Title: WebLogic JDBC execution watchdog, version 0.42
Date: March 24, 2015 at 1:18 PM

# WebLogic JDBC monitor

Ryszard Styczynski, April 2015, version 0.42

*WebLogic provides rich JDBC monitoring and diagnostics capabilities (debug, WLDF), however does not provide lightweight facility to report executed SQL statements lasting longer than predefined time. Moreover available out of the box subsystems are little too heavy, in area of configuration and generate too much information. Practical use case is to be informed only about SQL executions which consumed too much time. Presented solution delivers such lightweight SQL monitoring functionality.*

***Note that it's an early version of the software. I've made a lot of work to tune and ensure that software is free of problems, but it's only software. Use it with caution.***

## JDBC monitor

Missing capability of WebLogic is filled with JDBC monitor utilizing interceptor interface exposed by WLS JDBC layer. The JDBC monitor traces all JDBC executions to extract SQL command, related variables, and final execution time. In case of execution longer than predefined threshold, complete information related to such execution is logged in an alert log.

JDBC monitor is shipped with user interface presenting real time SQL information and latest alerts. Note that user interface hows not yet completed SQL executions, while alert log shows only completed ones. In case of stuck execution on execute() JDBC method web interface is the only way to find out such situation.

## Alert log

All executions lasting more that defined threshold are reported to ODL log file. ODL log files are located in server's log directory. SQL alert file is suffixed by "-`sql[1-5].log`", where number at the end of file name indicates interceptor number. It's important as operator may configure different log files for different data sources.

```
$DOMAIN_HOME/servers/$SERVER_NAME/logs/$SERVER_NAME-sql1.log
```

Alert with too long SQL statement execution contains all standard Oracle Diagnostic Logger (ODL) information plus JDBC interceptor alert data. Alert is logged on Warning level.

```
[2015-03-24T08:50:44.017-07:00] [SOA_Server1] [WARNING] [] [weblogic.jdbc.debug] [tid: Workmanager: ,
Version: 0, Scheduled=false, Started=false, Wait time: 0 ms\n] [userId: <anonymous>] [ecid:
cd7222d84368db0c:41d95b8a:14c4c1494db:-8000-0000000000000217,0] [APP: soa-infra] SQL execution lasted more
than expected.[[
|------lasted [ms]:1
|------SQL:SELECT t1.ID, t1.SHORT_EXCEPTION_MSG, t1.PROPERTIES, t1.RECOVERABLE, t1.EXCEPTION_TYPE,
t1.CREATED_TIME, t1.CALLBACK_OPERATION, t1.CONTAINER_ID, t1.TARGET_ACTION_NAME, t1.FAULT_OBJ,
t1.TARGET_REFERENCE, t1.STEP, t1.TARGET_TYPE, t1.RETRY_INTERVAL, t1.CASE_NAME, t1.EXCEPTION_MSG,
t1.SYSTEM, t1.LOCK_TIME, t1.CONVERSATION_ID, t1.FAULT_NAME, t1.DEF_MESSAGE_ID, t1.EXCEPTION_TRACE,
t1.INSTANCE_ID, t1.STATUS, t1.TENANT_ID, t1.MI_PARTITION_DATE, t1.RETRY_COUNT, t1.SOURCE_URI, t0.CASE_ID,
t0.CONTAINER_ID, t0.DUMMY1, t0.MSG_ID, t0.COMPONENT_DN, t0.OPERATION, t0.CREATION_DATE, t0.PRIORITY,
t0.COMPONENT_STATUS, t0.QNAME_LOCAL_PART, t0.CASE_INFO, t0.QNAME_NAMESPACE, t0.LOCK_TIME, t0.SOURCE_URI,
t0.TENANT_ID, t0.INSTANCE_CREATED, t0.IS_EVENT, t0.STATUS FROM MEDIATOR_DEFERRED_MESSAGE t0,
MEDIATOR_CASE_INSTANCE t1 WHERE ((((t1.STATUS = ?) AND (t1.LOCK_TIME = ?)) AND (t1.CONTAINER_ID = ?)) AND
(t0.CASE_ID = t1.DEF_MESSAGE_ID))
|------with following modifiers:
       \------0: [name=setPoolable, parameters=[true]]
       \------1: [name=setObject, parameters=[1, locked]]
       \------2: [name=setObject, parameters=[2, 2015-03-24 08:50:44.013]]
       \------3: [name=setObject, parameters=[3, 221D7810D22F11E4BF91DF392C48EAB9]]
       \------4: [name=executeQuery, parameters=[]]

]]
```

Next to execution time, and SQL statement, all prepared statement's variables, and other executed methods are presented. Order of presentation is an order of executing them on prepared statement object.

JDBC monitor is shipped with one main interceptor, and five inherited classes using names suffixed by number 1..5. Such solution provides ability to configure different interceptors for different data sources, with separated log file for each of them. Additionally user interface presents interceptor class name next to SQL, what gives information about associated data source.

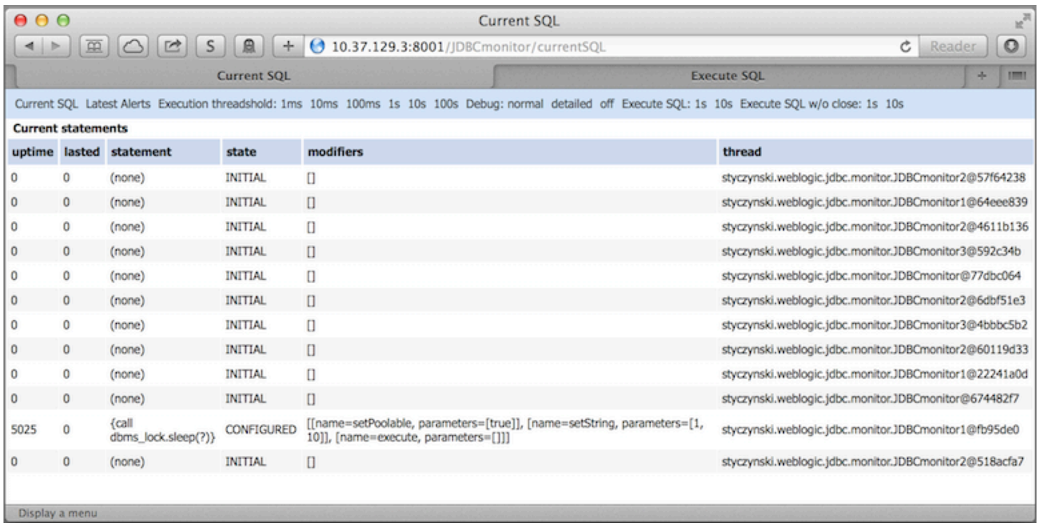
All interceptors share single debug log file. Apart of debug information you will find all content from `*sql[12345]*.log` files in the debug log.

## User interface

User interface provides two main functions: (1) view on currently active SQL statements, and (2) view on latest alerts.

View on active SQL is available under

```
http://<host>:<port>/JDBCmonitor/currentSQL
```



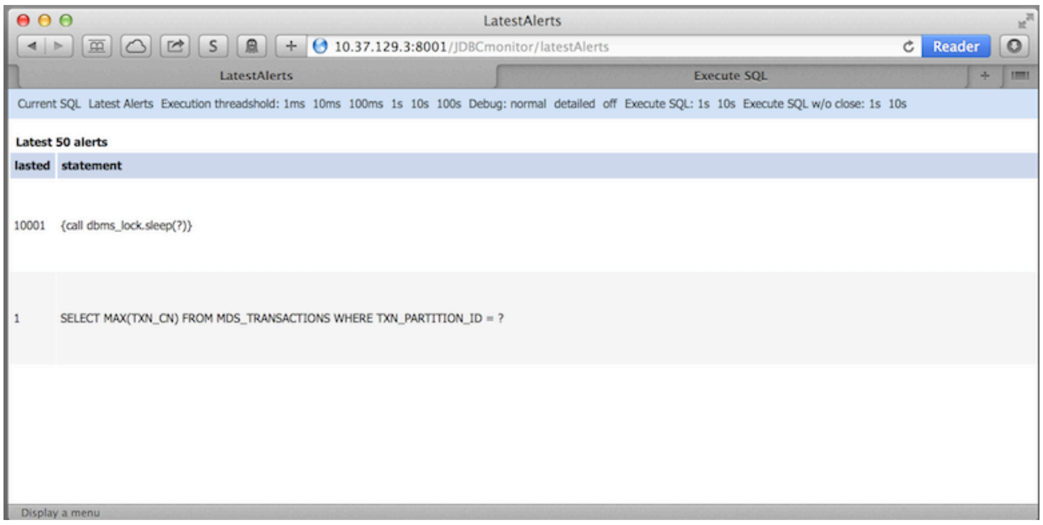
Current SQL  Latest Alerts  Execution threadshold: 1ms  10ms  100ms  1s  10s  100s  Debug: normal  detailed  off  Execute SQL: 1s  10s  Execute SQL w/o close: 1s  10s

**Current statements**

| uptime | lasted | statement | state | modifiers | thread |
|---|---|---|---|---|---|
| 0 | 0 | (none) | INITIAL | [] | styczynski.weblogic.jdbc.monitor.JDBCmonitor2@57f64238 |
| 0 | 0 | (none) | INITIAL | [] | styczynski.weblogic.jdbc.monitor.JDBCmonitor1@64eee839 |
| 0 | 0 | (none) | INITIAL | [] | styczynski.weblogic.jdbc.monitor.JDBCmonitor2@4611b136 |
| 0 | 0 | (none) | INITIAL | [] | styczynski.weblogic.jdbc.monitor.JDBCmonitor3@592c34b |
| 0 | 0 | (none) | INITIAL | [] | styczynski.weblogic.jdbc.monitor.JDBCmonitor@77dbc064 |
| 0 | 0 | (none) | INITIAL | [] | styczynski.weblogic.jdbc.monitor.JDBCmonitor2@6dbf51e3 |
| 0 | 0 | (none) | INITIAL | [] | styczynski.weblogic.jdbc.monitor.JDBCmonitor3@4bbbc5b2 |
| 0 | 0 | (none) | INITIAL | [] | styczynski.weblogic.jdbc.monitor.JDBCmonitor2@60119d33 |
| 0 | 0 | (none) | INITIAL | [] | styczynski.weblogic.jdbc.monitor.JDBCmonitor1@22241a0d |
| 0 | 0 | (none) | INITIAL | [] | styczynski.weblogic.jdbc.monitor.JDBCmonitor@674482f7 |
| 5025 | 0 | {call dbms_lock.sleep(?)} | CONFIGURED | [[name=setPoolable, parameters=[true]], [name=setString, parameters=[1, 10]], [name=execute, parameters=[]]] | styczynski.weblogic.jdbc.monitor.JDBCmonitor1@fb95de0 |
| 0 | 0 | (none) | INITIAL | [] | styczynski.weblogic.jdbc.monitor.JDBCmonitor2@518acfa7 |

View on latest alerts is available under

```
http://<host>:<port>/JDBCmonitor
```
or
```
http://<host>:<port>/JDBCmonitor/latestAlerts
```



Current SQL  Latest Alerts  Execution threadshold: 1ms  10ms  100ms  1s  10s  100s  Debug: normal  detailed  off  Execute SQL: 1s  10s  Execute SQL w/o close: 1s  10s

**Latest 50 alerts**

| lasted | statement |
|---|---|
| 10001 | {call dbms_lock.sleep(?)} |
| 1 | SELECT MAX(TXN_CN) FROM MDS_TRANSACTIONS WHERE TXN_PARTITION_ID = ? |

Provided menu makes it easy to quickly switch between both views.

Next to standard options, console menu contains possibility to configure sensitivity, and debug level. Use detailed debugging with caution, as amount of logged information is quite big, dramatically slowing down server JDBC operations.

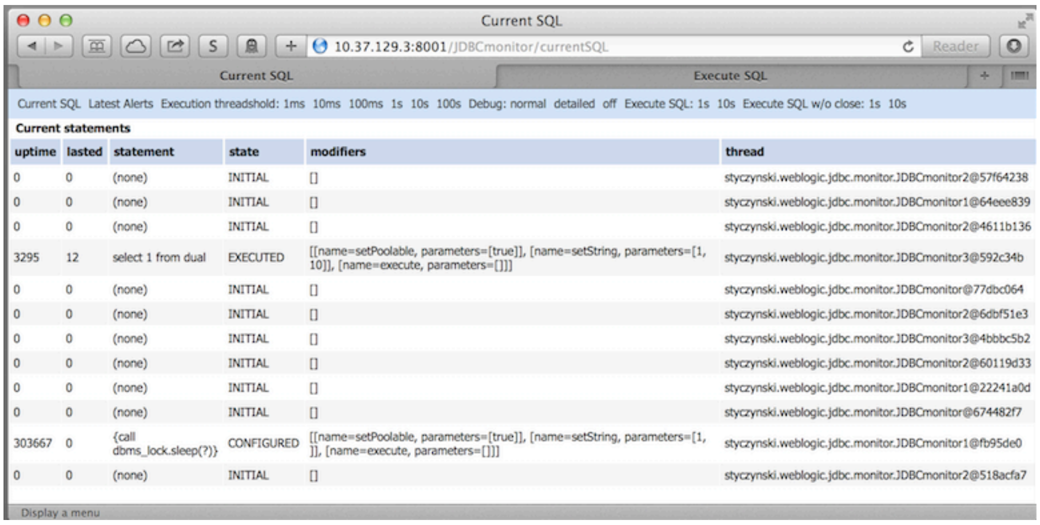
Menu provides ability to execute exemplary by intention slow SQL statement (`{call dbms_lock.sleep(?)}`) on "`jdbc/SOADataSource`". This feature is used to verify proper work of JDBC monitor. Execution of dbs_lock may be parametrized by providing (a) sleep time, (b) datasource, (c) sleep in Java code between JDBC operations, and (d) information to finalize sql interaction w/o close. It's possible to specify own SQL command. Blow URL shows sample calls:

1. sleep for 1s on jdbc/SOADataSource
```
http://10.37.129.3:8001/JDBCmonitor/executesql?param1=1&datasource=jdbc
%2FSOADataSource
```

2. sleep for 10s on jdbc/SOADataSource. Delay by 5 seconds each JDBC operation at client side.
```
http://10.37.129.3:8001/JDBCmonitor/executesql?param1=10&datasource=jdbc
%2FSOADataSource&param2=5
```

3. execution of provided SQL with missing ps.close operation
```
http://10.37.129.3:8001/JDBCmonitor/executesql?sqlCmd=select%201%20from
%20dual&noClose
```

Note that in case of executing SQL statements in improper way, without final jdbc "`close`" operation, JDBC monitor console presents counting up "uptime" with frozen "lasted" timer. The former informs about total time consumed by statement handling, and the former one about time of "`execute`" command. Such situation indicated bug in code.



Current SQL  Latest Alerts  Execution threadshold: 1ms  10ms  100ms  1s  10s  100s  Debug: normal  detailed  off  Execute SQL: 1s  10s  Execute SQL w/o close: 1s  10s

**Current statements**

| uptime | lasted | statement | state | modifiers | thread |
|---|---|---|---|---|---|
| 0 | 0 | (none) | INITIAL | [] | styczynski.weblogic.jdbc.monitor.JDBCmonitor2@57f64238 |
| 0 | 0 | (none) | INITIAL | [] | styczynski.weblogic.jdbc.monitor.JDBCmonitor1@64eee839 |
| 0 | 0 | (none) | INITIAL | [] | styczynski.weblogic.jdbc.monitor.JDBCmonitor2@4611b136 |
| 3295 | 12 | select 1 from dual | EXECUTED | [[name=setPoolable, parameters=[true]], [name=setString, parameters=[1, 10]], [name=execute, parameters=[]]] | styczynski.weblogic.jdbc.monitor.JDBCmonitor3@592c34b |
| 0 | 0 | (none) | INITIAL | [] | styczynski.weblogic.jdbc.monitor.JDBCmonitor@77dbc064 |
| 0 | 0 | (none) | INITIAL | [] | styczynski.weblogic.jdbc.monitor.JDBCmonitor2@6dbf51e3 |
| 0 | 0 | (none) | INITIAL | [] | styczynski.weblogic.jdbc.monitor.JDBCmonitor3@4bbbc5b2 |
| 0 | 0 | (none) | INITIAL | [] | styczynski.weblogic.jdbc.monitor.JDBCmonitor2@60119d33 |
| 0 | 0 | (none) | INITIAL | [] | styczynski.weblogic.jdbc.monitor.JDBCmonitor1@22241a0d |
| 0 | 0 | (none) | INITIAL | [] | styczynski.weblogic.jdbc.monitor.JDBCmonitor@674482f7 |
| 303667 | 0 | {call dbms_lock.sleep(?)} | CONFIGURED | [[name=setPoolable, parameters=[true]], [name=setString, parameters=[1, ]], [name=execute, parameters=[]]] | styczynski.weblogic.jdbc.monitor.JDBCmonitor1@fb95de0 |
| 0 | 0 | (none) | INITIAL | [] | styczynski.weblogic.jdbc.monitor.JDBCmonitor2@518acfa7 |

Another case on above screenshot shows broken execution of `dbms_lock`. In this situation statement was created and abandoned, due to Java exception. Both connections were forcibly closed by WebLogic once thread finished processing request. To simulate such situation execute the following URL:

```
http://10.37.129.3:8001/JDBCmonitor/executesql?param1=XXX
```

To change sensitivity of JDBC interceptor click on menu "`Execution threshold: 1ms 10ms 100ms 1s 10s 100s`". The same may be achieved invoking URL with execution threshold defined by sqlMaxExecutionTime. Providing value of 0 will trace all SQL executions. Note that sensitivity is shared among all JDBCmonitor classes, thus sensitivity change will influence all data sources.

```
http://10.37.129.3:8001/JDBCmonitor/setparameters?sqlMaxExecutionTime=1
```

## Configuration

Current version of JDBC interceptor is shipped w/o persistent configuration. Default SQL execution threshold is set to 1s, and number of in-memory collected alerts to 50. It's possible to change standard configuration via menu provided by console monitor, and a servlet behind it. Note that size of alert array is fixed. Potential change requires modification of `JDBCmonitor.topAlertsToStore` variable and recompilation.

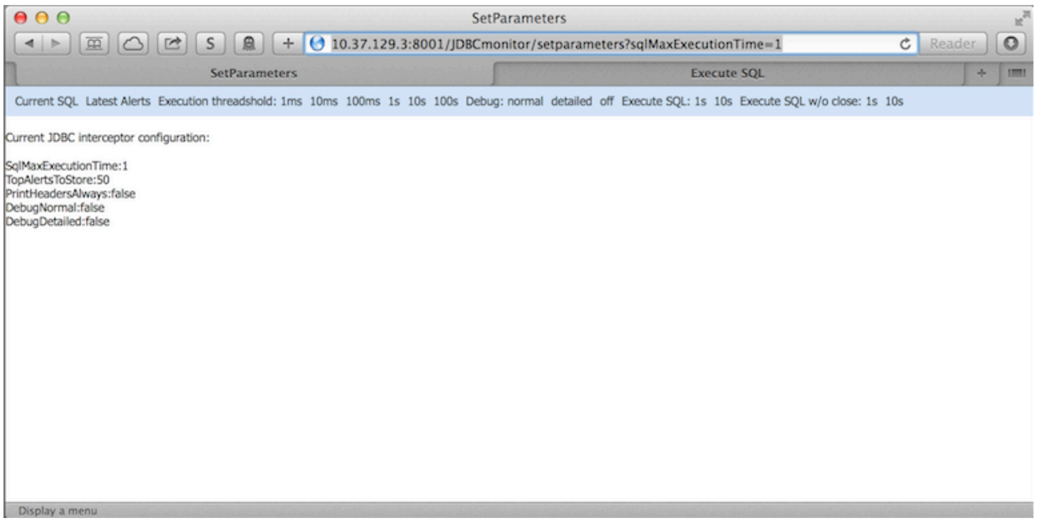
Runtime reconfiguration of monitor sensitivity and debug level  is possible by using provided user interface menu, or internal servlet. Servlet's parameters with its default values are presented below:

|  |  |
|---|---|
| `SqlMaxExecutionTime` | 1000 |
| `PrintHeadersAlways` | false |
| `DebugNormal` | false |
| `DebugDetailed` | false |

Exemplary servlet use:

```
http://10.37.129.3:8001/JDBCmonitor/setparameters?
printHeadersAlways=false&debugNormal=false&debugDetailed=false
```

After invocation of configuration servlet monitor shows page with current value of parameters.


SetParameters
10.37.129.3:8001/JDBCmonitor/setparameters?sqlMaxExecutionTime=1
Reader
SetParameters
Execute SQL
Current SQL  Latest Alerts  Execution threadshold: 1ms  10ms  100ms  1s  10s  100s  Debug: normal  detailed  off  Execute SQL: 1s  10s  Execute SQL w/o close: 1s  10s
Display a menu

```
Current JDBC interceptor configuration:

SqlMaxExecutionTime:1
TopAlertsToStore:50
PrintHeadersAlways:false
DebugNormal:false
DebugDetailed:false
```

## Installation

Installation of JDBC interceptor consist of five steps: (1) placing `wlsJDBCmonitor.jar` file in WLS file space, (2) adding `wlsJDBCmonitor.jar` to WLS system class path, (3) configuring ODL logging subsystem, (4) configuring domain data sources, and (5) optional deployment of web interface (`wlsJDBCmonitorConsole.war`).

*Note that it's mandatory to influence system class path, as interceptor classes are initialized during data source initialization. This operation is down by core WLS modules.*

### Ad.1 Placing jar file in WLS file space

To make use of JDBC interceptor you must place showJDBCcalls.jar in directory available for WebLogic process. Good place is a domain lib directory.

```
DOMAIN_HOME=/home/oracle/Oracle/Middleware/user_projects/domains/
SOA_domain
SSH_USER=oracle
SSH_HOST=10.37.129.3
scp wlsJDBCmonitor.jar $SSH_USER@$SSH_HOST:$WLS_DOMAIN/lib
```

**Ad.2 Add showJDBCcalls.jar to WLS system class path.**
Update domain startup script by adding following shell variable.

```
export POST_CLASSPATH=$DOMAIN_HOME/lib/wlsJDBCmonitor.jar
```

*Note that JDBC interceptor class must be added to system class path. Variable DOMAIN_HOME is available in configured (by setDomainEnv.sh) WLS environment.*

**Ad.3 Configuring ODL logging subsystem**
JDBC interceptor uses ODL logger. Proper configuration, directs SQL execution alerts and potential debug information to different log files. In non configured system, generated information is placed at standard out.

To configure ODL, edit two files:
1. template to be used when configuration of new servers is prepared:
`$ORACLE_DOMAIN_CONFIG_DIR/logging-template.xml`

2. configuration of currently started servers:
`$ORACLE_DOMAIN_CONFIG_DIR/servers/<server_name>/logging.xml`

In both files, put (copy/paste should work) the following:

In section <log_handlers>:
```
<log_handler name='sql-debug-handler'
class='oracle.core.ojdl.logging.ODLHandlerFactory' level='TRACE:32'>
   <property name='path' value='${domain.home}/servers/$
{weblogic.Name}/logs/${weblogic.Name}-sql-debug.log'/>
   <property name='maxFileSize' value='104857600'/>
   <property name='maxLogSize'  value='524288000'/>
   <property name='encoding' value='UTF-8'/>
   <property name='useThreadName' value='true'/>
</log_handler>

<log_handler name='sql-handler'
class='oracle.core.ojdl.logging.ODLHandlerFactory' level='WARNING'>
   <property name='path' value='${domain.home}/servers/$
{weblogic.Name}/logs/${weblogic.Name}-sql.log'/>
   <property name='maxFileSize' value='10485760'/>
   <property name='maxLogSize' value='104857600'/>
   <property name='encoding' value='UTF-8'/>
   <property name='useThreadName' value='true'/>
</log_handler>

<log_handler name='sql1-handler'
class='oracle.core.ojdl.logging.ODLHandlerFactory' level='WARNING'>
   <property name='path' value='${domain.home}/servers/$
{weblogic.Name}/logs/${weblogic.Name}-sql1.log'/>
   <property name='maxFileSize' value='10485760'/>
   <property name='maxLogSize' value='104857600'/>
   <property name='encoding' value='UTF-8'/>
   <property name='useThreadName' value='true'/>
</log_handler>

<log_handler name='sql2-handler'
class='oracle.core.ojdl.logging.ODLHandlerFactory' level='WARNING'>
   <property name='path' value='${domain.home}/servers/$
{weblogic.Name}/logs/${weblogic.Name}-sql2.log'/>
   <property name='maxFileSize' value='10485760'/>
   <property name='maxLogSize' value='104857600'/>
   <property name='encoding' value='UTF-8'/>
   <property name='useThreadName' value='true'/>
</log_handler>

<log_handler name='sql3-handler'
class='oracle.core.ojdl.logging.ODLHandlerFactory' level='WARNING'>
   <property name='path' value='${domain.home}/servers/$
{weblogic.Name}/logs/${weblogic.Name}-sql3.log'/>
   <property name='maxFileSize' value='10485760'/>
   <property name='maxLogSize' value='104857600'/>
   <property name='encoding' value='UTF-8'/>
   <property name='useThreadName' value='true'/>
</log_handler>

<log_handler name='sql4-handler'
class='oracle.core.ojdl.logging.ODLHandlerFactory' level='WARNING'>
   <property name='path' value='${domain.home}/servers/$
{weblogic.Name}/logs/${weblogic.Name}-sql4.log'/>
   <property name='maxFileSize' value='10485760'/>
   <property name='maxLogSize' value='104857600'/>
   <property name='encoding' value='UTF-8'/>
   <property name='useThreadName' value='true'/>
</log_handler>

<log_handler name='sql5-handler'
class='oracle.core.ojdl.logging.ODLHandlerFactory' level='WARNING'>
   <property name='path' value='${domain.home}/servers/$
{weblogic.Name}/logs/${weblogic.Name}-sql5.log'/>
   <property name='maxFileSize' value='10485760'/>
   <property name='maxLogSize' value='104857600'/>
   <property name='encoding' value='UTF-8'/>
   <property name='useThreadName' value='true'/>
</log_handler>
```

In section <loggers>:

```
  <logger name='styczynski.weblogic.jdbc' level='TRACE:32'
useParentHandlers='false'>
   <handler name='sql-debug-handler'/>
  </logger>
  <logger name='styczynski.weblogic.jdbc.monitor.JDBCmonitor'
level='TRACE:32' useParentHandlers='true'>
   <handler name='sql-handler'/>
  </logger>
  <logger name='styczynski.weblogic.jdbc.monitor.JDBCmonitor1'
```

```
level='TRACE:32' useParentHandlers='true'>
   <handler name='sql1-handler'/>
  </logger>
  <logger name='styczynski.weblogic.jdbc.monitor.JDBCmonitor2'
level='TRACE:32' useParentHandlers='true'>
   <handler name='sql2-handler'/>
  </logger>
  <logger name='styczynski.weblogic.jdbc.monitor.JDBCmonitor3'
level='TRACE:32' useParentHandlers='true'>
   <handler name='sql3-handler'/>
  </logger>
  <logger name='styczynski.weblogic.jdbc.monitor.JDBCmonitor4'
level='TRACE:32' useParentHandlers='true'>
   <handler name='sql4-handler'/>
  </logger>
  <logger name='styczynski.weblogic.jdbc.monitor.JDBCmonitor5'
level='TRACE:32' useParentHandlers='true'>
   <handler name='sql5-handler'/>
  </logger>
```

**Ad.4. Configure Data Sources**
Data source configuration may be done using one of three available methods: (1) manual interaction with console, (2) use of WLST, and (3) edit of config.xml. The last option is typically not recommended as may lead to corruption of config.xml. You will probably go for it - please be cerefull.

To configure Data Source manually go to Data Source configuration page: `Diagnostics/Driver Interceptor`, and provide interceptor class name: `styczynski.weblogic.jdbc.monitor.JDBCmonitor` Remember about possibility of adding different interceptor classes for different Data Sources.

WLST snippet is presented below. Replace `<ds_name>` with your data source name.

```
startEdit()
cd('/JDBCSystemResources/<ds_name>/JDBCResource/<ds_name>/
JDBCConnectionPoolParams/<ds_name>')
cmo.setDriverInterceptor('styczynski.weblogic.jdbc.monitor.JDBCmonitor'
)
```

To configure system editing `config.xml` put following line in data source definition file above `</jdbc-connection-pool-params>` tag:

```
<driver-interceptor>styczynski.weblogic.jdbc.monitor.JDBCmonitor</
driver-interceptor>
```

*Note that to benefit from multiple log files, you should use different monitor classes from 6 available: JDBCmonitor, JDBCmonitor1, JDBCmonitor2, ..., JDBCmonitor5.*

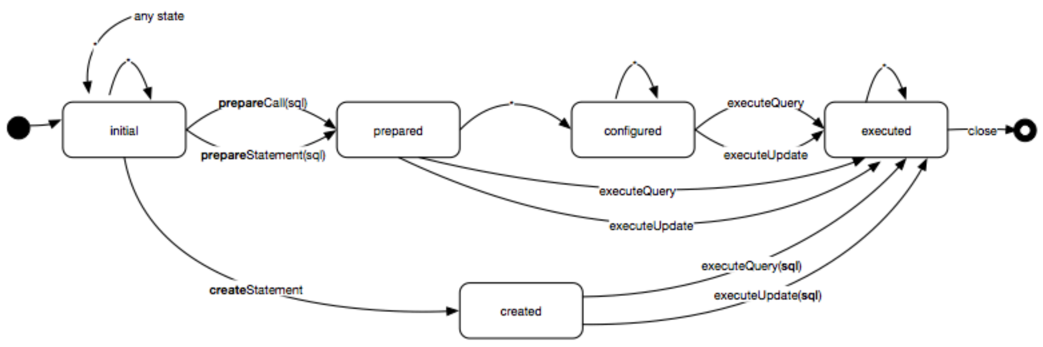
System must be restarted to activate JDBC interceptor.

### Ad.5. Deployment of web interface

Deploy `wlsJDBCmonitorConsole.war` to each host.

## Implementation

WebLogic provides possibility to configure JDBC interceptor - custom code implementing weblogic.jdbc.extensions.DriverInterceptor interface, which is called by each invocation of JDBC layer methods. Provided JDBC monitor is based on this standard WebLogic interface. Internally the code is based on a state machine, what guarantees complete and error free handling of defined flows. Below diagram presents state machine representing SQL execution of both statements and prepared statements.



Picture 1. SQL execution state machine

Interception supports both prepared and normal statements executed on XA and non-XA connections. Each call is traced using state machine logic from its creation, trough configuration, up to statement execution. Executed statements are verified against configured maximum execution time threshold, to be reported in case of too long time spent at database side. Execution alert is reported as a SQL string, execution time, and all setters and method executions used to configure prepared statement.

Using different logger names JDBC monitor directs SQL alerts coming from different Data Sources into separated log files. One more shared file is used to keep all information on debug level.

JDBC monitor is shipped with real time monitoring capabilities based on web interface. This part is a WAR module providing two pages: (1) view on SQL statements currently being executed, and (2) view on latest alerts. Apart of this console may be used to change configuration and trigger long SQL executions.

### Threading and synchronization

Internally all processing is single threaded with minimal synchronization points during normal processing. Whole JDBC interceptor execution is performed on threads executing JDBC calls, what is guaranteed by WebLogic, which physically calls interceptor, before, and after JDBC method invocation. All state information is stored in ThreadLocal variable, making each interceptor independent instance from each other. ThreadLocal is mandatory as WLS uses the same instances of interceptor class to process multiple threads. Due to this, use of local variables (preferred) is not possible, as values are shared between multiple threads.

The only synchronized operation is done during real time view on the system. To make it possible interceptor class exposes static object (`ConcurrentHashMap jdbcGlobalState`), being an interface between interceptor and user interface layer. Global state is updated when new connection is created. The point of synchronized access to `ConcurrentHashMap.put` operation is perceived as a lightweight operation, which does not limit system performance in multithreaded environment. Moreover connection creation should be very rear operation in a running system.

Another potential synchronization point is inside of a web page presenting global state (`CurrentSQL`). Rendering procedure enumerates `jdbcGlobalState` variable exposed by interceptor class. This operation is not synchronized by additional means, using only internal `ConcurrentHashMap` synchronization. Due to lack of extra synchronization, page refresh may sometimes report `ConcurrentModificationException` as backing data structure is continuously updated. This unhanded exception is left in such condition by intention.

Exposing alerts on a user interface is implemented in a synchronization free way. All alerts raised by each interceptor, are stored in an array stored in `ThreadLocal` variable. In case of page rendering, all instances are enumerated to gather arrays' content. Collected data is stored in single local list structure, and sorted by alert's timestamp. Finally page rendering logic gets top N alerts. Note that each interceptor instance stores up to configured number of alerts. Array is a fixed length structure, thus potential change of number of captured/presented alerts requires code recompilation, and WebLogic restart.

Below table shows how alert store array works to implement sliding window over produced alerts.

| **value added** | **counter** | **location in array** | **0** | **1** | **2** | **3** | **4** |
|---|---|---|---|---|---|---|---|
| 1 | **1** | 1 | null | 1 | null | null | null |
| 2 | **2** | 2 | null | 1 | 2 | null | null |
| 3 | **3** | 3 | null | 1 | 2 | 3 | |
| 4 | **4** | 4 | null | 1 | 2 | 3 | 4 |
| 5 | **5** | 0 | 5 | 1 | 2 | 3 | 4 |
| 6 | **1** | 1 | 5 | 6 | 2 | 3 | 4 |
| 7 | **2** | 2 | 5 | 6 | 7 | 3 | 4 |
| 8 | **3** | 3 | 5 | 6 | 7 | 8 | 4 |
| 9 | **4** | 4 | 5 | 6 | 7 | 8 | 9 |
| 10 | **5** | 0 | 10 | 6 | 7 | 8 | 9 |

**Memory considerations**
Global state is indexed by interceptor name concatenated with thread name. Number of instances is limited by number of connections. Global state does not remove objects. It's assumed in normal WLS environment connections are not recreated constantly. Such thing as connection pool shrinking may create potential memory leak situation.

Alerts are stored in a bounded array structure, guaranteeing that only configured number of alerts is stored by each interceptor instance. In case of putting new alert object, oldest one is removed, if structure already reached maximum allowed size.

## Limitations

Current release of the software comes with number of limitations:

- lack of persistent configuration. JDBC interceptor comes with default values, which may be changed by configuration servlet.
- user interface provides no other that viewing features. It's not possible to drill down, sort, filter, etc.
- user interface works only locally - does not provide global view on whole cluster.
- interceptor does not mask sensitive information thus things like passwords will be visible in logs and web interface.

## Known problems

Current release of the software comes with number of known problems:

- real time reporting of SQL commands and alerts is based on unbounded structure, which grows in case of opening new connections. As I do not know how to detect data source shrinking, I'm not able to remove objects associated with removed connections. Workaround of this is to set Data Source Maximum Capacity, and Minimum Capacity to the same value, it will guarantee that connections will not be removed.

## How to get the software

Sources and jar/war files are available on github: `https://github.com/rstyczynski/JDBCinterceptor`

###