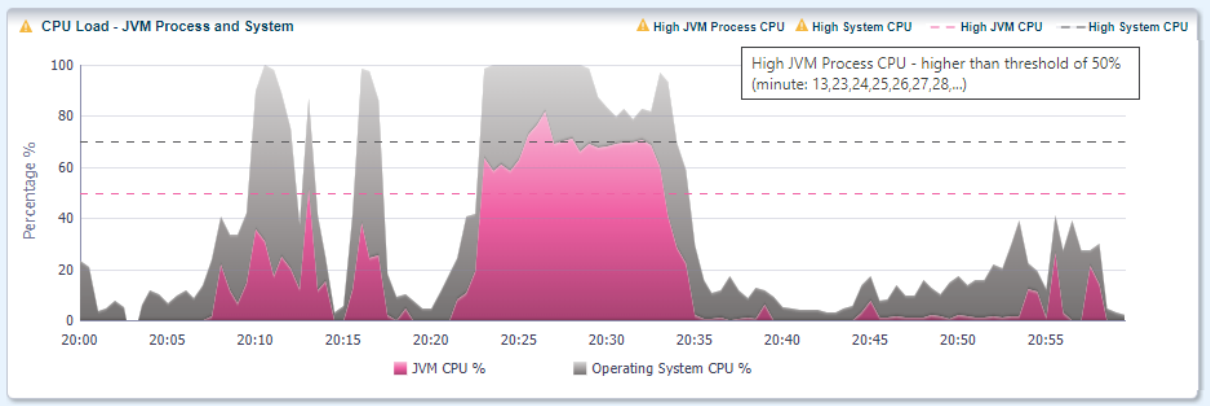


Figure 8: The JVM CPU load and the System CPU load were quite high, ranging from 20:23 to 20:35.

- **CPU Load of the JVM Process**
  The JVM CPU load (pink) is dependent on the activities going on in the ADF application program code or in libraries. This is useful when the ADF application puts too much load on the available CPU's. If this load is 100%, then CPU's are actively running threads from the JVM 100% of the time (this includes application threads as well as the JVM internal threads). If frequently during the peak hours the JVM process load is (on average) more than 60% the monitor gives a warning, this should be a trigger to investigate this and to try to bring it down.

- **CPU Load of the Whole Underlying Operating System**
  The Operating System CPU Load (grey) depends on all the activities going on in the whole system. When there are other big background processes executed on the same machine (!), it is important like to monitor them, the performance of the ADF application will likely be influenced. This can also be the case when multiple applications are deployed on the same machine. If during the peak hours the total system load is (on average) more than 80% the monitor gives a warning. It is very wise to investigate this warning, and to limit the CPU burning processes. In the hour overview screenshot above shows that from 20:20 to 20:30 the total system CPU load was 100% (!).

JVM CPU Time Consumption

It is very useful to monitor the total execution time of the JVM process. This corresponds a little bit with the application % usage but is also different. It is in the total CPU time that the JVM process (the WebLogic managed server where ADF application is deployed on). It is useful to get insight in the CPU time consumption of the ADF application, and especially when the load is high. Different releases can be compared, and ADF developers can try to bring down (unnecessary) CPU time of expensive, 'CPU burning', operations in the ADF application.

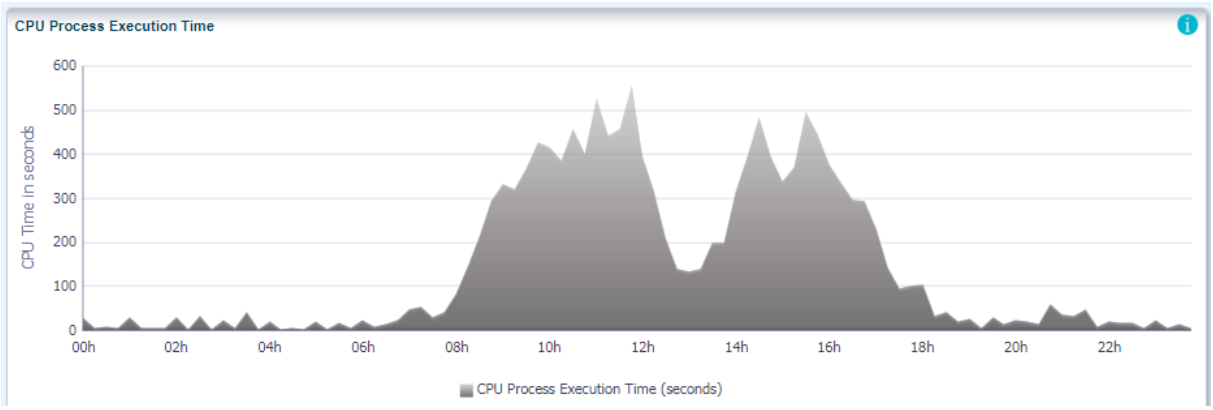


Figure 9: The JVM CPU Time. This shows a typical day pattern: employees arrive around 08:00, work the most between 10:00 and 12:00, go to lunch between 12:00 and 14:00, work some more, and go home around 17:00.

Used vs. Free System Memory

Just as important it is to monitor the JVM heap memory and garbage collections, it is important to monitor the whole operating system memory for over-consumption. When an operating system runs low on memory, it swaps memory in and out, which is expensive and resource-consuming. The ADF Performance Monitor keeps track of how much memory is available, avoiding that scenario.

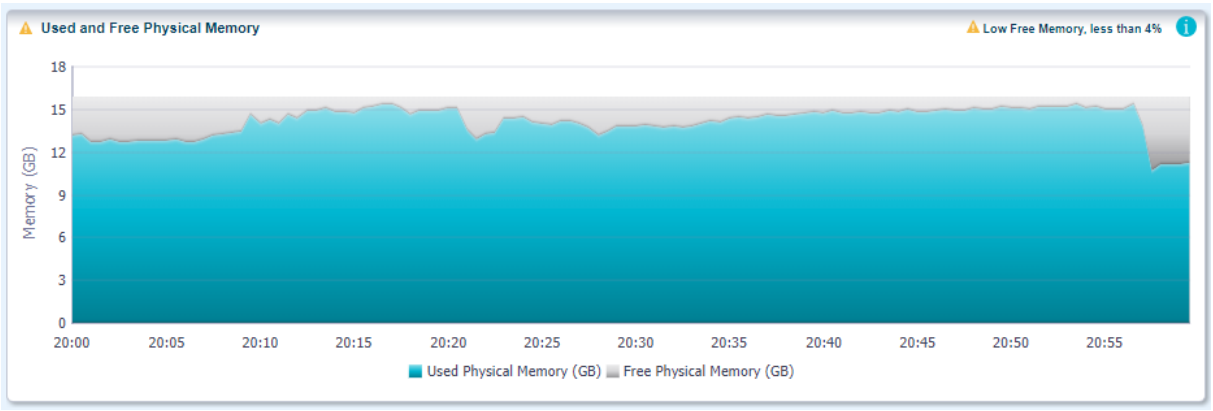


Figure 10: There is limited free memory—less than 4% around 20:15-20:20, and around 20:50-20:55.

AVG Response Time and 50th, 90th, 95th Percentiles

Percentiles, when compared with averages, describe the consistency of application response times. Percentiles provide good approximations and can be used for trend analysis, SLA agreement monitoring, and daily performance evaluation and troubleshooting (see Figure 11). After a new release on 17 June, the 95th, 90th, and 50th percentile response times decreased, indicating that performance had improved.

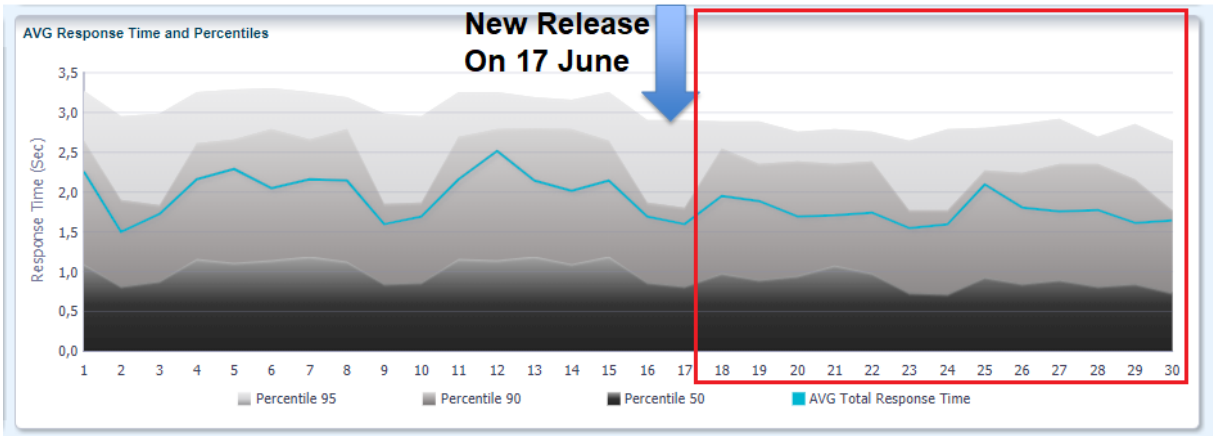


Figure 11: After a new release on 17 June, the 95th, 90th, and 50th percentile response times decreased, indicating that the performance had improved.

We can use percentiles for all kinds of performance evaluations, including regressions and trend analyses after new releases, allowing us to know whether performance was really improved. Sometimes performance metrics increase or decrease after new releases, so it is useful to be able to visualize the specifics.

Active End-Users and HTTP Sessions

This overview is particularly useful for evaluating the number of active end-users and HTTP sessions (Figure 12) on either a single managed server or all managed servers together. You can compare the load for each server and compare the values against the ADF Performance Monitor's other metrics, such as JVM, CPU, percentiles, SLA health metrics, and time spent in layers. This can be useful when you want to know the maximum number of unique users that a managed server can handle, before the performance get worse. Or if you simply want to know how many end-users are actively using the application.

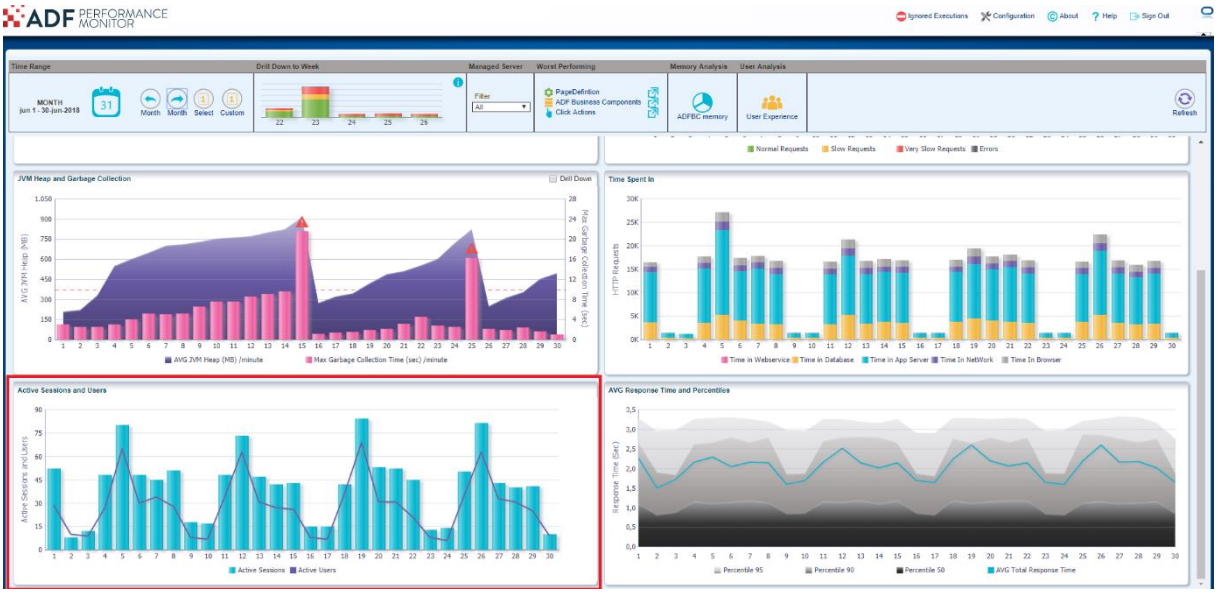


Figure 12: Active end-users and active HTTP sessions.

Linux Load Averages

An alternative and excellent way to look at the load of the whole system is the Linux load averages. Linux load averages are “system load averages” that show the running thread (task) demand on the system as an average number of running threads on the processors plus waiting threads (waiting threads queued to the available processors). This measures demand, which can be greater than what the system is currently processing. The ADF Performance Monitor shows the Linux load averages of the last minute (this is what Linux shows in the loadavg command of the last minute).

It looks a bit like the CPU system load and CPU process time, but the difference is that the Linux load average includes waiting threads on the queue. During a very high (over) load spikes are much more visible in this chart because it includes waiting thread queued to the CPU’s.

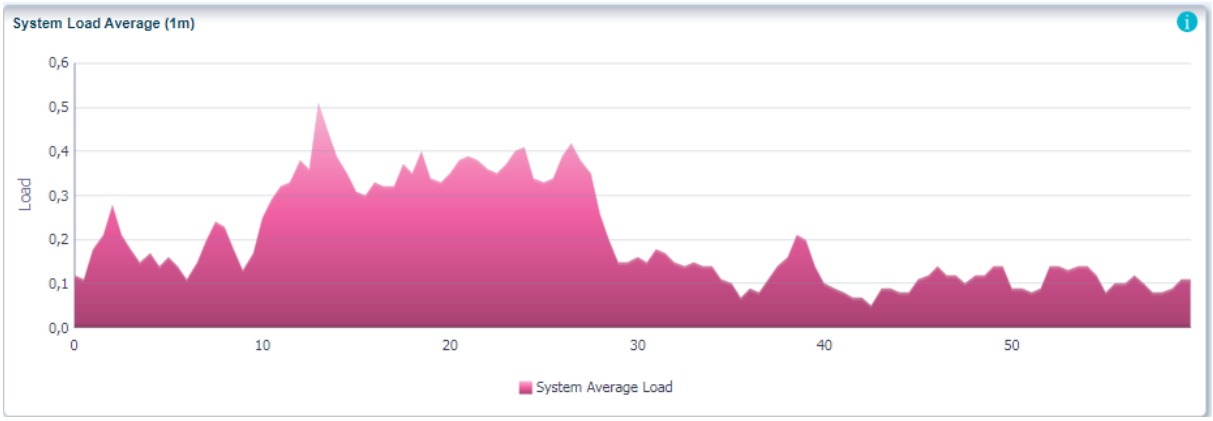


Figure 13: Linux Load Averages

Errors/Faults

Insight into the number, type, and severity of errors that occur in a test or production environment is crucial to resolving them and making a stable ADF application that is less error-prone. Application errors (and their stack traces) are often difficult to retrieve or time-consuming to find. Project teams commonly depend on end-user and tester reports; typically, these reports do not include all errors or feature insufficient information about the errors. Operational teams do not always have the time to monitor for errors in the WebLogic console or the Enterprise Manager, nor to manually trawl expansive log files to identify the error piecemeal. Errors are collected by the ADF Performance Monitor to address this issue. Development, QA, and operational teams can drill down to the error messages, containing type, severity, and stack traces, to quickly troubleshoot errors and make the application more stable.

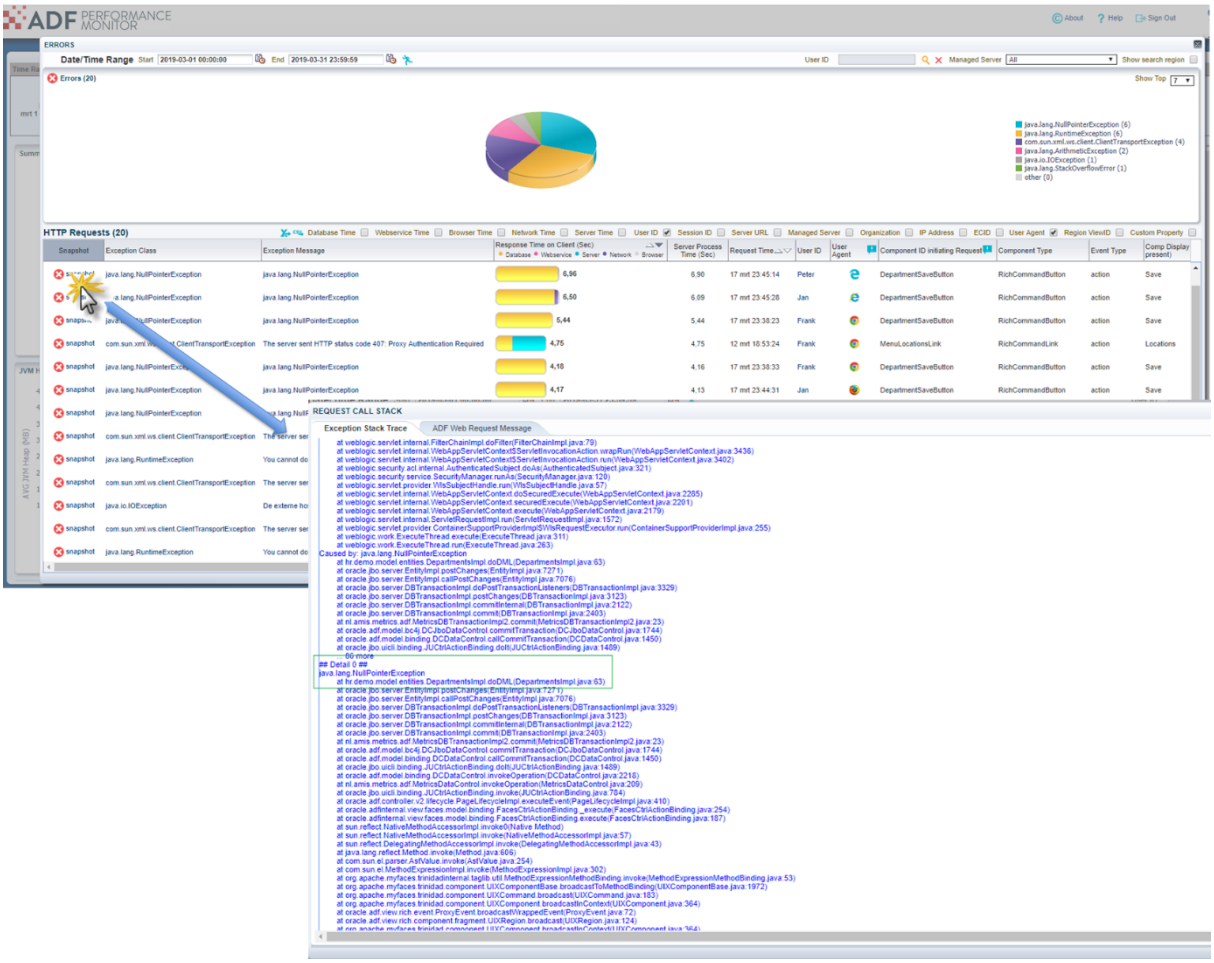
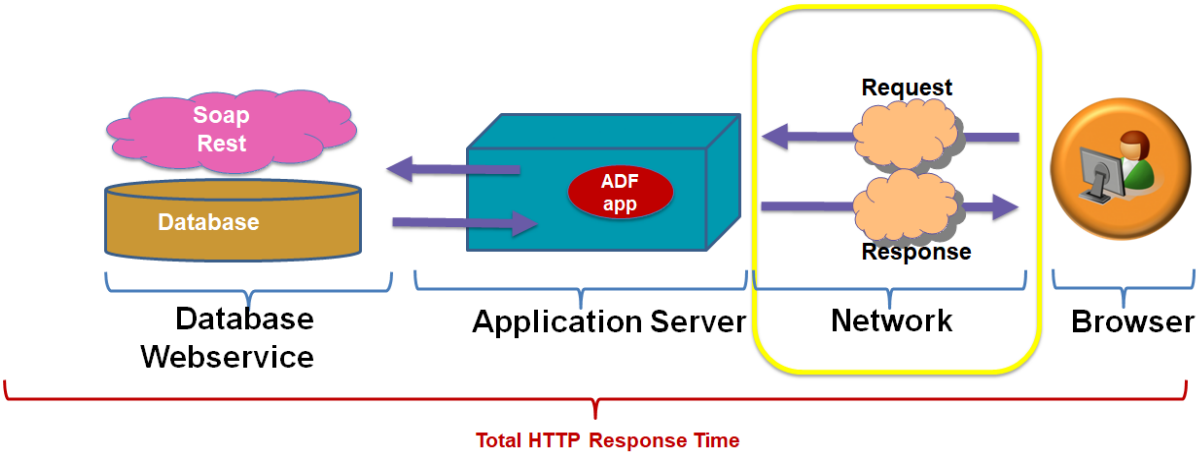


Figure 14: Top 10 errors/faults overview by occurrences. Example of a *NullPointerException* exception stack trace.

HTTP-Request and HTTP-Response Network Time

Network time is the time it takes to send an HTTP request from a browser (HTTP-request network time) to the appserver and from the appserver back to the browser (HTTP-response network time).

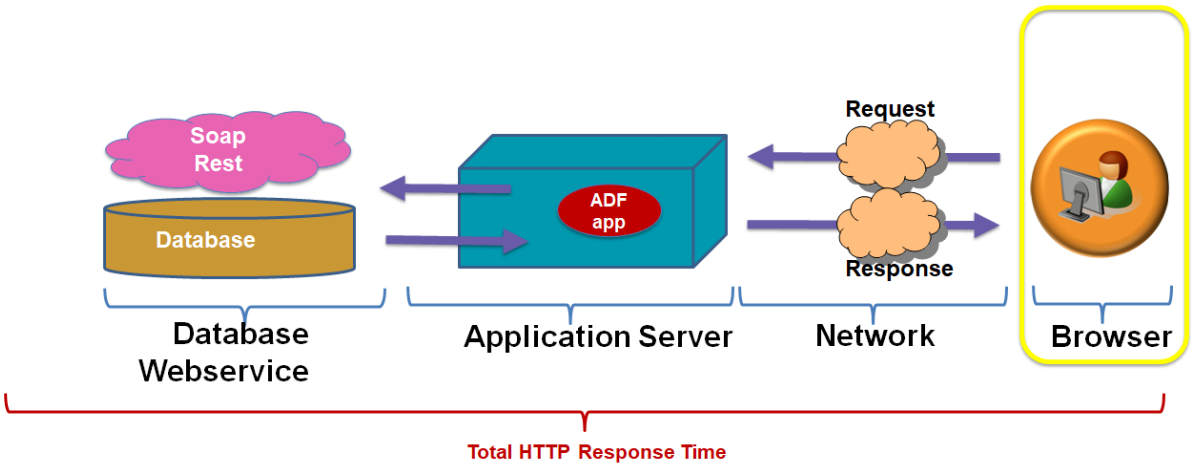


For each time range, the ADF Performance Monitor (Figure 15, bottom-right box) shows the layer where the most time is spent, indicating time spent with the database, webservice, appserver, network, and browser.

Figure 15: In the top-right graph (hourly performance), there are a lot of red bars. Specifically, between 11:00 and 12:00, there were many “very slow” requests. The bottom-right graph can explain what has happened: there were substantial network problems (bottom-right graph’s purple component).

Browser Load Time

Although performance problems are often found among the usual suspects - server, database, webservices, and network - substantial problems can also be located in browser load time. Browser load time is the time a browser requires to build the Document Object Model tree and load the page. Browser load time is an important part of HTTP-request and HTTP-response handling, as is time spent with the appserver, database, webservices, or network. Browser load time can add several seconds onto the server and network processing time, all of which precedes the end-user receiving the HTTP response enabling them to continue their work.



Browser load time can sometimes be exceptionally slow (see Figure 16), often caused by “bad” page design - that is, too many ADF components, such as table columns and rows, rendered and displayed simultaneously.

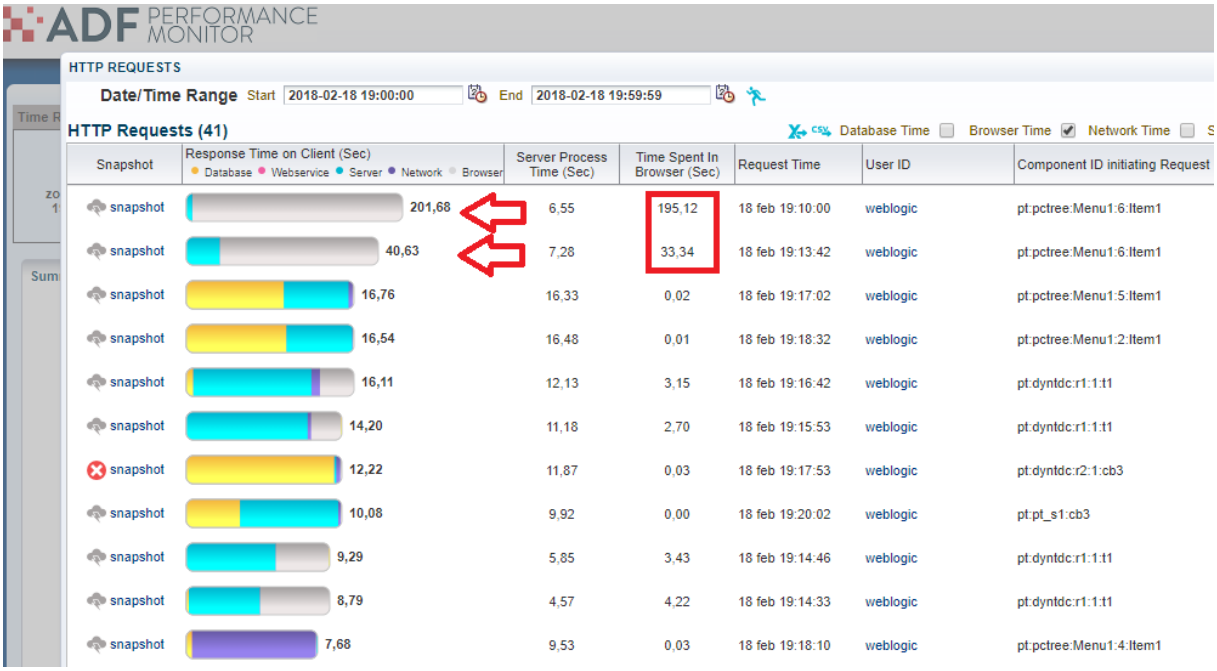


Figure 16: HTTP-request overview ordered from slowest to fastest. In this case, the two slowest have an extremely high browser load time (gray component) - specifically, we can see that those browser load times are 195 and 33 seconds.

ADF Click Actions

A click action triggers an HTTP request by the browser due to an action that a user has taken within the UI, most often physical clicks on UI elements such as buttons, links, icons, charts, and tabs, though also scrolling and selection events involving tables, chart renderings, polling events, input-field auto-submits, among others. Monitoring by click actions provides insight into the worst-performing, most error-prone, and most frequently used. That is, you can see which layer (database, webservice, appserver, network, or browser) has demanded the most execution time, ordered by total (sum) processing time, enabling SLA monitoring of the business functions behind click actions from the perspective of the end-user.

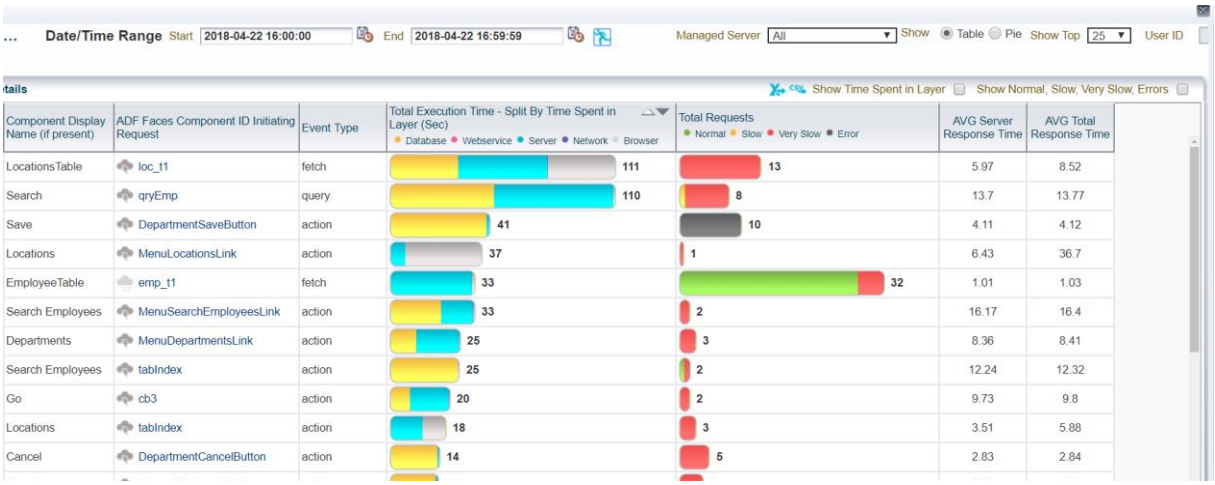


Figure 17: The component with the ID loc_t1 consumes the most sum execution time (1.8 minutes, around 110 seconds). The figure also shows that the Java class component is an *oracle.adf.RichTable*, displayed as *LocationsTable*, and that the event is a fetch event, as well as that 13 HTTP requests are “very slow” (red), constituting almost one-third of the request time spent in the browser (gray). This should trigger further investigation.

The overview shows:

- Component ID (the ADF Faces component ID that started the request);
- Component Type (the ADF Faces component Java Class);
- Display name (label/text if present on component);
- Event type (e.g., action, query, fetch, valueChange, selection, popupRemove, dvtImageView);
- Total (sum) processing time (split by time spent with database, webservice, appserver, network, and browser);
- AVG Server processing time;
- AVG End-user time (exactly as the end-user experiences it in the browser); and
- Total requests (split by SLA; rates of “normal,” “slow,” and “very slow” requests and errors).

This overview shows a high-level view of the worst-performing click action (i.e., the click action responsible for the most total processing time). Figure 18 shows an outlier; a poll action of type *oracle.adf.RichPoll*.

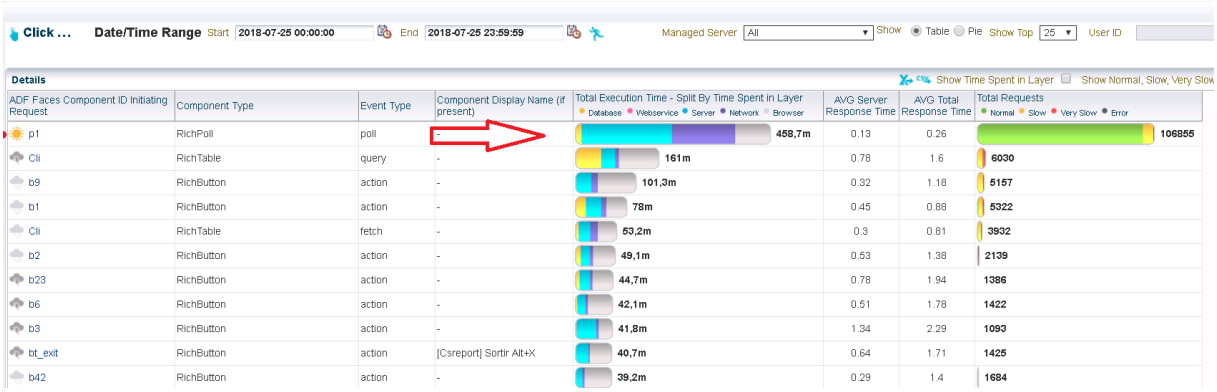


Figure 18: We see here that a poll event (the ADF Faces component of type *oracle.adf.RichPoll*, with id “p1”) is responsible for the most total processing time. On this day, there were 106.855 poll requests in total, more than one-third of all HTTP requests (296.435).

HTTP-Request Details and ADF Call Stack

After clicking on the number of HTTP requests in the dashboard, a popup shows the HTTP request details sorted by client response time in ascending order (i.e., the worst HTTP response times are at the top—see Figure 19).

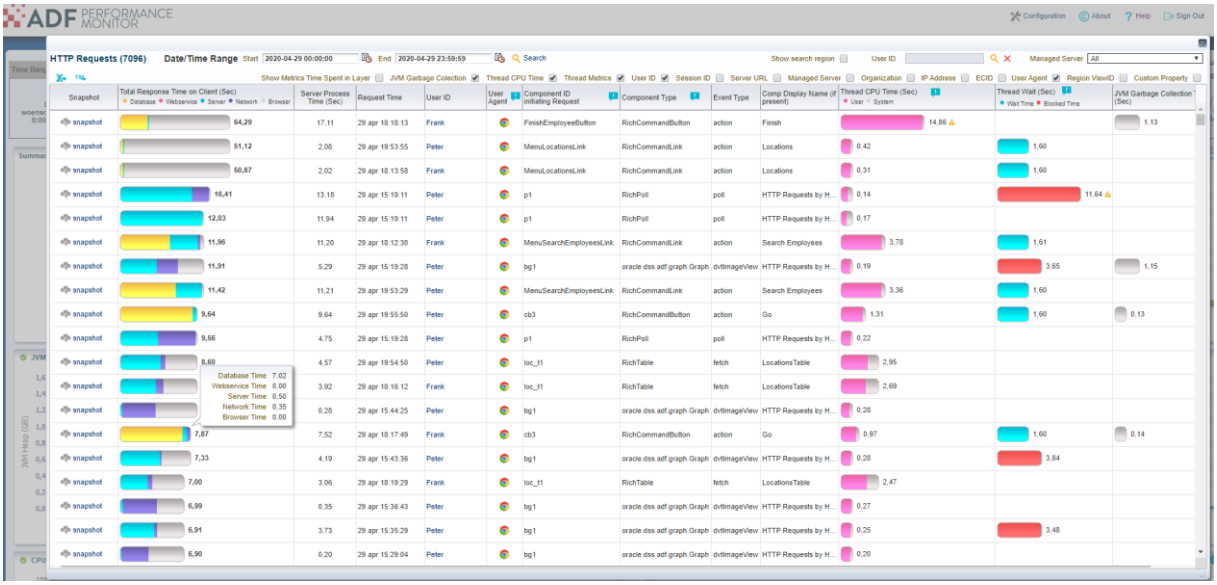


Figure 19: HTTP request details.

The monitor presents HTTP requests details for further analysis:

- total response times for the end-user:
  - time spent in appserver (blue);
  - time spent in the database (yellow);
  - time spent in web service (pink);
  - time spent in network (purple); and
  - time spent in browser (gray);
- link to ADF call stack (snapshot);
- user ID;
- request timestamp;
- component initiating request (click action Id, ADF component type, component display name);
- thread CPU time (user and system mode);
- thread wait time and blocked time;
- JVM garbage collection;
- appserver URIs;
- WebLogic managed server;
- session details (link) to drill down to all HTTP requests of this session;
- ECID;
- user agent;
- custom property; and
- user organization.

User ID Filter

If end-users (or local testers) complain about today's or yesterday's performance, the ADF Performance Monitor enables fast troubleshooting because you can filter using a specific end-user ID, and analyze the performance for that specific user ID. Also, the product provides overviews of the best-and-worst-performing end-users, providing a genuine representation of every individual user through logs of their every interaction (like clicks and scrolls) and records of every session.

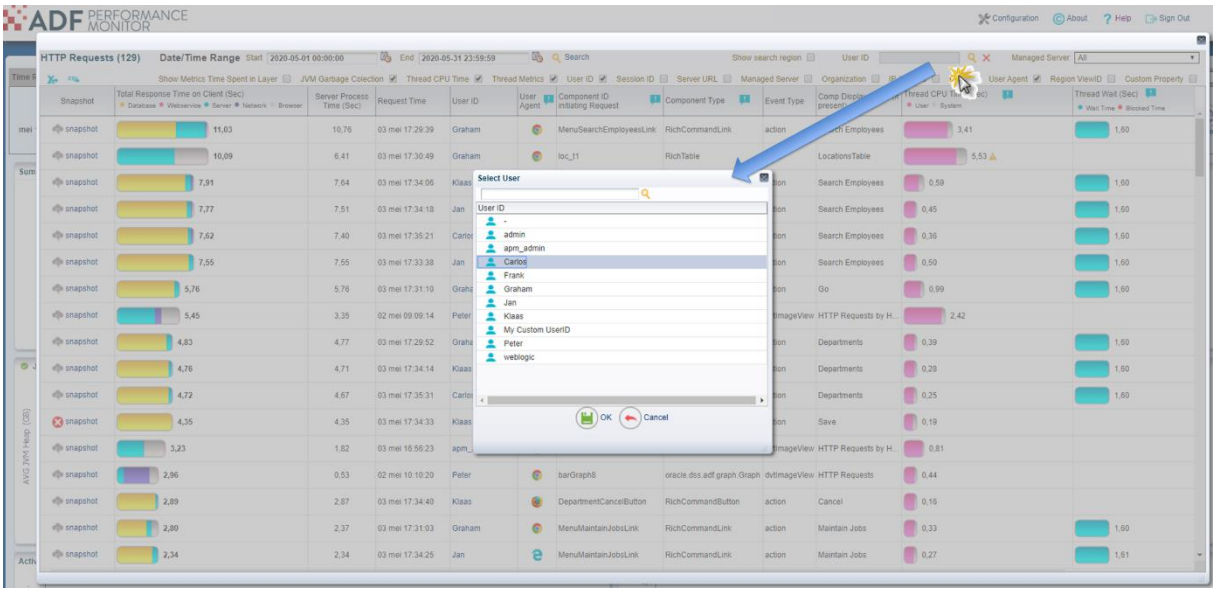


Figure 20: Filter for a specific user ID.

Thread Waited and Blocked Time

In the world of Java threads, we can often explain poor performance or interruptions: certain threads might have been waiting on a particular resource, such as a data source from the data-source connection pool, or might have been blocked. We can inspect this, in detail, for each HTTP request. We have much more insight into time gaps that were sometimes hard to explain before.

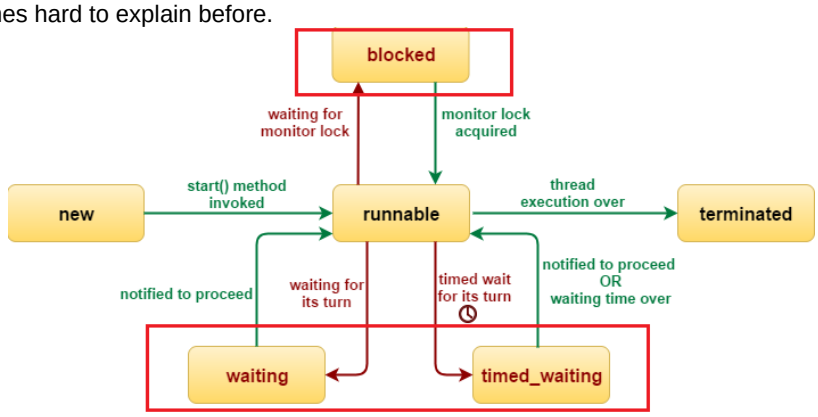


Figure 21

- Waiting State: a thread that is waiting (indefinitely) for another thread to perform a particular action. For example: under high load, if all datasources of a datasource connection pool are already occupied, the next thread will need to wait for one to become available from the pool, and will be in a waiting state;
- Blocked State: a thread that is waiting for a monitor lock to enter a synchronized block/method. If program code or code in a (framework) library calls an (expensive) synchronized method, other threads will be blocked until the thread finishes.

Waiting/Timed Waiting State

The monitor shows for each HTTP request the waiting and times-waiting state together:

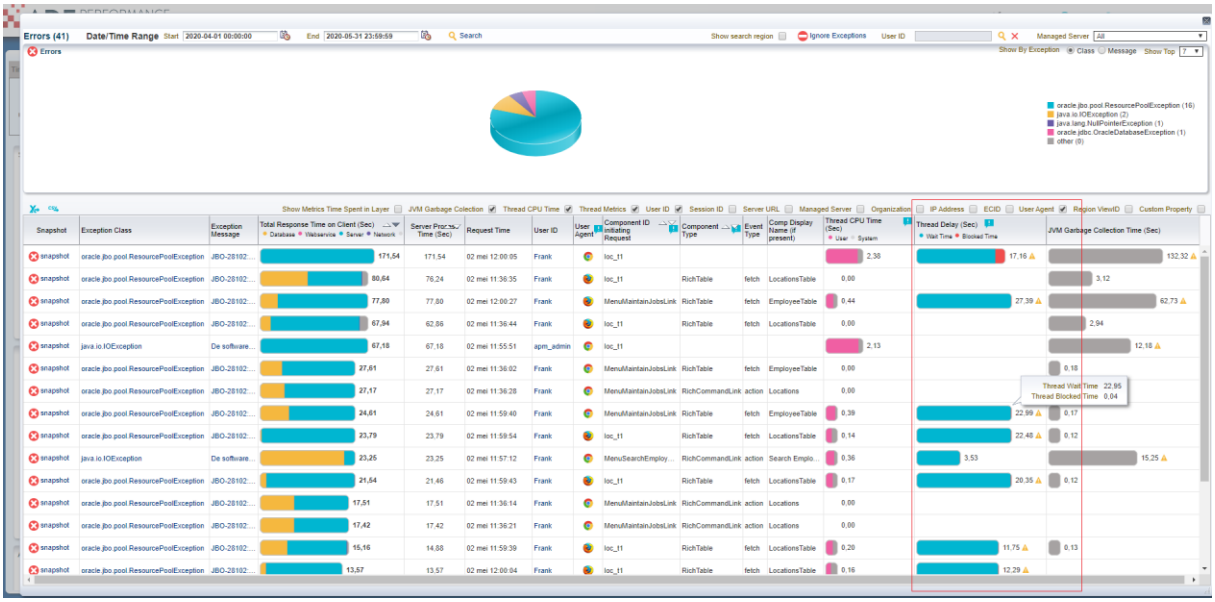


Figure 22: Many requests that were in the waiting state, timed out, and were thrown as errors.

In this screenshot (Errors Overview) we see many requests that were in the waiting state for more than 20 seconds(!). They had to wait for a database connection to become available from the application server. Datasources are being shared in a pool. If all datasources are already being used under heavy load, and a new request for a datasource comes, this thread needs to wait until one becomes available for use. If this takes too long, the request will timeout, and throw an exception (this is what we see in the screenshot):

oracle.jbo.pool.ResourcePoolException: JBO-28102: A request timed out while waiting for a resource from the resource group hr.demo.model.service.HRServiceLocal to be returned.

Blocked State

In the following screenshot (HTTP Requests Overview) we can see some threads that were in the blocked state (red colour), for more than 10 seconds(!):

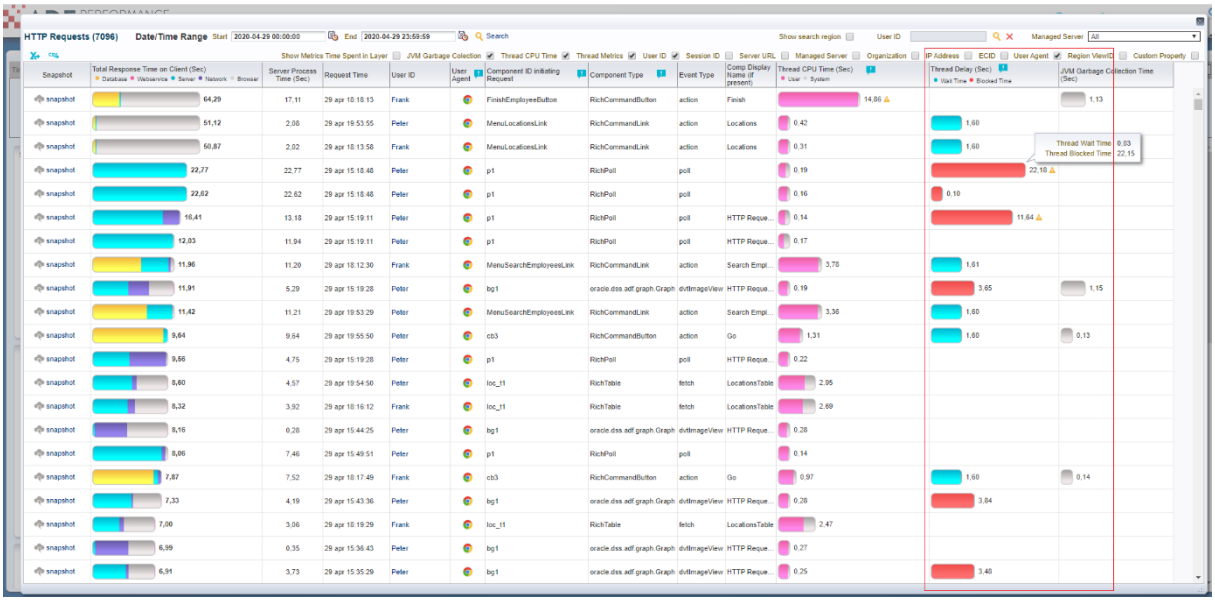


Figure 23: Many thread were being blocked for several seconds (red colour).

Apparently, many times a blocked state was eventually started by a poll event, coming an ADF Faces component oracle.adf.RichPoll, with ID “p1”.

Thread CPU Time of Requests

"What request/click action in the application is responsible for burning that CPU ? " That question can be answered with the monitor. With this we can determine their load on the CPU's of the underlying system. It gives a clear indication how expensive certain requests are in terms of CPU cost. In the HTTP requests overview, for each request (or click action) the User CPU time and the System CPU time is shown:

- User CPU time. This is time spent on the processor in user mode running your program's code (or code in libraries) of the HTTP request/click action (thread); and
- System CPU time. This is the total time spent running code in user mode as well as in kernel mode on behalf of the program request (thread).

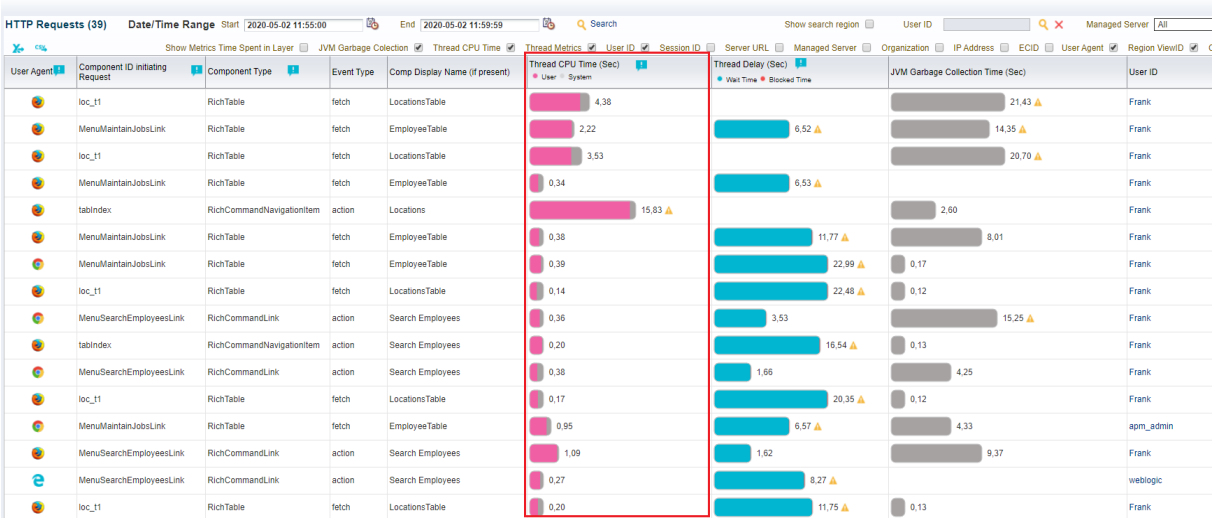


Figure 24: In the HTTP requests overview, the thread CPU (user and system mode) are shown, along with Thread Wait and Thread Blocked Time

A very high value (for example more than 5 seconds) indicates that very expensive program code is executed. For example:

- expensive long loops through large collections in Java code;
- retrieving/fetching large collections of database rows from the database. And do some processing for each row;
- repetitive ViewObject queries;
- repetitive (redundant) ViewAccessor executions; and
- rendering of tables with many columns and rows in the render response phase.

In this case you should investigate these further, for example by looking into the ADF callstacks of the ADF Performance Monitor what is happening behind the scenes.

ADF Call Stacks

A call stack (Figure 24) makes visible which ADF method caused other methods to execute; it is organized by execution sequence and includes runtime method and query parameter values. This is the same call stack printed in JDeveloper during development. A complete breakdown of the HTTP request is shown by actions in the ADF framework (e.g., Fusion life-cycle phases, model (BindingContainer), and ADF BC executions, start and end of task flows), with elapsed times and a representation of the sequence of events. The time-consuming parts of the ADF request are highlighted and indicated by an alert signal.

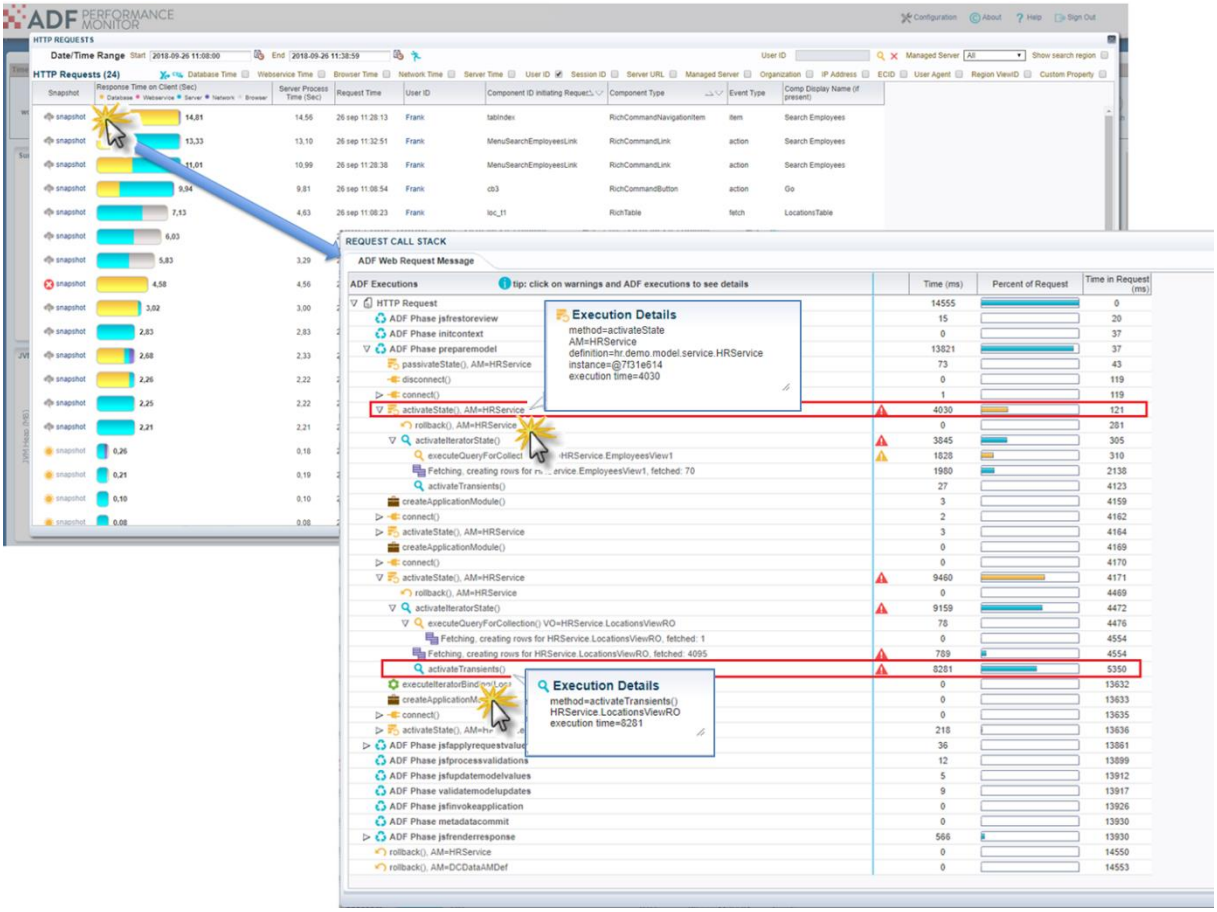


Figure 24: The call stack shows the slow activation of the ApplicationModule called HRService (4030 milliseconds). Later, in the HTTP call stack, we see slower activation of a different HRService instance (9,460 milliseconds), caused by the slow activation of transient attributes in activateTransients() of ViewObject HRService.LocationViewRO (8,281 milliseconds). We can also see that there are 4095 rows of LocationViewRO fetched (!).

When an execution is clicked, a popup shows details such as ViewObject usage name, ViewObject definition name, bind variable values, applied ViewCriteria names, the number of rows fetched from database, and ADF object names.

Troubleshoot Slow SQL Queries

In the configuration settings of the ADF Performance Monitor, a parameter can be set for whether the monitor should log the SQL and DML statements of ADF Business Components or EJB/JPA. If it should, a threshold can be set for when it should be collected. For example, the parameter can be set for all SQL Queries and SQL DML statements over 5 seconds (5000 milliseconds). Then, runtime SQL executed against the database can be analyzed (Figure 25).

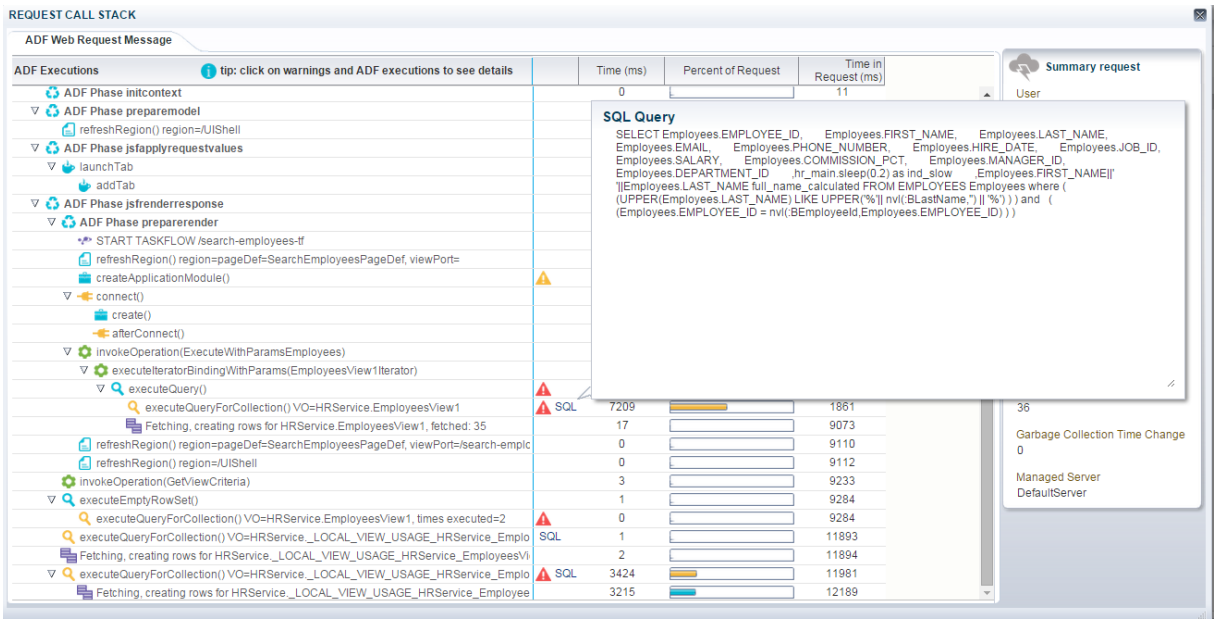


Figure 25: ViewObject SQL executed runtime against the database. The runtime bind parameter values are also made visible using the ViewObject *executeQueryForCollection()* method.

ADF BC Memory Analyzer

Like other web applications, ADF applications potentially use a lot of memory. Often, the root cause of high memory usage is improper limitations of the application data retrieved from the database into the working memory; that is, too many rows with too many attributes are fetched and held in memory. The ADF BC Memory Analyzer (Figure 26) detects how many database rows are fetched by ADF ViewObjects. Loading too many database rows can lead to memory overconsumption. When a ViewObject loads more than 250 rows (configurable) into memory, the monitor gives a warning sign in the call stacks. It also suggests solutions like using ViewObject range paging or setting an appropriate (maximum) fetchsize on the ViewObject.

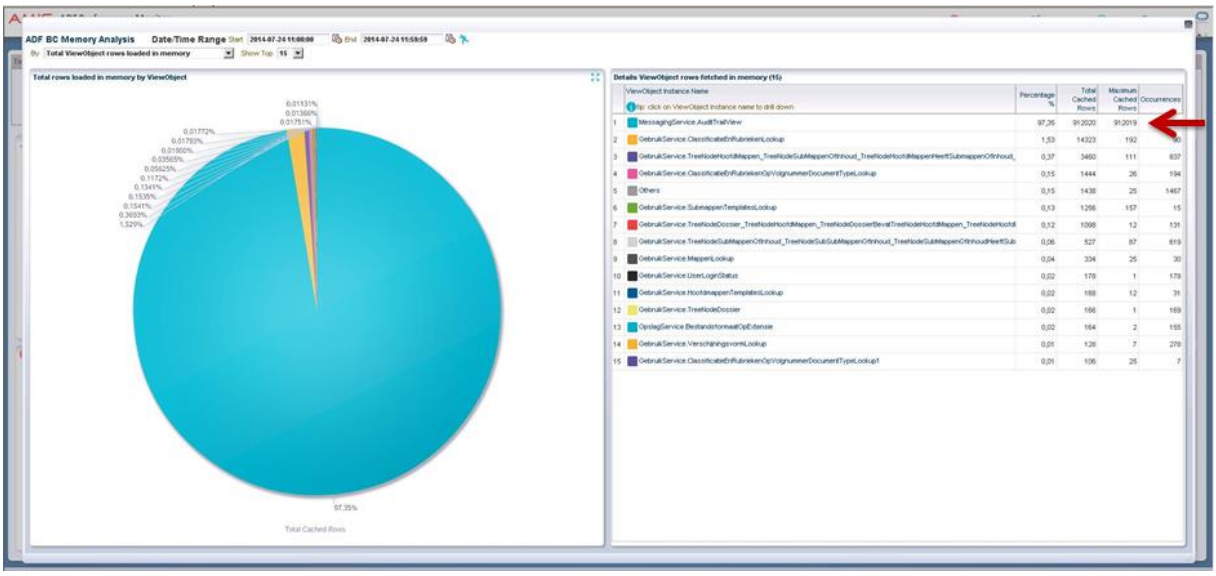


Figure 26: ADF BC Memory Analyzer. In this production ADF app, for a single ViewObject instance, more than 900,000 rows were loaded (in blue).

Warnings and Suggested Solutions

Throughout the application life cycle, QA teams can ensure problems are identified and dealt with. They can routinely check to ensure the application meets the desired quality. In the call stacks, clickable warning signs (Figure 27) show slow executions and detected inefficiencies and offer efficient assistant with suggested solution(s) and links to more detailed advice.

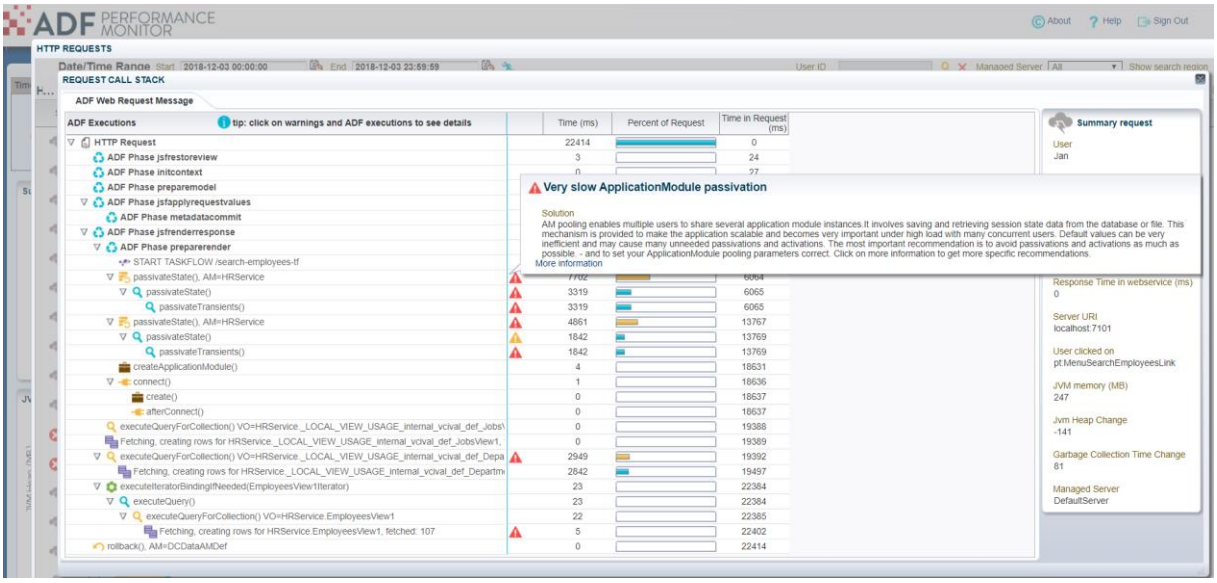


Figure 27: Very slow ApplicationModule passivation. By clicking on the warning image, a quick-help popup shows a suggested solution. More detailed help is also available.

For example, warnings and suggested solutions are shown in the following cases:

- slow executions:
    - ViewObject queries;
    - EntityObject DML operations;
    - ApplicationModule activations, passivations, transactions;
    - PL/SQL calls;
    - BindingContainer operations, iterator executions;
    - Java methods; and
    - Webservice calls;
- slow passivation and activation of transient ViewObject attributes;
- fetching an exceedingly high number of rows from database into Java memory;
- multiple redundant ViewObject query executions during the same HTTP request;
- inefficient ViewObject fetchsize; and
- inefficient BindingContainer iterator rangesize.

End-User Metrics

There are several overviews of summarized end-user metrics: Worst Performance by User, Top Users by Requests, and Most Errors by Users (Figure 28).

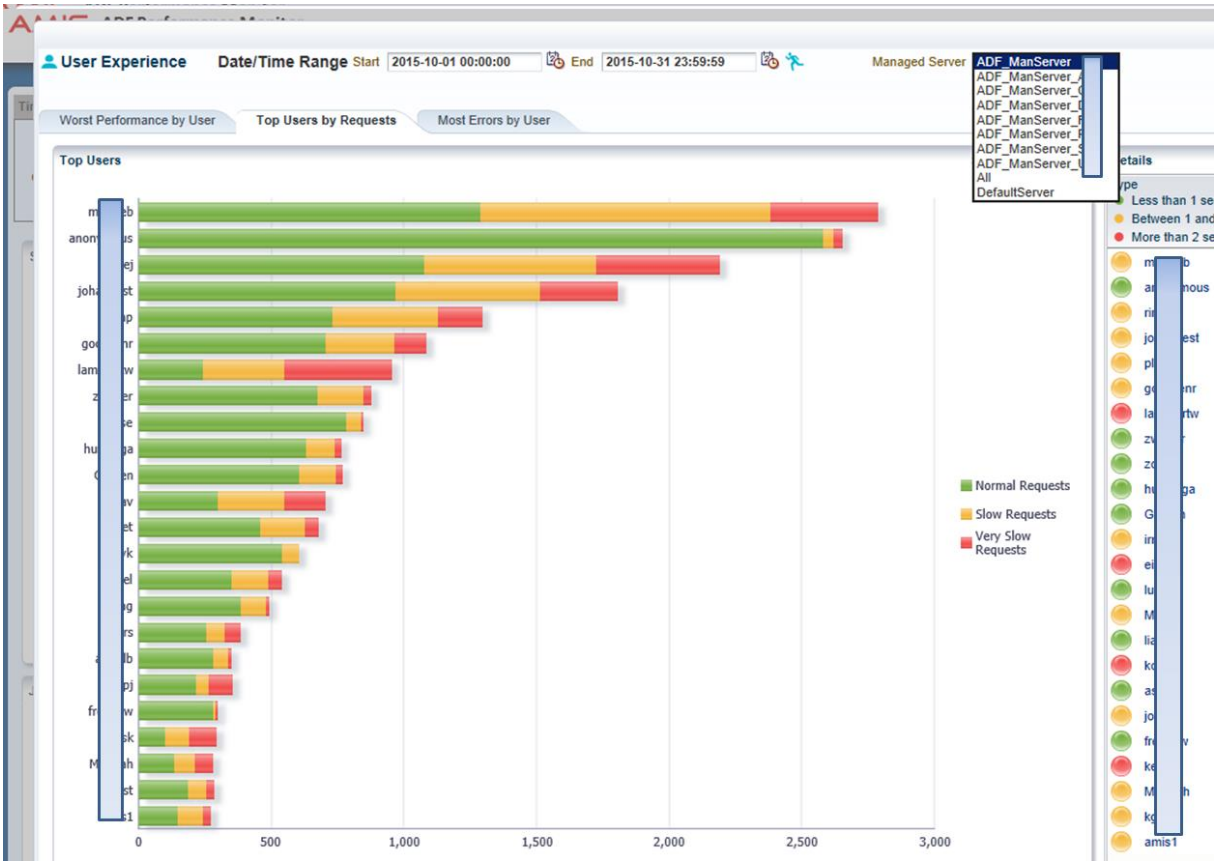


Figure 28: There are several overviews of summarized end-user metrics: Worst Performance by User, Top Users by Requests, and Most Errors by Users.

Worst Performing Executions in ADF BC and ADF Model Layer

Various overviews present the worst-performing components of the ADF BC and ADF model layer, indicating bottlenecks and allowing the development team to act to improve the application. The following overviews are available:

- ADF business components;
- (PageDefinition) BindingContainer executions; and
- Webservices (calls to JAX-WS webservices);

Worst ADF BC Executions

Figure 29 is an example of an ADF BC overview. It shows “very slow,” “slow,” and “normal” ADF BC executions:

- ApplicationModule pooling passivations/activations;
- ApplicationModule transactions;
- ViewObject queries;
- EntityObject DML operations; and
- PL/SQL procedure/function calls executed from ApplicationModules.

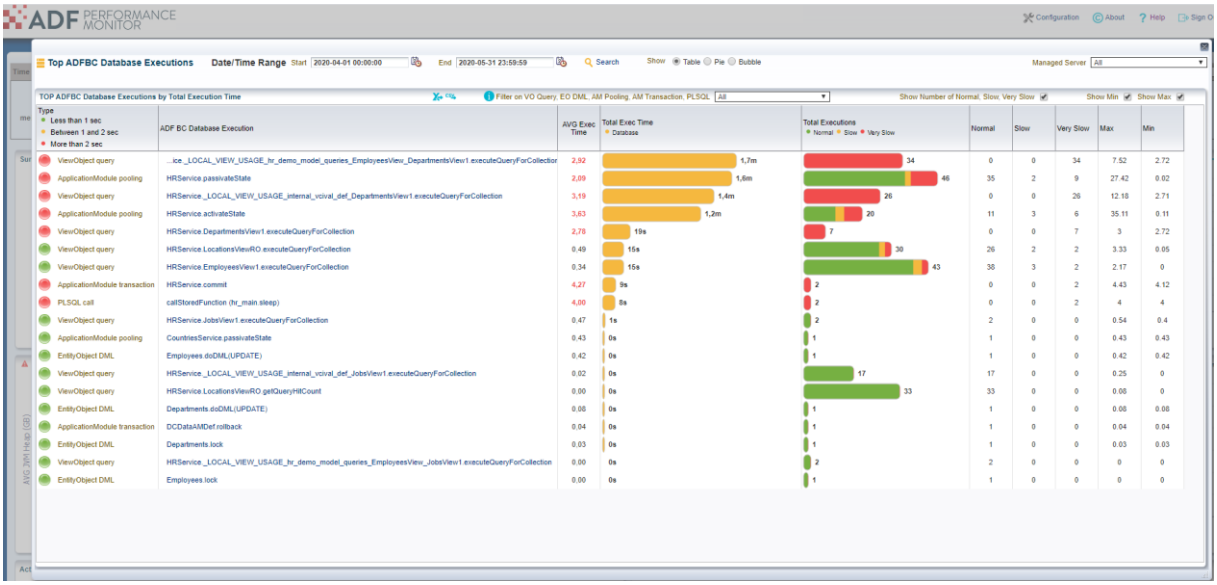


Figure 29: Overview of ADF Business Components by AVG execution time. First, a marker (red, orange, and green) is shown (configurable). Then the type, the usage (instance) name, the AVG execution time, occurrences, and the total time are shown. The occurrences and their call stacks can be further analyzed in a popup window.

ApplicationModule Pooling Analysis

It is important to choose the right combination of ApplicationModule pooling parameter settings to make the application scalable. Incorrectly setting these parameters (and leaving them at the default values) can be very inefficient and may produce undesirable passivations and activations. In the ADF BC overview (filtered on ApplicationModule pooling), the ApplicationModule pooling performance can be analyzed. This overview provides insight into the effect of these parameter settings (how often passivations and activations happen and how long their AVG and total execution times are), enabling QA teams to research and experiment with parameter settings on test environments during load tests before evaluating the performance results using the ADF Performance Monitor. The monitor also detects which VO data are being activated and passivated. The root cause of long-running activations/passivations is often unnecessarily calculated and/or transient attributes in combination with an unnecessarily high number of fetched database rows.

Worst BindingContainer Executions

The monitor shows “very slow,” “slow,” and “normal” BindingContainer executions (Figure 30):

- operations (all standard ADF operations, client methods on ApplicationModule and ViewObject); and
- iterators from the executable section.

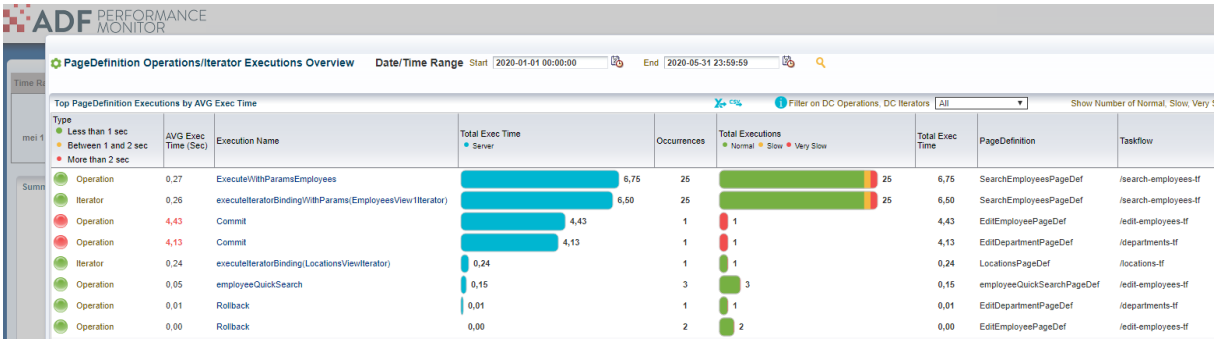


Figure 30: Overview of the most time-consuming executions from the BindingContainer. In this case, they are operation *ExecuteWithParamsEmployees* and iterator execution of *EmployeesView1Iterator*.

Total Execution Time of Worst-Performing Executions

The performance impact of frequently invoked executions can be much higher than executions that occur only a few times but are slow on average. For example, a “slow” ViewObject query with an average time of 1500 milliseconds that is executed 10,000 times a day has more impact than a “very slow” ViewObject query with an average time of 15 seconds that is executed only twice a day. There are additional overviews for the total execution time.

Support for EJB/JPA (EclipseLink)

Not all Oracle ADF projects are built with ADF Business Components. Some organizations use EJB/JPA (EclipseLink) instead of ADF BC. The ADF Performance Monitor supports EJB/JPA. Among others, EJB/JPA DataControl operations/methods, runtime invoked methods on EJB SessionBeans, and SQL queries/DML actions executed are instrumented and visible on the ADF call stacks.

Support for Soap/REST

Webservices are supported. All webservice calls are instrumented and visible on the ADF call stacks. Time spent on webservice calls is shown on the main dashboard page.

Export to Excel or CSV File

You can export all monitor overviews to an Excel or CSV file, enabling filing for later analysis or trend analyses after new releases to check for improved performance.

Product Architecture

How It Works

The product comprises an ADF library, a database schema, and a dashboard reporting application. The ADF library, which attaches to your application, collects runtime behavior and performance metrics. The metrics are saved in a separate database schema. The dashboard reporting application retrieves metrics from the same database schema. (Figure 31).

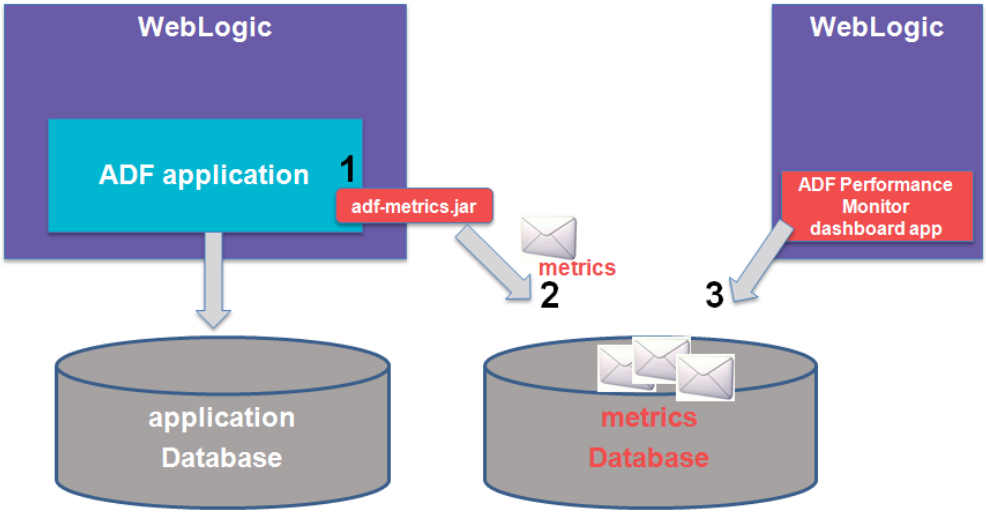


Figure 31: Schematic of the ADF Performance Monitor's architecture.

1. Metrics Collection

Raw metrics data are collected from every HTTP request by the attached ADF library. When the HTTP request is finished, the metrics data are buffered.

2. Metrics Logging

In the background, metrics are saved in an Oracle database. This database can be a separate database but can also be the same database that is used by the ADF application.

3. Dashboard Reporting Application

The dashboard reporting application is a web-based ADF application that can be accessed from any supported browser. This application can be deployed on a separate WebLogic managed server, or in the same managed server as the ADF application. It communicates with the metrics database.

Monitored Events

The monitor collects performance information from specific ADF objects.

Overhead

Load tests show that the overhead is less than 2%. This overhead is caused by the detailed collection of performance metrics. Metrics collection can always be turned on and off dynamically (at runtime). When the monitor is turned off, there is no performance overhead because the metrics classes are inactive.

Clear Error Messages

The ADF Performance Monitor logs clear error messages in the Enterprise Manager when required resources are unavailable (for example, if the metrics database is offline) or when an error occurs. This does not cause application errors in the ADF application.

Configuration

Prerequisites

The ADF Performance Monitor can be implemented for all ADF10g, ADF11g, and all ADF12c applications. Any Oracle11g or newer (also the free XE) database can be used for the metrics database. The use of ADF BC requires ADF BC framework extension classes.

Configuration

- An ADF library needs to be added to your ADF application, and a small amount of code needs to be added to instrument the ADF application. The metrics classes use extension points in the ADF framework to track the time required of every action or step in an HTTP request.
- In a test or production environment, a JDBC datasource must be configured on the managed server that connects to the metrics database.
- The dashboard application EAR file must be hosted on a WebLogic server. This EAR can also be run locally in JDeveloper.

Custom Instrumentation

It is possible to instrument (calls to) custom Java methods and third-party libraries. Custom methods are visible on the call stacks.

Always Turn On/Off

The ADF Performance Monitor uses the standard Oracle *ADFLogger* and can always be turned on and off at runtime in the Enterprise Manager.

Installation Guide

All configuration steps are well documented in the installation guide. Implementation takes less than one day and usually no more than a few hours. A user manual is available within the dashboard application.

Training

We can provide consulting and training on the ADF Performance Monitor, ADF performance tuning, and ADF best practices (on location or through an online webcast).

Website

All information concerning this product is available on our website: adfpm.com.

Use in JDeveloper During Development

The ADF call stacks of the ADF Performance Monitor can be used during development, enabling developers to diagnose and solve performance problems at an early stage and to build an efficient, responsive ADF application before production. In JDeveloper, ADF developers can switch metrics logging on and off using Oracle's standard ADFLogger. For every HTTP request, a call stack is printed. A call stack makes visible which ADF methods executed other methods, organized by execution sequence.

A complete breakdown of the ADF request processing, including all ADF method executions and elapsed times, is printed, organized by the life-cycle phase. Inefficiencies can be identified based on this report, as in the case of the long-running (31,858 milliseconds) ViewObject query seen in Figure 32.

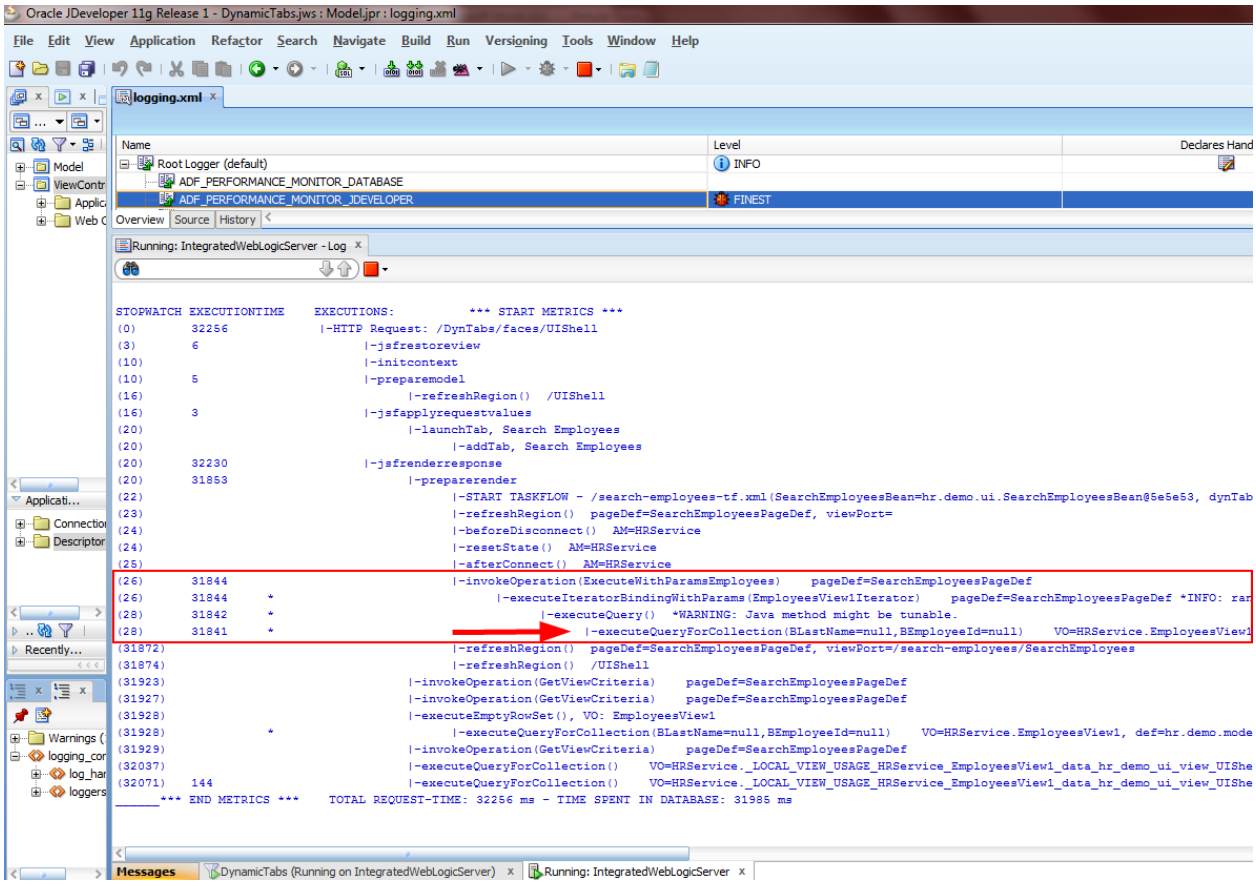


Figure 32: ADF metrics are printed in JDeveloper WebLogic Server Log (can be switched on and off). Shown is an example of a long-running ViewObject query (31,858 milliseconds; usage name of HRService.EmployeesView1).

Conclusion

The ADF Performance Monitor is a tool for measuring, analyzing, tuning, and checking the performance of Oracle ADF applications. Throughout the application's life cycle, the development, QA, and operation teams gain invaluable insights into the functioning of their ADF application. These insights enable ADF developers to diagnose and solve performance problems at an early stage, make better architectural decisions, and build more responsive and scalable ADF applications. Using warnings and suggested solutions, they can circumvent common and uncommon performance problems, implement best practices, and deliver a higher quality product. Ultimately, this results in a more consistent and easier-to-maintain application. Finally, the ADF Performance Monitor can reduce infrastructure, hardware, and license demands.

With performance overheads of only 2%, the ADF Performance Monitor can be easily implemented in production environments, ensuring performance bottlenecks and errors are proactively monitored and resolved.

Resources

More information is available at adfpm.com, including a demo video.

Disclaimer

The information in this white paper is intended for information purposes only and may not be incorporated into any contract. It is not a commitment to deliver any material, code, or functionality and should not be relied upon when making purchasing decisions. The development, release, and timing of any features or functionality of this product remain at the sole discretion of AMIS/ADF Performance Monitor.

Appendix A: Comparison with Other Tools

Application performance tools on the market are generally not designed for the intricacies of the ADF framework. They focus on generic, low-level JVM metrics or on high-level business transactions (like “add to cart” or “pay bill”), missing the relevant key-action and resource-usage metrics of the ADF framework.

Oracle provides some support for performance analysis with WebLogic Enterprise Manager. Enterprise Manager provides only basic monitoring information for ADF applications. Also, in JDeveloper, there is some support for performance analysis through the ODL Analyzer. This tool can be useful in the development stage but cannot be used in test and production environments because of the amount of logging generated and the significant performance overheads.

The table below shows a comparison between different monitoring tools in the ADF landscape.

Monitoring Features	**WebLogic Enterprise Manager**	**Oracle Real User Experience Insight**	**Generic APM Tools (New Relic, AppDynamics)**	**ADF Performance Monitor**
High-level HTTP Requests Overview	±	+	+	+
Detailed HTTP Requests Diagnostics	-	+	-	+
Filter on User ID (Select User Sessions of Interest, Current, or in History)	-	+	-	+
Insight in Real End-User Experience	-	+	+	+
Time Spent in Database, Appserver, Webservice, Network/Browser Load Time	-	-	+	+
JVM Metrics	+	-	+	+
Error Diagnostics	+	±	+	+
ADF Request Call Stack (Code Level Visibility of ADF Key Actions)	-	-	-	+
ADF Business Components Memory Analyzer	-	-	-	+
ADF Specific Warnings and Suggested Solutions	-	-	-	+
Worst ADF BC Slow SQL Queries (Runtime SQL Query and Bind Variables)	-	-	±	+
Worst ADF BC ApplicationModule Pooling (Passivations and Activations)	-	-	-	+
Worst ADF BC EntityObject DML Operations	-	-	-	+
Worst PL/SQL Procedure/Function Calls			±	+
Worst (PageDefinition) BindingContainer Executions	-	-	-	+
Support for EJB/JPA (EclipseLink)	-	-	±	+
Support for JAX-WS/Apache CXF Webservices	-	-	-	+
Diagnostics and Warnings in JDeveloper During Development	-	-	-	+
Performance Overhead	0%	0%	2%	2%