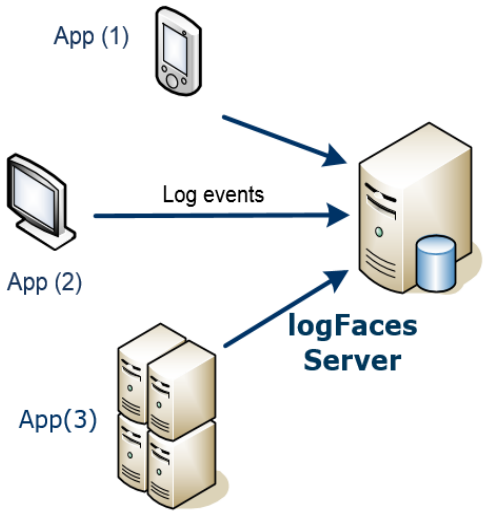
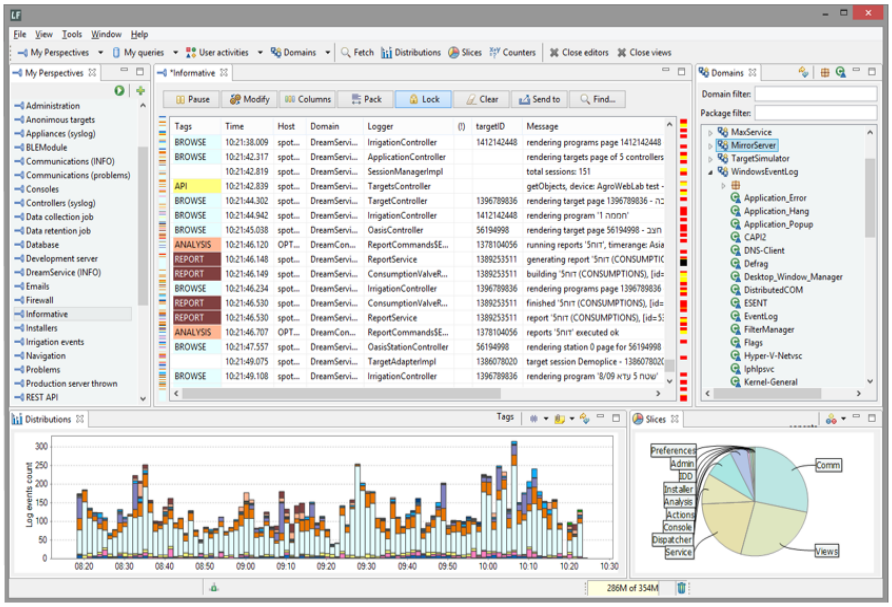
User Manual

v5.3.2

1 Introduction

logFaces is a centralized logging system for applications. It aggregates, analyzes, stores and dispatches log data and events related to the log data.





log4j | log4js | log4net | log4php | log4cxx | logback | NLog | syslog

There are three players in logFaces architecture:

1. your system **producing** log data
2. logFaces server **consuming** log data from large amount of apps and hosts
3. logFaces client **presenting** log data in real-time, historical or analytical form.

logFaces is designed to work with the following sources and network protocols:

- Apache log4xxx API appenders over TCP or UDP
- Syslog RFC5424 and RFC3164 over TCP or UDP
- Graylog GELF over TCP or UDP
- Plain HTTP/s POST requests with JSON or XML payload
- Raw text log files (for offline processing only)

2 Getting started with logFaces Server

This section will guide you through the quick process of installation and configuration. logFaces Server will start after the installation with default settings and trial license for **10 days evaluation**.

2.1 Installing and Uninstalling

On Windows, download and run the installer which will walk you through the process. During installation you will be asked to register and/or run logFaces as Windows service. Linux and Solaris distributions come as `tar.gz` archives, just unzip the archive and you're ready to go. Uninstalling logFaces server is as simple as running the uninstall file located in installation directory. Archived distributions don't require anything, just remove the entire directory.

2.2 Using silent installer options

Running installer exe with `-h` flag will display list of options you can use. Amongst them is `-q` option for running in unattended mode. It is also possible to use a response file from previous installation and re-use it, response files located in `.install4j` directory and named `response.varfile`. For example, to run installation silently you can do "**lfs.windows.win32.x86.exe -q -varfile response.varfile**". This will silently install the server using all options specified in response file.

2.3 Running logFaces Server on Windows

If you selected to run logFaces as Windows service, the service named `LFS` will be registered on your computer automatically and will run every time you start your computer. Use conventional service commands to start/stop the service by doing so in command prompt - `net start lfs` or `net stop lfs.`

You can also run the server as a console application by using `/bin/lfs.bat`.

2.4 Running logFaces Server on Linux/Solaris

- In order to start the server in the terminal do `./bin/lfs console`
- In order to start the server as daemon do `./bin/lfs start`
- In order to stop the server process do `./bin/lfs stop`
- In order to check the server process status do `./bin/lfs status`

2.5 Server directory tree

Once installed and ran for the first time, the server will explode its default configuration and you will find several new folders under installation directory. The table below is a brief summary of folders and their purpose. Normally you won't need all the technical details, but it's a good idea to familiarize yourself with some internals, some of the files and directories are referred to throughout this manual.

Path	Updated with version increments	Description
/.install4j	**yes**	Visible only on Windows, created by installers.
/bin	**binaries only**	Binaries for bootstrapping the server, content is OS dependent
/admin	**yes**	Admin web application and resources
/conf	no	Server configuration files
/db	no	Relevant only for embedded database, contains actual embedded database storage files
/doc	**yes**	Holds release notes and other documents
/legal	no	Holds end user license agreements
/lib	**yes**	Binary distribution libraries and dependencies
/log	no	Holds server internal logs
/overflow	no	Contains overflow files when server overloads
/dropzone	no	Location for manual log data imports
/temp	no	Temporally files, cleared on each server restart

Note that folders which are created and updated by the server (those which are not part of version updates), can be located elsewhere and not necessarily under installation directory. We call these folders **artifacts** and sometimes it's a good idea to keep them separately, for example for the backup purposes or switching from one setup to another. By default all artifacts reside under installation directory, this is the argument for running the server, and you can change it in **/bin/lfs.conf** like this:

```
wrapper.app.parameter.1=HOME
```

`HOME` can be an empty folder; when you run the server for the first time, it will create all default artifacts automatically. Alternatively, you can move other artifacts into `HOME` or point the above parameter to another location with different artifacts in there - this is what we often do during tests. Once again, all this is not required, in most cases the default directory structure is good for nearly all circumstances.

2.6 Integrating log4j-like applications

To work with logFaces, your application needs to be configured by adding several elements to its logging configuration file. Provided that your system is based on log4xxx API, you should be having log4xxx configuration file/s, which usually come as property or XML files.

To allow communication with logFaces Server we add an appender to your logging setup. Sections below will show how to setup such appenders in various situations and using different logging frameworks.

As of this writing we provide our own appenders for log4j and logback frameworks while .Net, PHP and C++ can be freely obtained from Apache downloads and don't require any change to work with logFaces.

Log4j (version 2) as well as logback Java appenders can be obtained from our downloads, place `lfappenders-xxx.jar` under your application classpath. Source code is included in the jar, it's free for everyone to use.

Note that we no longer support log4j v1 because it reached **End Of Life** and is no longer updated. Starting from logFaces version 5.3 we only offer appenders based on log4j2.

Alternatively, in case your project is using **Maven**, you may want to use the following **dependency** in your pom.xml. Just make sure to use the most latest version available in the repository.

```
<dependency>
  <groupId>com.logfaces</groupId>
  <artifactId>lfsappenders</artifactId>
  <version>5.3.0</version>
</dependency>
```

2.6.1 log4j v2.x appenders

We offer a standard log4j2 appender with asynchronous TCP socket connections and fail over mechanism. When application emits log statements, they will be queued and sent to logFaces server by the background thread. In addition to queuing, the appender also knows how to fail over to another host or save logs to a local file. You can specify how often to retry connections, how many times to retry and to what host to switch to when all retries are exhausted. This is an example of log4j2 configuration:

```
<Configuration packages="com.moonlit.logfaces.appenders.log4j2">
   <Appenders>
      <logFaces name="LFST" application="app1" protocol="tcp" remoteHost="10.0.0.110"
                format="json" backup="STDOUT"/>
      <logFaces name="LFSU" application="app1" protocol="udp" remoteHost="10.0.0.111"
                format="xml"/>
      <Console name="STDOUT" target="SYSTEM_OUT">
         PatternLayout pattern="%m%n"/>
      </Console>
   </Appenders>
   <Loggers>
      <Root level="trace">
         <AppenderRef ref="LFST"/>
         <!--
         <AppenderRef ref="LFSU"/>
         -->
      </Root>
   </Loggers>
</Configuration>
```

The appender works over TCP or UDP, see the `protocol` attribute. Both can use either JSON (recommended) or XML `format` serializers. This format should be matched by the receiver format on server side.

Configuration above demonstrates both appenders while TCP appender has a `backup` reference to CONSOLE appender which gets called when server is down. In fact you can use any appenders for the backup by referencing its name with `backup` attribute.

Table below summarizes all configuration properties supported.

Property	Description	Default	Mandatory
application	Identifies application under this name. All logs coming through this appender will be stamped with this name, which can later be used on client.	-	no
remoteHost	Comma separated list of logFaces servers. If more than one host specified, the appender will automatically fail over to the next host when current host becomes unavailable. Switching hosts is done in the loop. If only one host specified, the retries will be done indefinitely with this host.	-	yes
port	Port where logFaces server will accept the connection from this appender.	55200	no
format	Defines data serialization format to use with this server, available options are 'xml' and 'json'. When xml option is specifies, the appender will produce events in log4j 1.x dtd compliant xml format. When json options is specified, the appender will produce events in logFaces proprietary JSON format	xml	no
protocol	Defines which protocol to use with server. Allowed values are 'tcp' or 'udp'.	tcp	no
locationInfo	Specifies whether to include location data, such as class name, method name and line numbers.	false	no
reconnectionDelay	Rate of reconnection retries in milliseconds.	5000	no
nofRetries	How many times to retry before dropping current host and switching to the next one. If only one host specified in remoteHost attribute, the retries will go indefinitely to the same host.	3	no
queueSize	Size of the event queue. The larger the size, the less likely the data will get lost when connection is lost, because events will be re-transmitted to the server when connection recovers. However, queue size affects JVM heap memory, so be considerate.	500	no
offerTimeout	How long to wait (in ms) while offering event to the appender queue. When server is slower than application and queue gets full, the caller has an option to wait before giving up. Queue can typically get full when server is down or when server can't consume log data in the rate of this appender. **WARNING: Use with care as it will slow down the calling thread when queue fills up.**	0	no
backup	Reference to an appender to use when logFaces server is not reachable on any host. This is a backup delegate appender. When specified and server is unreachable with full queue, every log event will be delegated to the referenced appender. When not specified, the events will be discarded. Used only with XML based configuration.	-	no
trustStore	Location of the trust store holding SSL certificates for connecting to the remote server, required if server uses SSL receivers	-	no
trustStorePassword	Password of the trust store file	-	no

Table 2.1: log4j2 appender configuration properties

The appenders are capable to delegate markers as MDC variables named `marker`. For example:

```
Marker ADMIN = MarkerManager.getMarker("SYS-ADMIN");
logger.trace(ADMIN, "this event is for the attention of sysadmin");
```

You then will be able to fetch events marked with 'SYS-ADMIN' property. Make sure to specify MDC mapping on server side so that it contains `marker` property, this is done in server administration Context page.

2.6.1.1 SSL setup

It is also possible to configure TCP appender over SSL transport by adding trust store options as shown below:

```
<Configuration packages="com.moonlit.logfaces.appenders.log4j2">
   <Appenders>
      <logFaces name="LFSSL" application="app1" protocol="tcp" remoteHost="10.0.0.1"
                format="json" backup="STDOUT"/>

            <SSL>
              <TrustStore location="file://C:/my-store" password="mypass"/>
            </SSL>

      </logFaces>
   </Appenders>
   <Loggers>
      <Root level="trace">
         <AppenderRef ref="LFSSL"/>
      </Root>
   </Loggers>
</Configuration>
```

2.6.2 logback appender

If you are using logback framework in your applications, you can easily integrate with logFaces. Below is an example logback configuration :

```
<configuration>
   <contextName>MYAPPLICATION</contextName>
   <appender name="CONSOLE" class="ch.qos.logback.core.ConsoleAppender">
      <layout class="ch.qos.logback.classic.PatternLayout">
         <Pattern>%d{HH:mm:ss.SSS} [%-5level] %logger{36} | %msg%n</Pattern>
      </layout>
   </appender>
  <appender name="LFS" class="com.moonlit.logfaces.appenders.logback.LogfacesAppender">
      <remoteHost>host1,host2</remoteHost>
      <port>55200</port>
      <locationInfo>true</locationInfo>
      <application>${CONTEXT_NAME}</application>
      <reconnectionDelay>1000</reconnectionDelay>
      <offerTimeout>0</offerTimeout>
      <queueSize>200</queueSize>
      <appender-ref ref="CONSOLE" />
      <delegateMarker>true</delegateMarker>
      <format>json</format>
  </appender>
  <root level="trace">
      <appender-ref ref="CONSOLE" />
      <appender-ref ref="LFS" />
  </root>
</configuration>
```

Note how **contextName** can be referenced to specify the application name. When LFS appender will unable to work with server, it will fall back to CONSOLE appender referenced by **appender-ref –** normally you would want to use some rolling file appender instead of just console.

Meaning of the attributes are identical to our log4j appender described above except "**delegateMarker**" option which is specific to logback. If set to true, the appender will automatically copy logback MARKER into event context which will then appear on server as special MDC property named "**marker**". This will allow you to filter and query logs by the markers set in your application. For example, if you do this in your code :

```
Marker ADMIN = MarkerFactory.getMarker("SYS-ADMIN");

logger.trace(ADMIN, "this event is for the attention of sysadmin");
```

Then you will be able easily to fish our all events marked with 'SYS-ADMIN' token. Make sure to specify MDC mapping on server so that it contains "**marker**" property, see Context section under Administration [49] for more details.

Make sure to place **lfsappenders.jar** into the class path of your application, it can be found either in /lib directory of server installation or from our download page. Logback dependency jars must be in the class path as well, make sure you grab them from the authors web site.

2.6.2.1 SSL Setup

It is also possible to use SSL transport by adding trust store options as shown below:

```
...
    <trustStore>c:/my-store-file</trustStore>
    <trustStorePassword>my-password</trustStorePassword>
...
```

2.6.3 log4php appender

log4php comes out of the box with its own socket appender and can be directed to logFaces server, using conventional configuration file. Below is an example of such configuration for both TCP and UDP transports. You will normally need to use one of them:

```
<configuration xmlns="http://logging.apache.org/log4php/">
    <appender name="lfs-tcp" class="LoggerAppenderSocket">
        <param name="remoteHost" value="localhost" />
        <param name="port" value="55200" />
        <param name="timeout" value="2" />
        <layout class="LoggerLayoutXml">
            <param name="locationInfo" value="true" />
            <param name="log4jNamespace" value="true" />
        </layout>
    </appender>
    <appender name="lfs-udp" class="LoggerAppenderSocket">
        <param name="remoteHost" value="udp://localhost" />
        <param name="port" value="55201" />
        <layout class="LoggerLayoutXml">
            <param name="locationInfo" value="true" />
            <param name="log4jNamespace" value="true" />
        </layout>
    </appender>
    <root>
        <level value="TRACE" />
        <appender_ref ref="lfs-tcp" />
        <appender_ref ref="lfs-udp" />
    </root>
</configuration>
```

Note the timeout parameter in lfs-tcp appender, if you choose to use TCP version beware that every log statement will cause new socket connection to open against logFaces server. Being very reliable, the TCP appender may cause delays when your server is down or its network is slow. Having very long timeouts will cause your page visitors to wait if log server is unavailable. UDP appender comes to the rescue, but UDP is inherently unreliable protocol so there is always a chance that some of the datagrams will get lost. The conclusion – choose wisely upon your needs and circumstances.

Below is an example of what normally happens in PHP code. Note how we use MDC '**application**' property to let logFaces server know which application the logs are coming from:

```php
<?php
    // 1. use the path where you unpacked log4php
    include('/development/testphp/php/Logger.php');

    // 2. point to configuration file which must include logFaces appenders
    Logger::configure('/development/testphp/log4php.xml');

    // 3. fetch a logger, any name is OK, best to use names which will
    // be easy to use in logFaces hierarchy like this
    // logFaces will split it into package-like notation for easier traceability
    $log = Logger::getLogger('com.mycompany.myproject.mypage');

    // 4. specify application (domain) name, this is optional.
    // it will allow logFaces server to associate logs with 'application' token
    // if not specified, the logFaces will use the host name of the originating logs
    LoggerMDC::put("application", "my-product");

    // 5. typical logging stuff...
    $log->trace("this is a trace message");
    $log->debug("this is an info message");
    $log->info("this is an info message");
    $log->warn("this is a warning message");
    $log->error("this is an error message");
    $log->fatal("this is a fatal message");
?>
```

2.6.4 log4js-node appenders

The log4js-node project is an awesome Java Script implementation of log4j which can be used in server applications (node.js) as well as client side (browsers). We have contributed UDP and HTTP appenders to this project. There you will find the up to date usage instructions and code samples.

This is how you will want to consume either of the appenders in your work:

```
npm install log4js @log4js-node/logfaces-http
npm install log4js @log4js-node/logfaces-udp
```

Server side application based on node.js can use either HTTP or UDP appenders. Client side application, obviously, can only use HTTP appenders, there is no way to send UDP datagrams from the browser as of this writing.

Logs generated by these appenders will have to find a receiver at logFaces server side, so when you configure the appenders make sure there is a receiver configured on the other end to process the incoming data. Note that both receivers must use the **JSON format** because both appenders generate data in this format.

If you are going to use UDP appender, there should be a log4j UDP receiver on the server side. Make sure that the `host` and `port` appender parameters correspond the receiver configuration.

If you are going to use HTTP appender, there should be a web receiver on the server side. Make sure that the appender `url` configuration matches the one mapped by the receiver.

Note the `application` parameter in both appenders - use it to identify the application (or as we call it - 'domain') on the server side. All the logs generated by appender will automatically show up as part of configured `application` to use later in queries, filters, etc.

2.6.5 log4net appender

If your system is .Net based and using Apache log4net API for logging, you can use its out of the box UdpAppender :

```
<log4net>
    <appender name="logFaces" type="log4net.Appender.UdpAppender">
        <param name="RemoteAddress" value="10.0.0.110" />
        <param name="RemotePort" value="55201" />
        <param name="Encoding" value="UTF-8" />
        <layout type="log4net.Layout.XmlLayoutSchemaLog4j, log4net">
            <locationInfo value="true" />
        </layout>
    </appender>

    <root>
       <level value="ALL" />
       <appender-ref ref="logFaces" />
    </root>
</log4net>
```

As mentioned earlier, logFaces can listen for TCP and/or UDP. In this example, we use UDP appender - make sure that RemotePort attribute in this example corresponds to the one configured in logFaces.

2.6.6 NLog appender

If your system is .Net based and using NLog logging platform, you can use either TCP or UDP or even both of them depending on your needs. Here is the configuration sample :

```
<?xml version="1.0" encoding="utf-8" ?>
<nlog xmlns="http://www.nlog-project.org/schemas/NLog.xsd"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">

    <targets>
      <target name="console" xsi:type="Console" />
      <target name="logfile" xsi:type="File" fileName="file.txt" />
      <target name="lfs-tcp" xsi:type="Network" address="tcp://localhost:55200" />
              layout="${log4jxmlevent:includeMdc=true:appInfo=MYAPP}"
      <target name="lfs-udp" xsi:type="Network" address="udp://localhost:55201" />
              layout="${log4jxmlevent:includeMdc=true:appInfo=MYAPP}"
    </targets>

    <rules>
      <logger name="com.package1.*" minlevel="Error" writeTo="lfs-tcp" />
      <logger name="com.package2.*" minlevel="Trace" writeTo="lfs-udp" />
    </rules>
</nlog>
```

Note how we use '**appInfo**' attribute to identify your application name, and '**includeMDC**' to make sure mapped diagnostic context is transmitted to the server (optional).

2.6.7 log4cxx appender

If your system is based on C++ and using Apache Log4cxx API for logging, you should configure it by adding XMLSocketAppender included in log4cxx API itself. Here is a snippet of configuration example:

```
log4j.appender.stdout=org.apache.log4j.ConsoleAppender
log4j.appender.stdout.layout=org.apache.log4j.PatternLayout
log4j.appender.stdout.layout.ConversionPattern= %-5p %d{HH:mm:ss} %-20c{1} %X{stam} | %m%n

log4j.appender.LFS=org.apache.log4j.net.XMLSocketAppender
log4j.appender.LFS.RemoteHost=10.200.1.110
log4j.appender.LFS.Port=55200

log4j.rootLogger=debug, stdout, LFS, FILE
```

The meaning of the attributes is identical to the previous Java™ example. However, note that XMLSocketAppender doesn't (yet?) provide "Application" and "LocationInfo" attributes. This is not a problem for logFaces – those loggers which don't correspond to any logging domain will be automatically grouped in logFaces under name "**Default Domain**". Unfortunately, until those attributes are supported by the underlying API's, we will have to add some code when initializing the logger in the application. The code snippet is shown below, what we do is simply getting into a root logger, digging out the LFS appender from there and manually set the missing attributes of the layout like this:

```
// this is a workaround for XMLSocketAppender to allow routing of properties
// over the network, we manually setup the layout
LoggerPtr root   = Logger::getRootLogger();
AppenderPtr app = root->getAppender(LOG4CXX_STR("LFS"));
if(app != NULL){
   LayoutPtr layout = app->getLayout();
   if(layout != NULL){
      layout->setOption(LOG4CXX_STR("locationinfo"), LOG4CXX_STR("true"));
      layout->setOption(LOG4CXX_STR("properties"), LOG4CXX_STR("true"));
      MDC::put("application", "WSC");
   }
}
```

In any case, those missing attributes are not a show stoppers, your application can still work with logFaces out of the box with those limitations.

IMPORTANT:

> The MDC (message diagnostic context) works only in the context of the current thread. In case you have several threads in your application you should add **MDC::put("application", "xxx")** call in the beginning of every thread. Otherwise, the log statements coming from those threads will be orphaned and server will automatically put them under "Default Domain" which might be a bit confusing. Future versions of logFaces will include proper appender to avoid those workarounds.

2.7 Understanding data model

Log data is represented in logFaces with two entities – **repository** and **log event**. You will also find corresponding tables in RDBMS schema or as MongoDB collections, depending which database you are using.

Repository holds a description of entire log storage, meta-data would be a good term to describe it. These are names of hosts, applications, loggers and exceptions ever recorded by the log server. Repository becomes more or less static once the system fills up with data and it's used mostly as a helper for getting lists of things. For example, when you need to fill in a query involving a host name, the repository will provide a list of all known host names.

Log events are represented in logFaces as a flat collection of fixed attributes. The attribute names are summarized in the table below:

Attribute name	Short name	Type	Description
loggerTimeStamp	**t**	Long	Time stamp as specified by the source or server
sequenceNumber	**q**	Long	Sequence number, each event produced by logFaces has running sequence number
loggerLevel	**p**	Integer	Severity of event expressed in term of log4j levels
domainName	**a**	String	Name of the domain (or application) originating the event
hostName	**h**	String	Name of the host originating the event
loggerName	**g**	String	Name of the logger (class, module, etc) originating the event
threadName	**r**	String	Name of the thread originating the event
message	**m**	String	Message content
ndc	**n**	String	Network diagnostic context
thrown	**w**	Boolean	Indication whether the event is a thrown exception
throwableInfo	**i**	String	Stack trace of thrown exceptions
locFileName	**f**	String	File name (of the source code location originating the event)
locClassName	**c**	String	Class name (of the source code location originating the event)
locMethodName	**e**	String	Method name (of the source code location originating the event)
locLineNumber	**l**	String	Line number (of the source code location originating the event)
properties	**p_XXX**	String	MDC (Mapped Diagnostic Context) properties where XXX is a property name. There could be an arbitrary number of named properties which get specified manually when setting up the server. See 'Context' section in administration for more details.

This data model is further realized in RDBMS schema for each database type supported as well as main data collection in MongoDB. In case of MongoDB the attribute names are reduced to a minimum to save the storage space as shown in “Short name” column. These short names are also used by TCP appenders when using JSON format as well as web receivers.

2.8 Working with regular expressions

Regular expressions are used quite extensively in logFaces. You will meet them to process things like unstructured syslog content or parse log files in arbitrary text format.

Using regular expressions effectively requires some practice and could be frustrating in the beginning. Working with complex expressions to crunch large amounts of unstructured text may easily become a daunting task if not applied systematically. So here we come with a solution to make your life easier even if you are a beginner with regular expressions.

The idea was originally borrowed from logstash project (all the credits go to this community!) and adopted for our needs with some insignificant polish.

logFaces server is shipped with regular expression **patterns library** – a text file you will find under /conf directory on your server. This library initially contains some commonly used expressions and you can extend and modify them as you go.

Patterns library can be shared amongst clients – they have a built-in capability for parsing log files, see 'Viewing raw log files' section for more details.

Consider some simple pattern examples:

```
WORD    \b\w+\b
HOUR    (?:2[0123]|[01]?[0-9])
MINUTE  (?:[0-5][0-9])
SECOND  (?:(?:[0-5][0-9]|60)(?:[:.,][0-9]+)?)
TIME    (?!<[0-9])%{HOUR}:%{MINUTE}(?::%{SECOND})(?![0-9])
```

Each line in the library contains exactly one pattern which has a name (left part) and the expression (right part). Name and expression must be separated by space character.

Note how any expression can reference other patterns by name using **%{xxx}** notation. Before the expression is used, server will unfold all the references to produce plain standard regular expression. In practice this could be very large piece of text, quite often non-readable by humans.

Using this technique makes it very easy to prepare fairly complex expressions to parse unstructured log data in almost any format.

And now is the key part – using **named groups** to extract text phrases. All modern regular expression engines support capturing groups, which are numbered from left to right. This feature is hidden in the format we use. Consider an expression like this.

%(WORD:hostName) %(WORD:loggerName)

Note how **WORD** pattern is used along with **hostName** and **loggerName** which are fixed attributes in logFaces, or so called named groups in regex language. The example above will take a two word sentence and extract first word into hostName variable, and the second word into loggerName variable.

Here is a real world example, the expression below will parse Apache access log and extract the data into logFaces attributes (it should be one line!):

```
APACHELOG %{IPORHOST:peer} %{USER} %{USER} \[%{HTTPDATE:loggerTimeStamp}\] "(?:%
{WORD:verb} %{NOTSPACE:request}(?: HTTP/%{NUMBER:httpversion})?|%{DATA:rawrequest})" %
{NUMBER:response} (?:%{NUMBER:bytes}|-) %{QS:referrer} %{QS:agent}
```

Note that attributes named '*peer*', '*verb*', '*request*', '*httpversion*', '*rawrequest*', '*response*', '*bytes*', '*referrer*', '*agent*' are not part of the logFaces schema but they still can be extracted and used. This is when MDC (mapped diagnostic context) comes into play.

MDC allows you to extend the fixed attributes with more attributes and be able to index them. So, if you map the above attributes as MDC in your server context, you will be able to use those attributes in tracing, lockups and other logFaces features. MDC management is described in server administration Context section.

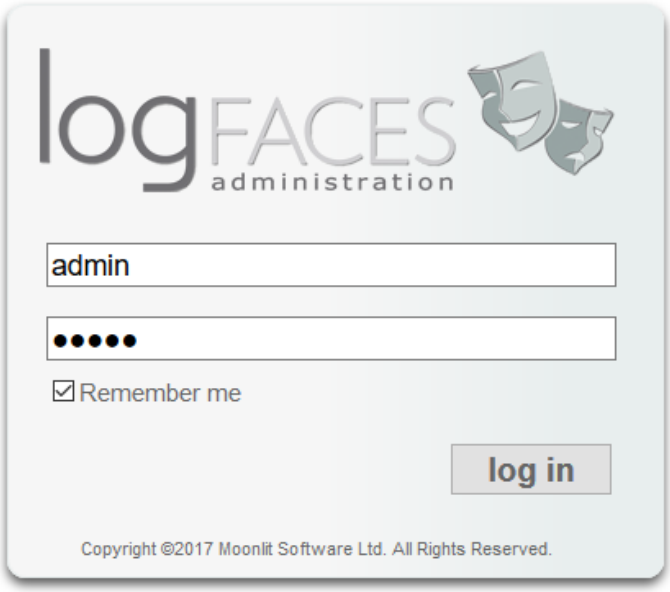
2.9 Server administration

logFaces server is administered through a web interface which you will find for the first time at:

http://your-host:8050

It would also be a good idea to bookmark this URL in your browser – you will visit these pages quite often during initial setup and acquaintance with the product.

Normally logFaces servers are shared amongst many users within organization, this is why the access to its administration is restricted to someone with special credentials. The user name and password will be required to get in:



By default, the user name and password are stored on server disk in obfuscated form. Default user name and password are both '**admin**' after installation. Do make sure to change them eventually.

Instead of local authentication it is also possible to delegate user name and passwords to your own LDAP directory and let it verify the credentials on behalf of logFaces.

By default the server gets accessed over plain HTTP. But it is possible to switch server entirely to be used over SSL.

2.9.1 Front end connectivity

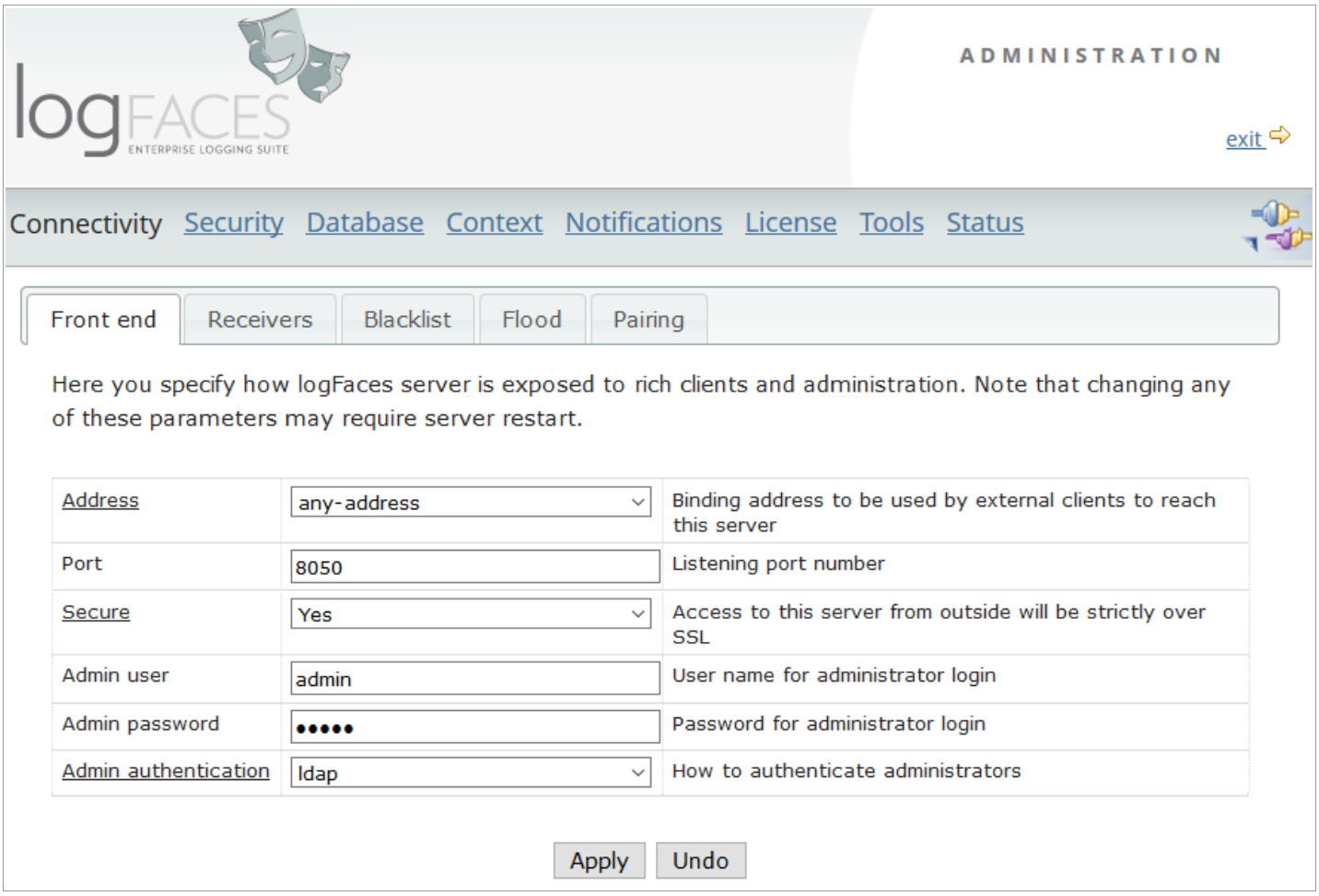


Figure 2.9.1: Front end connectivity

Address	If server is installed on a computer with several network cards, you can bind server sockets to a particular address or a host name. This is a good idea if you need physical separation of the transport. When '**any-address**' is specified, the server will pick the default binding, this will allow access to server by any address it supports. Otherwise, the access will be strictly by the address you select.
Port	Listening port number (default is 8050)
Secure	Access to this server can be secured over SSL. This option is available only after setting up a key store in the Security/Network section.
Admin user	User name for accessing administration
Admin password	Password for administrator access. Stored in obfuscated form on local disk unless used with LDAP.
Admin authentication mode	In **local** mode, the authentication of admin users will be performed against locally stored credentials. This is the default authentication mode. In **ldap** mode, the authentication of admin user will be delegated to your LDAP server. Make sure to specify the admin user name, this is what will be sent to your LDAP. The password is not required.

2.9.2 Receivers

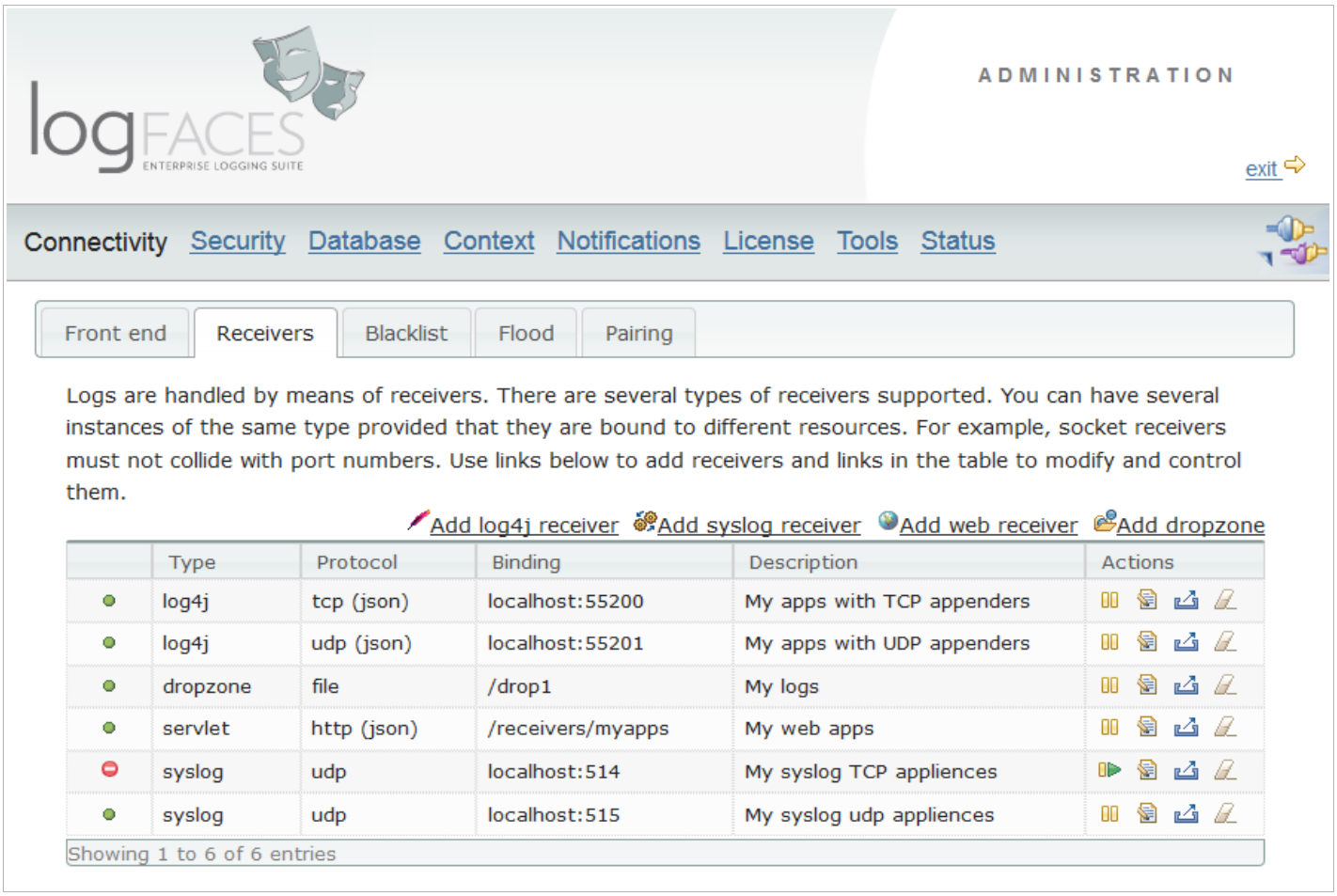


Figure 2.9.2: Receivers

Receivers are receiving log data from outside world into logFaces server. There are four types of receivers currently available:

1. Log4j receivers - XML or JSON over TCP, SSL or UDP. These types of receivers will work with anything based on Apache Logging Framework – log4j, log4net, log4php, logback.
2. Syslog RFC5424 and RFC3164 over TCP, SSL or UDP
3. Web receivers over HTTP
4. Drop zones – offline processing of raw text

It is possible to have as many instances of the same receiver type provided that there is no clash of resources. Using the links on the right it is possible to pause, resume, modify, remove and test each individual receiver. Testing is very convenient because it verifies how the receiver will work in real life under conditions you specify. Icons on the left indicate the state of the receiver – active or not active.

2.9.2.1 Socket receivers

Socket receivers can process log data in various formats generated by remote applications and delivered over simple raw sockets. There is a whole array of frameworks offering ready to use appenders with log4j **XML** formats, there are variants available for Java, Python, Perl, PHP, .Net.

If your application is Java based and you are using log4j or log4j2, it is highly recommend to use our asynchronous socket appenders with **JSON** format, it has very compact footprint which is great for systems with high volumes of log data.

It is also possible to use socket receivers with **GELF** format which is popular amongst tools such as Graylog, Logstash, Fluentd, etc. The **G**raylog **E**xtended **L**og **F**ormat is a log format that avoids the shortcomings and limitations of classic plain syslog.

Typical form for adding new or modifying existing log4j receiver is shown below:

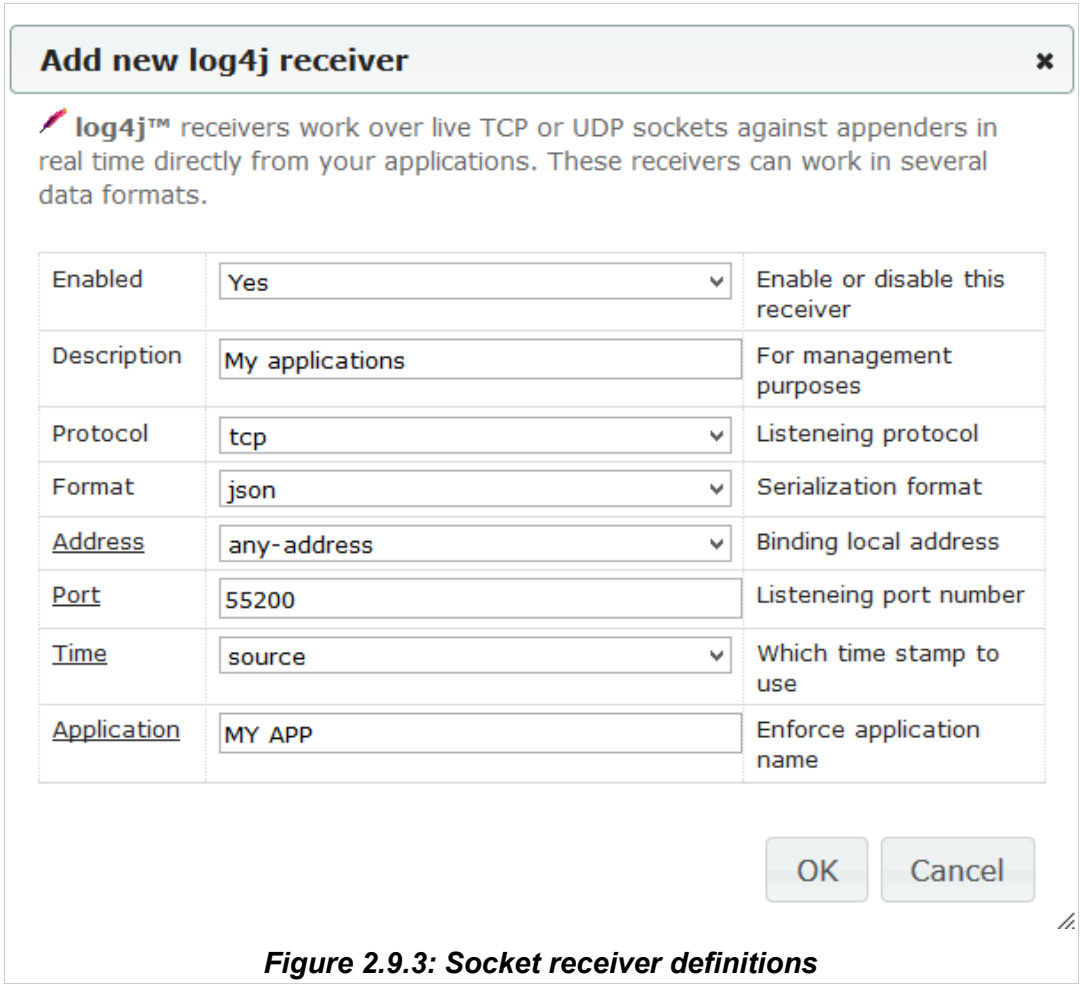


Figure 2.9.3: Socket receiver definitions

Enabled	Makes receiver active or non-active. Not active receivers will stay in the list but will not work.
Description	Friendly description or title of receiver, for management purposes only
Protocol	Type of protocol to use – TCP, SSL, UDP
Format	Serialization format understood by this receiver. Must correspond to the transmitted data format. Available options are **xml**, **json, gelf**. When **xml** option is specifies, the appender will produce events in log4j 1.x dtd compliant xml format. When **json** options is specified, the appender will produce events in logFaces proprietary JSON format. When **gelf** option is specified the data source is expected to produce events according to Graylog specification.
Address	If server is installed on a computer with several network cards, you can bind server sockets to a particular address or a host name. This is a good idea if you need physical separation of the transport. When '**any-address**' is specified, the server will pick the default binding, this will allow access to server by any address it supports. Otherwise, the access will be strictly by the address you select.
Port	Listening port, make sure it's available before enabling the receiver
Time	Which time stamp to use for received log events – the one originating from the source, or the server local time. Use this option when there are many sources and there is no time server to unify the times across the applications.

2.9.2.2 Syslog receivers

logFaces server has its own embedded syslog server which can be used to consume syslog messages from any source using TCP, SSL or UDP connections compliant with RFC5424 or RFC3164. Current implementation of syslog server is designed to works as a **collector**, or final destination of syslog events, meaning that it's not designed to relay (or forward) received events to other syslog servers.

Consuming syslog data is no different from consuming log data from other socket appenders. All you need to do is to define syslog receivers and setup their parameters properly. Because syslog is a very loose specification and incredibly fragmented amongst magnitude of devices using it, there are many ways of how to extract the important information and map it to real data which is later used by logFaces.

Using regular expressions you will be able to do most of the mappings. Everything about working with syslog revolves around settings up and testing syslog receivers:

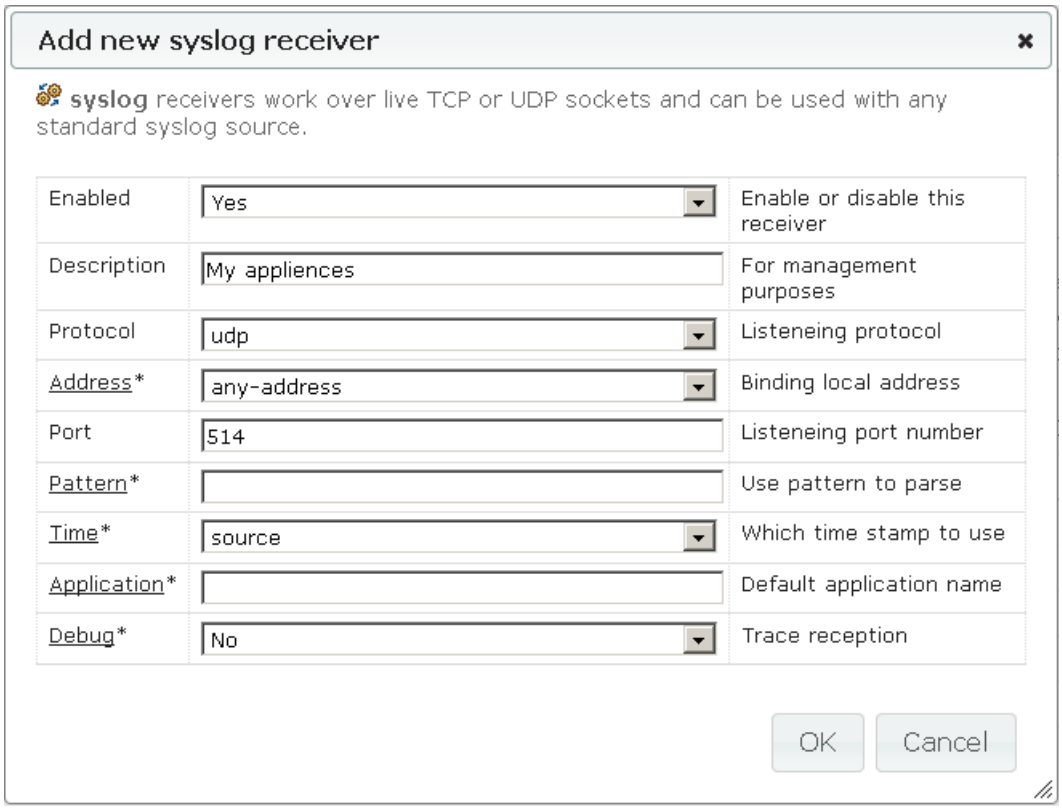


Figure 2.9.4: Adding syslog receivers

Enabled	Makes receiver active or non-active. Not active receivers will stay in the list but will not work.
Description	Friendly description or title of receiver, for management purposes only
Protocol	Type of protocol to use – TCP, SSL, UDP
Address	If server is installed on a computer with several network cards, you can bind server sockets to a particular address or a host name. This is a good idea if you need physical separation of the transport. When '**any-address**' is specified, the server will pick the default binding, this will allow access to server by any address it supports. Otherwise, the access will be strictly by the address you select.
Port	Listening port, make sure it's available before enabling the receiver
Pattern	Provide your own interpretation of syslog message structure by setting a regular expression pattern. Use patterns library to build and test patterns. If pattern is not specified, logFaces will try its best to structure the incoming data. This is not always possible and very much depends on the log data source. See examples below.
Time	Which time stamp to use for received log events – the one originating from the source, or the server local time. Use this option when there are many sources and there is no time server to unify the times across the applications.
Application	Leave blank if name of the application is properly transmitted by syslog source. Otherwise this name will be used as default substitute. If none specified and can't be resolved from the message, logFaces will use 'appliances' as a default. It is a good practice to always have some meaningful application name – it will help clients to display structure of logs and do queries.
Debug	Enable this option if you want to see what exactly your sources transmit over the wire. logFaces server will log incoming traffic in its internal log file. This option should help you to pick a best pattern and structure the data for indexing. When something gets recorded into internal log, simply pick up the raw text and use pattern debugger for creating matching patterns. Or use 'test' link to inject the message directly into receiver. Make sure to enable verbose logging to see the traces. Do not leave verbose logging for production use for better performance.

To understand the usage of regular expression patterns, lets demonstrate a real world example. Below is a syslog message transmitted from one of the popular bridges sending a syslog message from Windows Event Log subsystem:

```
<29>Aug 24 10:57:06 SERVER-1 Security-Auditing: 4624: An account logged on
```

If no pattern is used, the receiver will produce a log event with severity **INFO**, time **Aug 24 10:57:06**, host **SERVER-1** and message text "**Security-Auditing: 4624: An account logged on**". The rest of the attributes will be ignored.

In some situation this is good enough.

But things can be improved greatly if you tell the parser how to extract the number **4624** which is ID of windows event in this example and can be indexed.

You may also want to have a "**Security-Auditing**" token to be used as name of the logger or an application name, or any other mapped property. So, consider a pattern like this instead:

```
%{HOSTNAME:hostName} %{NOTSPACE:loggerName}\: %{NOTSPACE:eventID}\: %{GREEDYDATA:message}
```

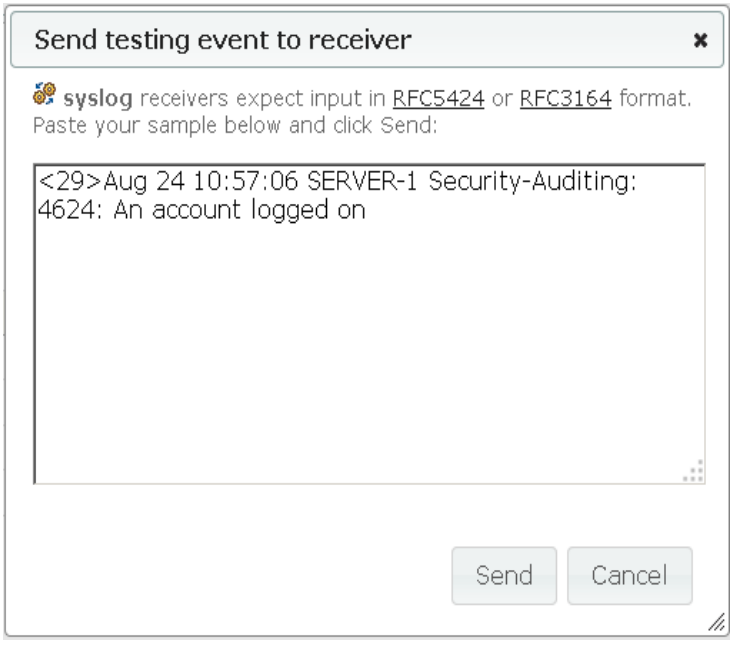
Note how this regular expression is based on predefined **patterns** wrapped into %{xxx} tokens and doesn't seem to resemble what we used to call regular expression. In fact it is a valid regular expression, it's just that it's written in a very concise and structured form. Make sure to familiarize yourself with regular expression patterns before trying to understand this format. It's very simple and powerful way for dealing with really massive and complex regular expressions which may span pages of text.

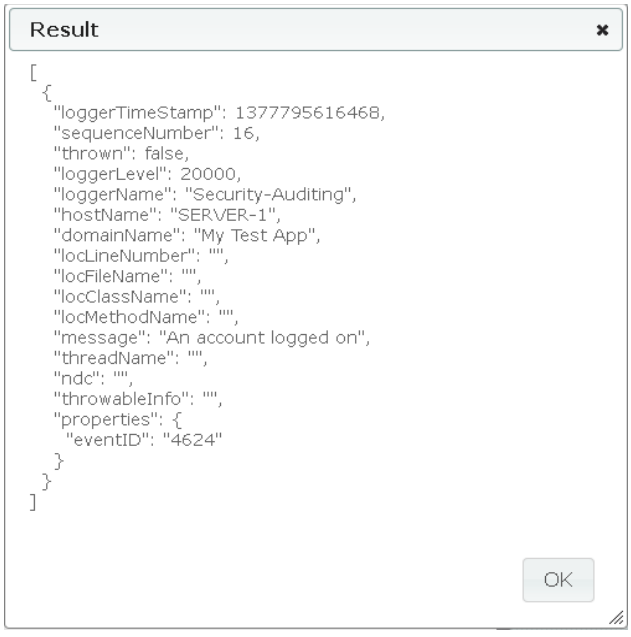
Applying this pattern will produce an event object with two additional pieces of information we couldn't get before. It's the **loggerName** matching to "**Security-Auditing**" substring in this example. Logger name is a logFaces attribute, so it can be indexed, used in queries, displayed in separate columns, filtered by, etc.

Also there is an **eventID** which is not part of logFaces attributes but is MDC (mapped diagnostic context) which you can freely use for extending the fixed set of attributes used by logFaces. In this example, the parser will extract "**4624**" and set a property named **eventID** before sending the event through the process chain. When you map this property name as an MDC context, you will be able to index those properties, query their values, and so on.

The message body in this example is also altered to "**An account logged on**" since we extracted the other pieces from it.

Finally, make sure to test your receivers by clicking on icon on the receivers list. Paste some actual syslog formatted message and send it to server, the response will be a JSON-like structure representing the logFaces event. Keep tuning receiver options until test result is just right.



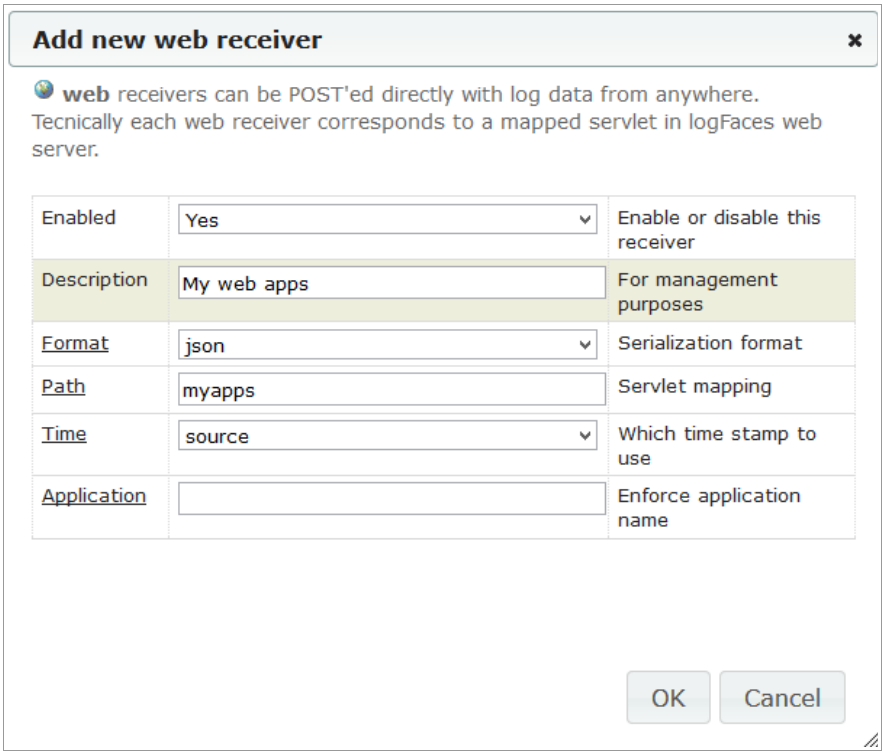


2.9.2.3 Web receivers

Web receivers process HTTP POST requests bound to a specified URL. Server implementation is targeted for `cross domain` requests meaning that sender can be located anywhere. Each request is expected to carry one or several log events in a compact JSON format described here. All web receivers are mapped to logFaces server base URL which looks like this:

```
http://your-host:port/receivers
```

Each individual web receiver URL is appended to the one above, this is how web receiver is configured in admin:



The receiver above will be mapped to

```
http://your-host:port/receivers/myapps
```

It will attempt to process any incoming requests, convert them into log events and pass through the regular chain of handlers like database persistence, real-time monitoring, triggers, etc.

2.9.2.4 Drop zones

logFaces server is capable of importing raw text files, converting them into data, storing them into database and let you query them. There are no special requirements for the format of the raw files except that they should be parse-able by means of regular expressions. If you can write a regular expression to parse your logs, logFaces will parse and index them for you.

Importing is done by copying files into a special folder which logFaces monitors – **drop zone**. Arrived files are then processed one by one by applying parsing rules defined for this drop zone.

Drop zones prevent duplications - when identical content is dropped for processing twice, it will be ignored for the second time. Drop zones will detect that new data was appended to a file previously processed – this is done by tracking the check sums of the top and bottom parts of each processed file. Dropped files are deleted from the drop zone folder once processed successfully or failed. Logs which failed are stored in separate folder for your inspection, you may want to adjust some regular expressions and re-drop the content again until it gets through.

Drop zones are defined and set up in the same way as other receivers:

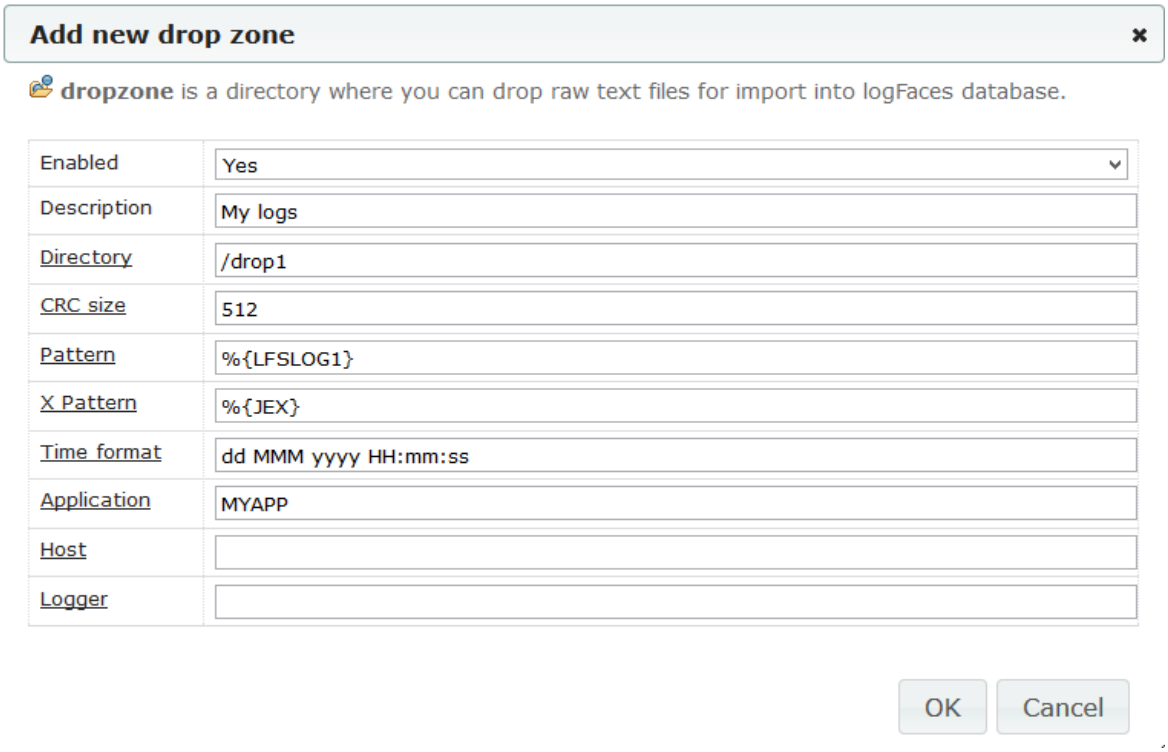


Figure 2.9.5: Adding drop zones

Enabled	Makes receiver active or non-active. Not active receivers will stay in the list but will not work.
Description	Friendly description or title of drop zone, for management purposes only
Directory	Drop zone is a monitored location where you can drop raw text files for import. Those locations are relative to your server installation **/dropzone** directory. Files can be in any format provided that they can be parsed by means of regular expressions. Files will be permanently deleted from this location as soon as server attempts to process them. When a file gets processed partially, the server will create a special file containing lines which failed parsing. It will be located under **/unprocessed** directory in this drop zone location. You will then be able to examine unprocessed entries, adjust the patterns and re-drop.
CRC size	Server will look at the head and tail of each dropped file (first and last bytes), calculate their CRC check sums and keeps the track of every record. This parameter defines the **length** of the head and tail in bytes to be used for calculating the CRC. Must be positive non-zero value. CRC is used for dealing with duplicated and appended content. Looking into a head/tail CRC server will decide whether the content is new, partially processed or already processed in the past. For example, if several lines were added to the file since its last import, server will detect and import only those lines which were added Server keeps small local database where all CRC's are recorded. Every processed file CRC's gets registered in this database. If you want to clean up this database, remove directory named **/dzcache** under your server installation. By default the size of this database is 10000 and specified in **/conf/environment.properties** file. When this size is reached the database will start rotating by removing older records while inserting new.
Pattern	This is a regular expression pattern to match the text in dropped files and extract log data. It may be a conventional regular expression for matching event attributes, or a combination of pre built patterns. Use patterns library to build and test complex regular expressions. See examples below.
X Pattern	If you are expecting to process exception stack traces which are normally multi-line fragments of formatted text, consider specifying this pattern to extract the structure. Typical Java-like stack traces can be matched with pre-built pattern **%{JEX}**. If your stack traces look different, consider adding it to pattern library and re-use. See example below.
Time format	Here you specify the expected date time format of the logs to process in this drop zone. Look here for supported formats. logFaces will use this format to covert parsed text into numeric epoch time (number of milliseconds past since 1970). If no time format specified, the server will import all the logs with current server time, incrementing each event by one millisecond – this is generally not advisable option.
Application	Application name to be used if not present in the original logs. If not specified logFaces will use '**default**' word as a substitute.
Host	Host name to be used if not present in the original logs. If not specified logFaces will use '**default**' word as a substitute. In this context, under the host name we normally expect the host that originally produced the log event.
Logger	Logger name to be used if not present in the original logs. If not specified logFaces will use '**default**' word as a substitute. In this context under the logger name we normally expect a class, module or component produced the log event.

Make sure to test drop zones by sending some fragments of the raw files you are expecting to use directly from administration page. For example, pasting the following lines:

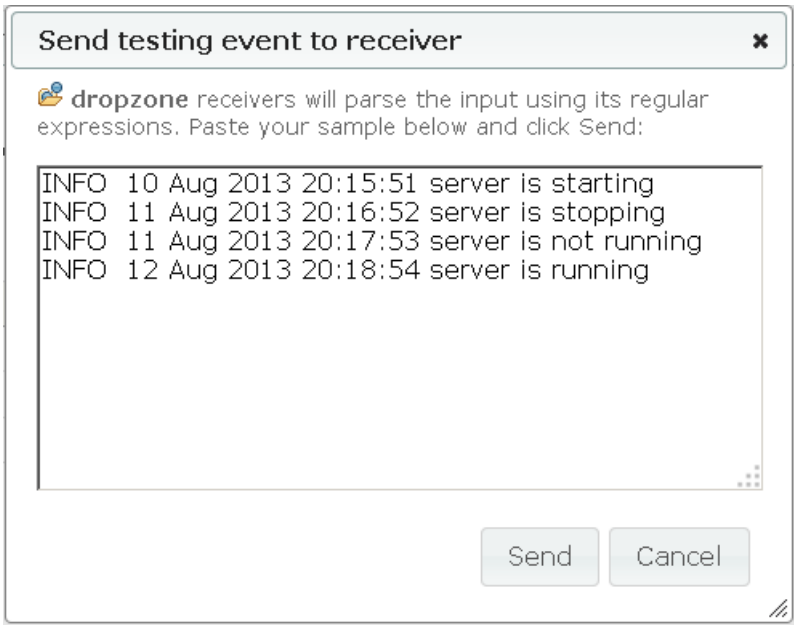


Figure 2.9.6: Testing drop zones

Server will reply with parsed structured data or an error. This way you can tune all parameters until everything works as expected and before placing the drop zone into a real work stream,

2.9.3 Black list

Often it is desirable to completely block some of the log traffic. This is exactly what **black list** does in logFaces. When specified, the black list criteria will discard matching log events before they get any attention by the server components.

For example, the criteria below will discard events with severity level below INFO or when host name is an IP address.

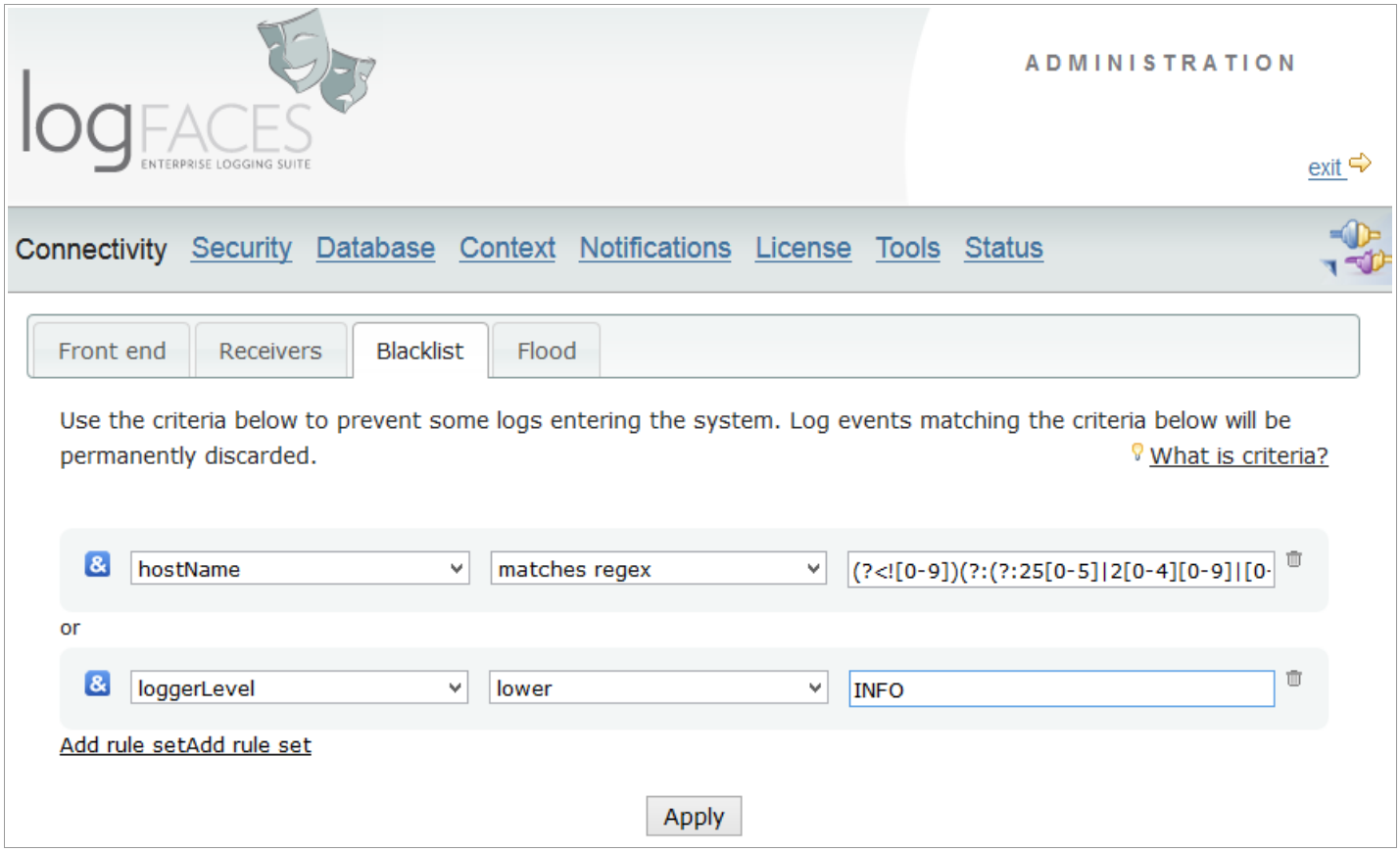


Figure 2.9.7: Black list example

2.9.4 Flood management

Sometimes applications would send bursts of repetitive logs or get into long lasting loops sending redundant useless logs, most of which can be safely ignored most of the time. Typically those are bugs on the application side or poorly implemented logging strategy. To reduce amount of this kind of traffic, logFaces offers flood detectors. They will keep your server healthier by reducing amount of useless inflow and make it easier finding useful data when there is less junk around.

When flood detectors specified, the incoming logs will be probed by each detector in turn and discarded when necessary. When at least one flood detector believes that log event looks like flood, this event won't enter the server processing and will be discarded early. So what is flood?

Flood is what we say it is by setting up detector rules - the minimum number (threshold) of events matching certain criteria detected within specified time window. By changing threshold and time window we can control the incoming traffic while matching criteria is used to focus on the content.

In the example below, the detector named “TRACE reduction” ensures that there are no more than 10 trace level events within 10 seconds window at all times. Whenever this rule is broken, the detector discards the incoming logs.

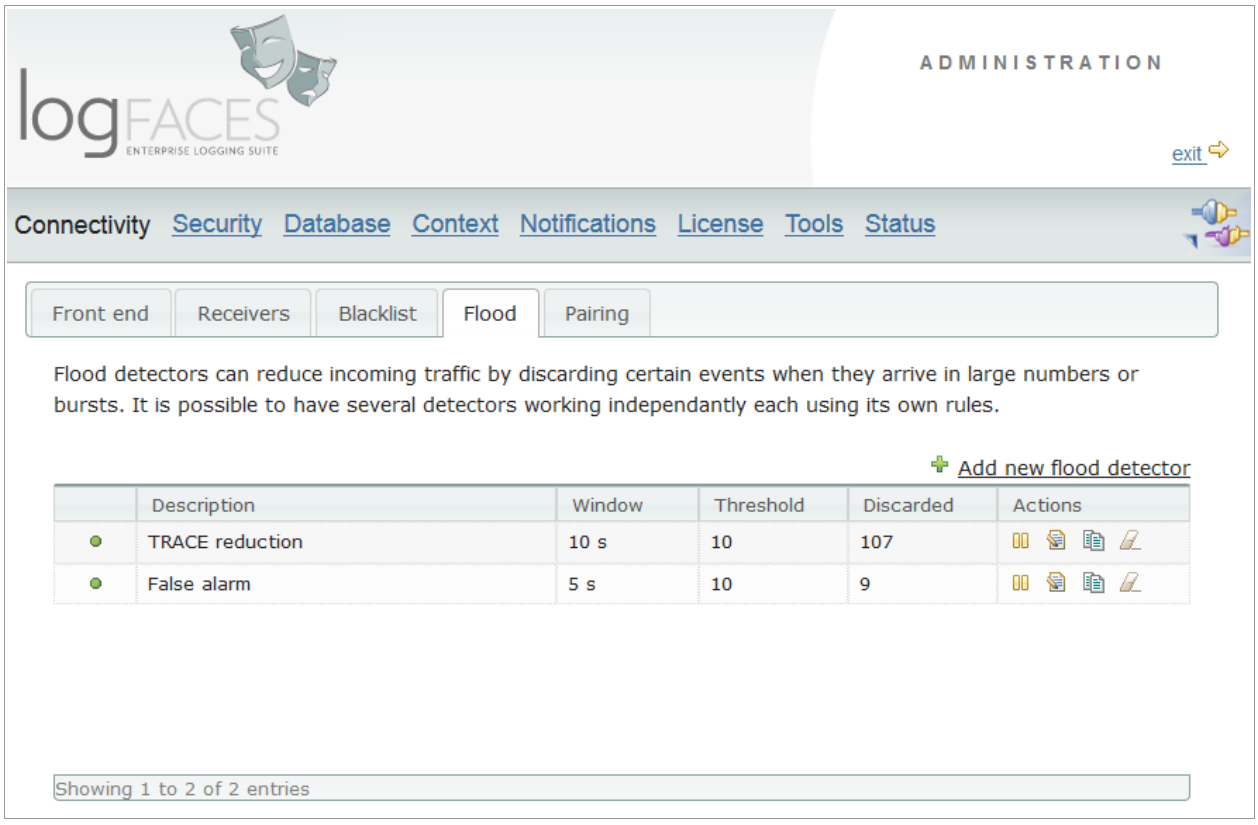


Figure 2.9.8: Flood detectors

2.9.5 Pairing

Pairing is a mechanism for providing single access point to the data when several logFaces nodes are deployed as a single system. Multiple nodes are used for two purposes: a) for splitting the inflow when working with large amount of applications and b) for providing fail over. In either case, you deploy several nodes which do actual data processing against the apps (**back end**) and another node for user access only (**front end**). This way, the users don't have to know which node to connect to in order to receive the logs, they always connect to a single **front end** node which is paired with **back end** nodes by specifying their connection end points – host, port and SSL option. It is possible to automatically deploy front end configuration to the back end nodes.

The example below illustrates how **this** node (front end) is paired with 2 back end nodes at 10.0.0.110 and 192.168.24.130. Since all three nodes are expected to use the same database, all queries and reports will be served from the front end node. Real time notifications like triggers and client perspectives are consolidated by all participating nodes. Note that back end administration is very limited because most of the tasks are done by the front node.

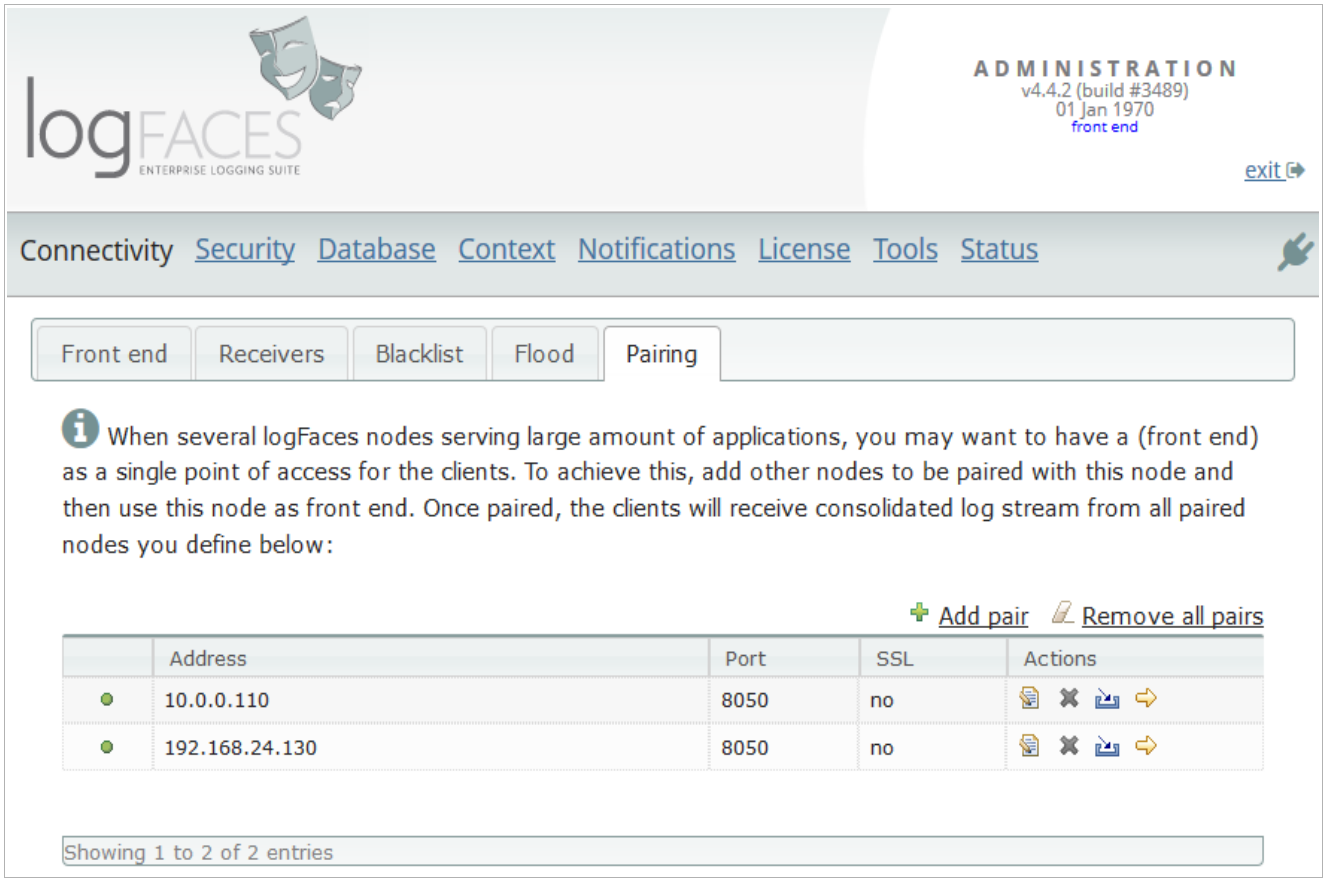


Figure 2.9.9: Pairing example

2.9.6 Using SSL

It is possible to secure socket layers at all fronts - with clients, browsers, applications, databases and even LDAP access.

logFaces server acts as server and client in the context of SSL simultaneously. It acts as a server when it accepts connections from clients and applications, and it also acts as a client to other services it uses like your database, LDAP service, etc. To secure these network layers, logFaces must be able to verify (and present) certificates to its peers. This is done by introducing SSL certificates into logFaces server JVM. Note that you can apply full or partial network security. For example, you may secure the front end or the application layers, but leave connection to database non-secure, it all depends on the setup you are trying to achieve.

Configuration is done by setting up a **Key store** for serving an incoming connections and **Trust store** for serving an outgoing connections. These stores are files located in /conf directory on your server and referenced by environment configuration which also keeps their passwords (in obfuscated form).

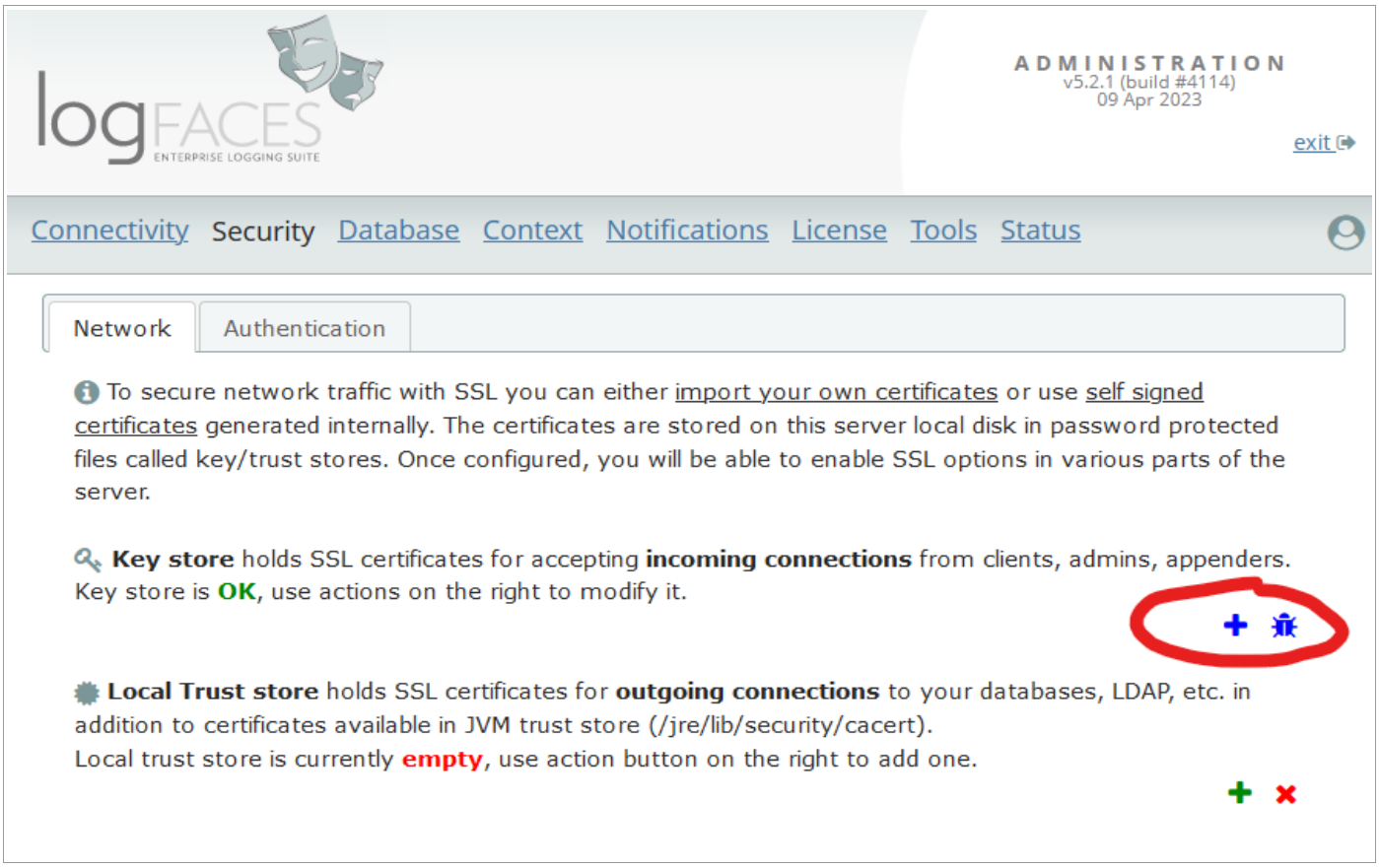


Figure 2.9.10: SSL setup

It is possible to use self signed certificates and if you choose doing so, the server can also automatically generate them for you. Beware of the limitations of self signed certificates, especially when it comes to the usage with browsers. For the production environments, the better choice is to use certificates issued by the well known certificate authorities (CA).

In order to import certificates into either of the stores, you will need to provide the certificate along with the its private key. The certificate must be in the X.509 DER encoded file, the private key must be PKCS#8 DER encoded file.

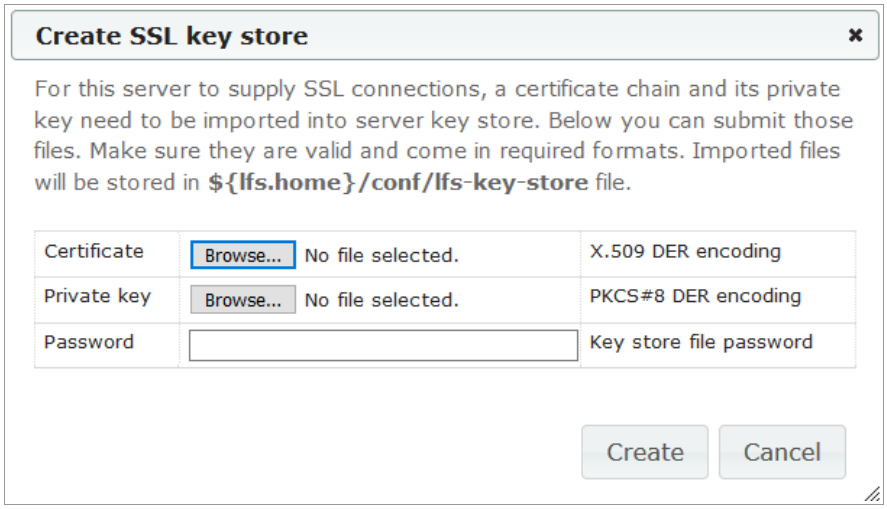


Figure 2.9.11: Importing SSL certificate into logFaces store

If your certificate comes as a chain of certificates and you must use them together, then follow the instructions below on how to create custom key store manually because it isn't possible (as of this writing) to use chain certificates in DER format.

2.9.6.1 Key store setup

Key stores are used for holding key pairs - certificates and private keys for securing incoming connections. The easiest way to generate a key store file is shown in previous section using admin interface. However, if this isn't possible the key store can be created manually or even re-used from an existing existing key store file.

If you have one already, the `/conf/environment.properties` file references them by means of these properties - location and password:

```
com.moonlit.logfaces.security.keyStore = ./conf/keystore-file
com.moonlit.logfaces.security.keyPass = my-password
```

Follow these instructions to manually create a key store file and populate it with your certificates and private key:

1. get hold of this tool named Key Store Explorer (or work from the JDK command line tools to achieve the same results outlined below)
2. place all certificate files into one file so that it includes the entire chain (just paste one into the other if there are several public keys), this is your PEM file for the import
3. make sure you have the private key file
4. in Key Store Explorer create new key store of type **JKS**
5. click on “Import Key Pair” and select **PKCS #8** format (if private key is encrypted make sure to obtain the password for the import)
6. select your certificate and private key files
7. when prompted for the alias - make sure to specify **lfs** (this is important as logFaces will look for this alias and if not found it will ignore the rest)
8. when prompted for the password - remember it for logFaces configuration
9. save the generated store and place it in the directory used in logFaces configuration as shown above

2.9.6.2 Trust store setup

Trust stores are used for holding certificates which can be used for the outgoing connections the logFaces server may do during its operation. By default, the trust store used is the one which comes with the JRE hosting the server instance. It will be found in `/jre/lib/security/cacerts` file of your JRE installation. In case of logFaces default JRE it will be just under the logFaces home directory.

In case default trust store doesn't contain the certificates needed for your setup, it is also possible to import your own certificates into another trust store located by default in `/conf/lfs-trust.store` file. We refer to this store as **local** or a **custom** trust store. This store can be created using the same technique described for the key stores above, but the easiest way to do this is to use the admin interface as shown below:

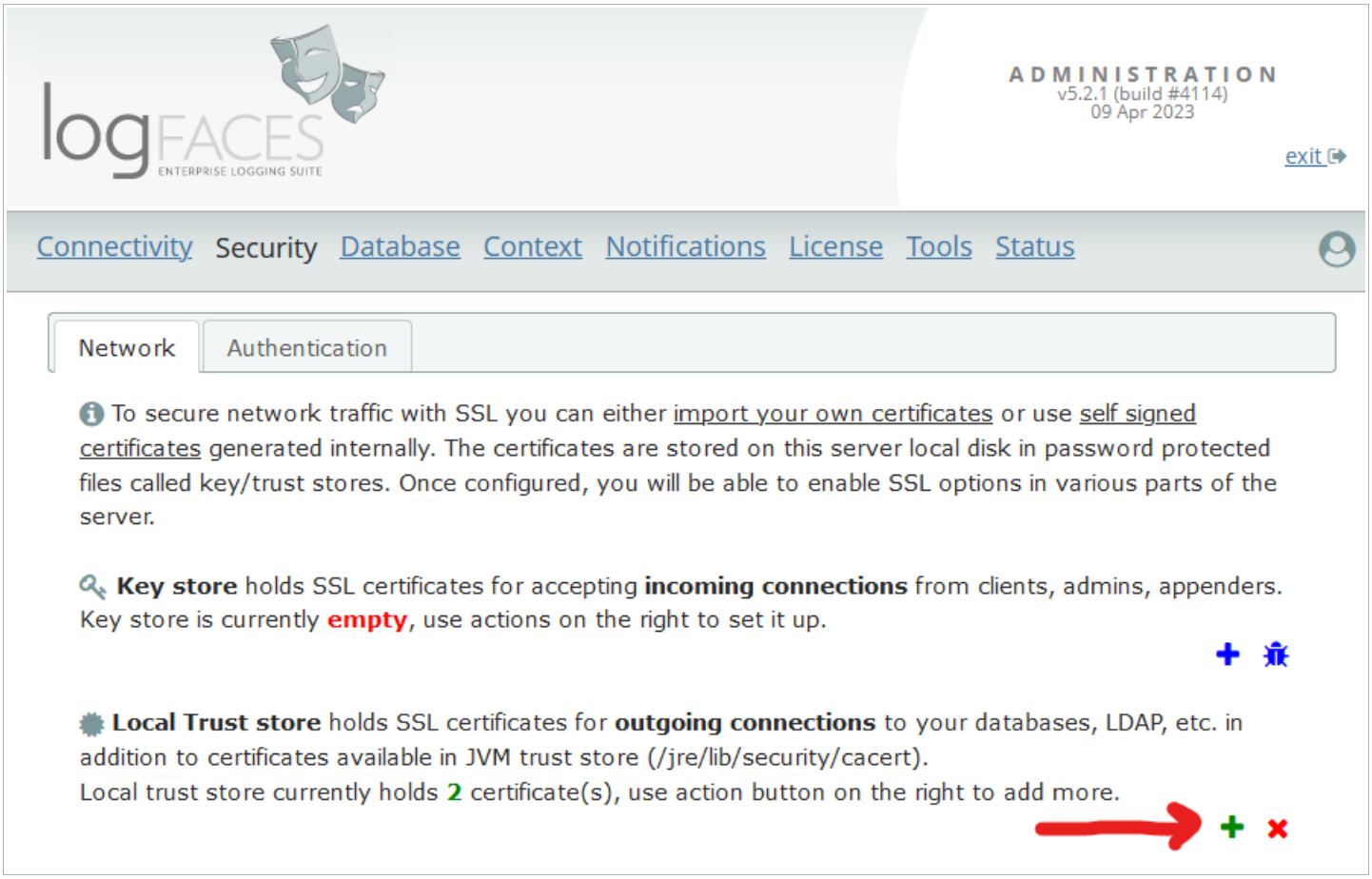


Figure 2.9.12: Adding certificates to custom trust store

Local trust store is configured in `/conf/environment.properties` file like this:

```
com.moonlit.logfaces.security.trustStore = ./conf/keystore-file
com.moonlit.logfaces.security.trustPass = my-password
```

2.9.7 Authentication

logFaces server can authenticate client connections. There are two authentication methods currently available - **SIMPLE** and **LDAP**.

Using SIMPLE authentication type requires a single user name and password which can be shared by team members which require access to the logFaces instance. This very basic authentication method is dedicated for preventing anonymous access to your server log data.

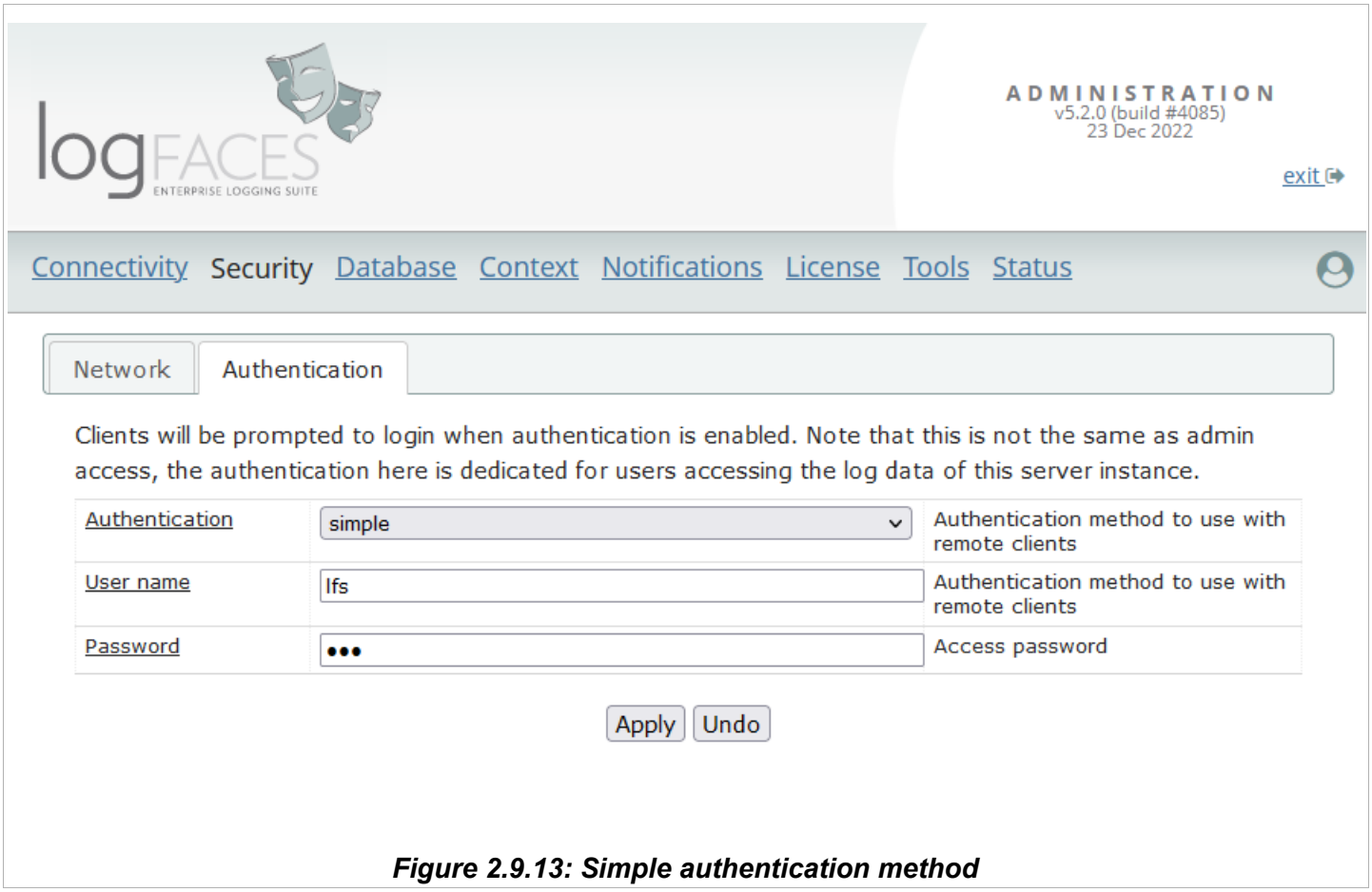


Figure 2.9.13: Simple authentication method

If you would like each team member to have an individual access to the logFaces server, consider using the LDAP authentication method as described below.

LDAP authentication method is a more sophisticated way to manage users access to your logFaces instances. It maps your logFaces instance to a name server in your organization holding the users and credentials. When enabled, any client trying to connect to logFaces server will be prompted to log-in. Collected credentials will then be delegated to the LDAP server which will do the actual authentication on logFaces instance's behalf.

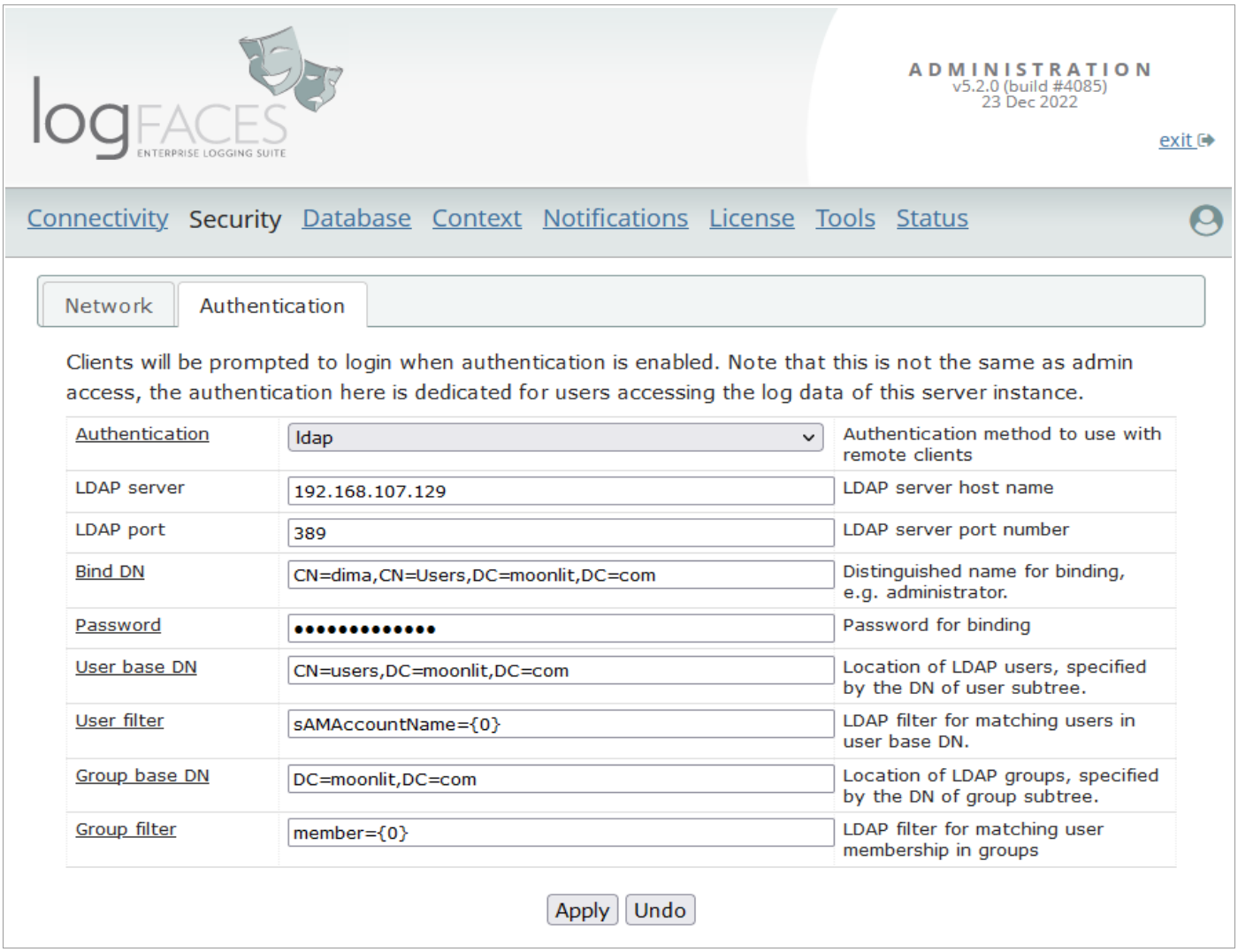


Figure 2.9.14: LDAP binding for authentication

LDAP server - host and port should be pointing to your LDAP server, make sure that it can be accessed from the network where logFaces server is installed.

Use SSL – use when LDAP server works over SSL. If LDAP server is using well known root CA, the communication should work straight away. Otherwise you will want to import your own certificates as described in Trust store setup section so that logFaces will be able to present them to its peers.

Bind DN - distinguished name for binding to the LDAP server, logFaces will use this DN in order to gain an access to user base. Usually those credentials are obtained from LDAP server administrator and must have permissions for walking user base tree.

Bind DN Password - corresponding password for the binding user.

User base DN - distinguished name corresponding to the location of users to be authenticated.

User filter - LDAP filter for matching users in user base DN. This parameter gives a very sophisticated way to match users in the user base. The default value **attr={0}** will match any user whose user ID is mapped to the attribute named '**attr**'. This attribute name varies in different LDAP implementations, for example in Apache DS this is normally '**uid**' while in MS Active Directory it show as **sAMAccountName**. Note the **{0}** parameter – it must be present all the time to match the actual user ID supplied by the user. When you want to do more complex matching of users, you can specify fairly complex LDAP filters in this field – please refer to LDAP documentation for the syntax details. Here is an example, the filter below will only match users from SALES organization unit

```
(&(ou=SALES)(uid={0}))
```

So, even when user is part of user base (uid={0}), it will only be attempted for authentication when she belongs to SALES unit. This way, having fairly large user base DN you filter out only relevant users for accessing logFaces.

Group base DN - location of user groups sub-tree. Groups will be used for authorization, if you don't need authorization - leave this field with default value.

Group filter - LDAP filter for matching user membership in groups, which is equivalent to granting authorities to users. When user logs in, its membership in user groups will be looked upon in “Group Base DN” and when found, this user will be granted authorities (as per group). And using this group filter you define how this matching should work, it is very similar to user filter except that it acts on user groups. The default value is again **attr{0}** which will match group members mapped to the attribute named **attr**. Usually this attribute is named '**member**' in most LDAP implementations.

Below is an example of filter which will gran authorities matching the group description (USA) and user ID matching the one provided by actual user during login.

```
(&(description=USA)(member={0}))
```

If you don't need this flexibility, just leave group filter to its default `member={0}`.

2.9.8 Authorization

After settings up and enabling LDAP for authentication, you can also enable authorization to control how users get exposed to log data. Authorization is optional.

The process is based on user group memberships.

Each user is normally a member of one a several user groups. You assign access criteria to groups of interest. When user logs in, logFaces will discover which groups this user is a member of and construct access criteria based on this information.

User groups are imported from your directory name service by clicking on "**Import All Groups**" or "**Import With Filters**".

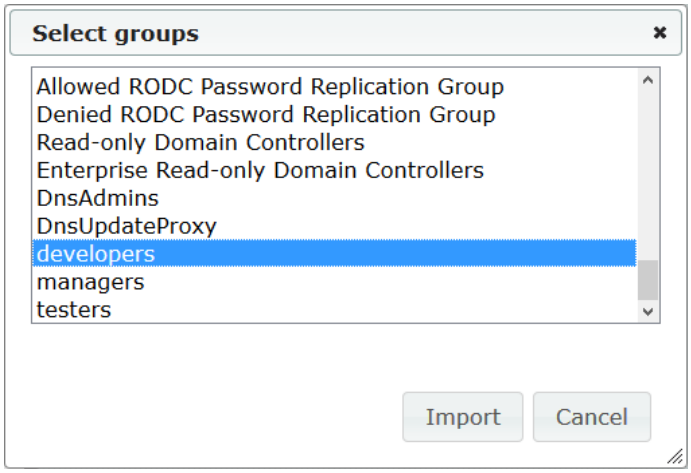


Figure 2.9.15: Importing user groups

Importing all groups will instruct logFaces server to walk your directory tree under **Group Base DN** and list all user groups found there.

Note that some directory servers may have constraints preventing enumeration of a very large lists. In such case try listing the groups with custom LDAP filter to narrow the search. For example, a filter like cn=D* will fetch only names starting with D.

Once group names are fetched, you will be able to select only particular groups of interest to use with logFaces authorization as shown on the illustration below.

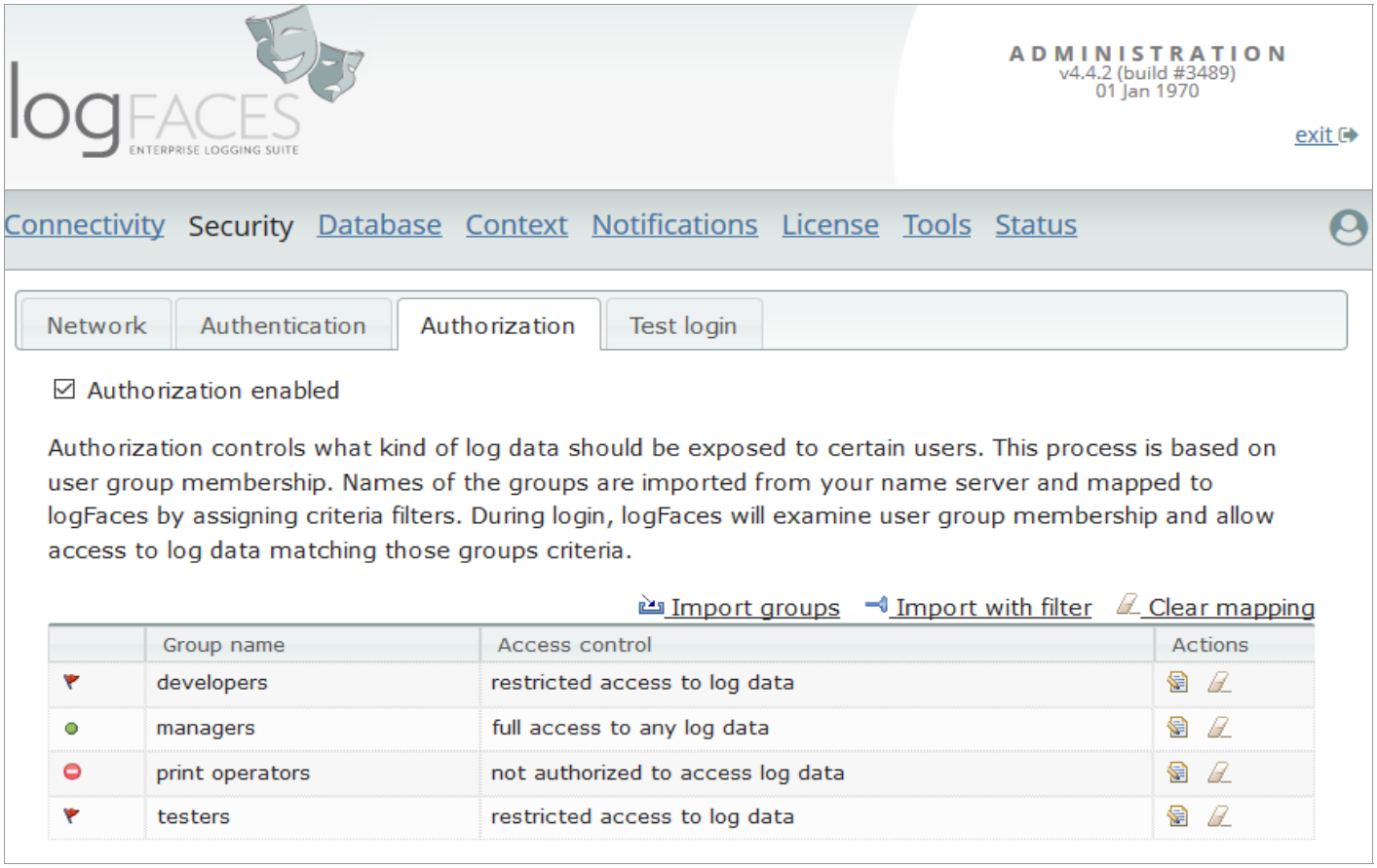


Figure 2.9.16: Authorization mapping

Once groups are imported, you can assign an individual access criteria. When a group doesn't have any criteria assigned, the users of this group will have no access to any logs. Criteria may contain any complex combination of rules, for example:

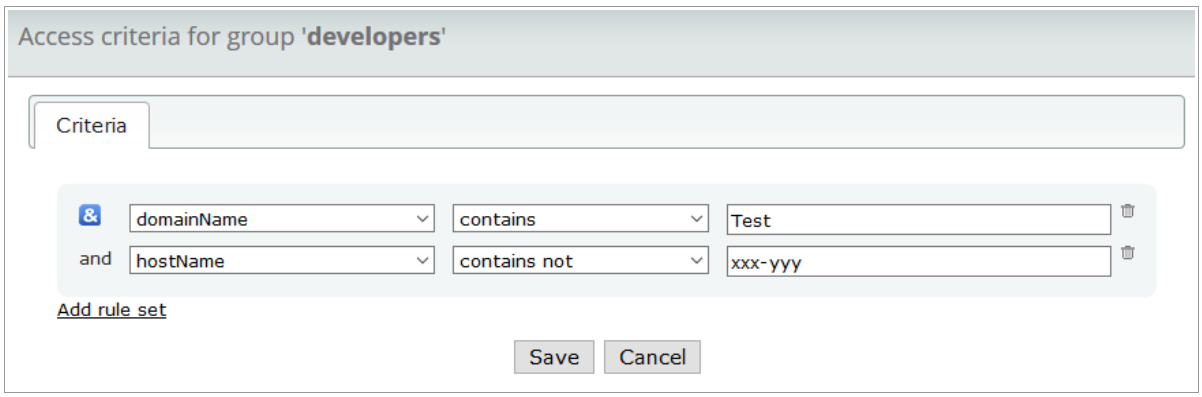


Figure 2.9.17: Access criteria example

2.9.9 Database

logFaces can work with different types of data storage which specified in the environment. Currently we support SQL, MongoDB and BigQuery storage types. This section describes settings relevant to all types of storage in general, but you will find the specifics in corresponding sections if needed.

2.9.9.1 General options

These are common options applicable to all types of databases used:

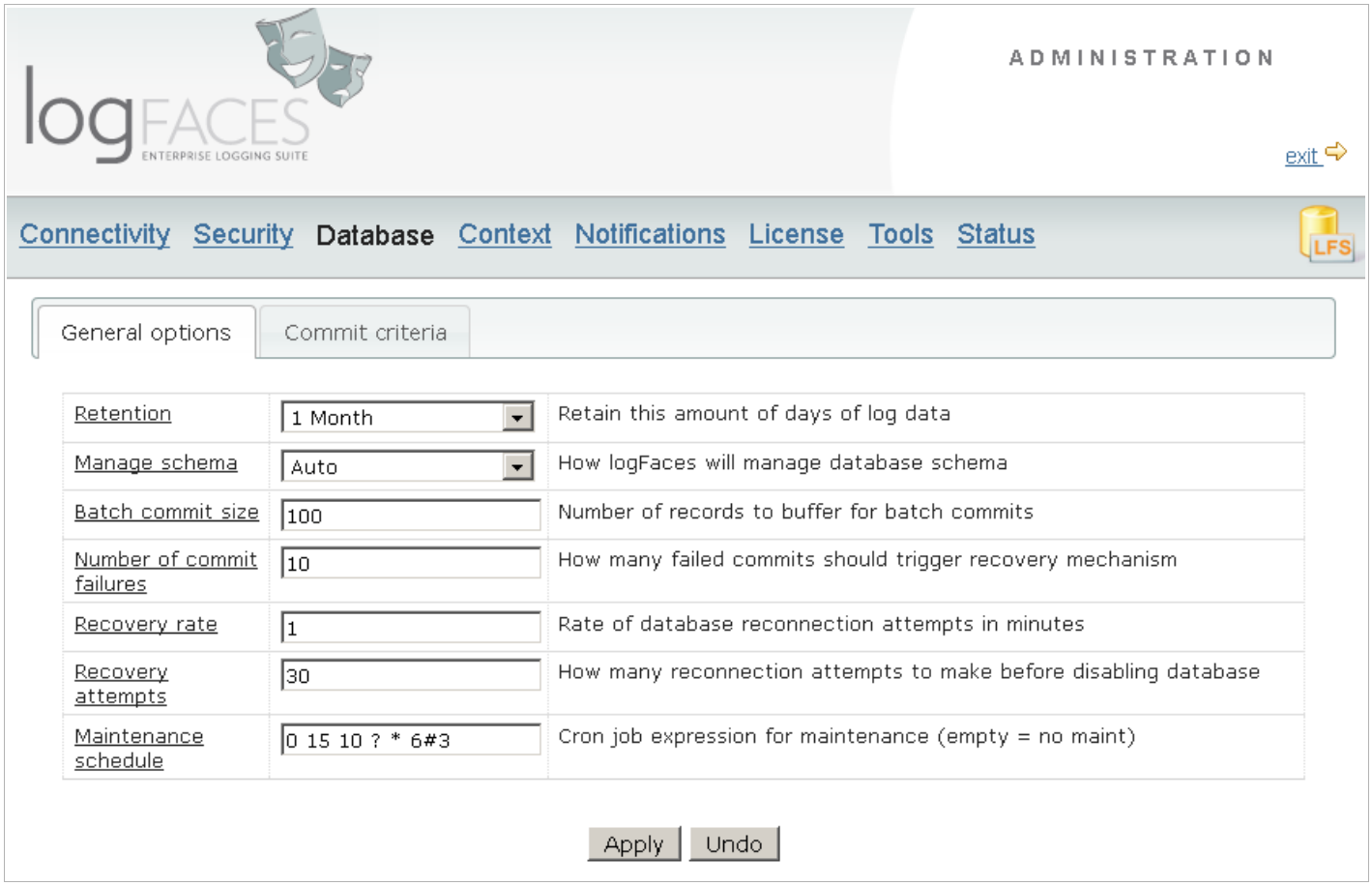


Figure 2.9.18: Database settings

Retention is specified in days of log. If you specify "1 week" for example, then latest week of data will always be available. As time goes, older records are automatically removed while new ones are appended. You should carefully specify this value according to your needs; it affects overall performance as well as disk space usage.

Manage schema option specifies whether server should enforce database schema or it should be managed externally. Default option is 'Auto' – server will create schema based on the templates provided in its configuration.

Batch commit size is the size of the buffer used to insert log statements into database as a batch. The smaller the buffer the more frequently commits will be performed. Depending on the data inflow intensity, the buffer should be adjusted in such way that it does less frequent commits. On the other hand, large commit buffer size could be stressful for the database. Optimal sizes are usually in range of 50 - 500. You should use higher number if your system has frequent spikes of log data, this will greatly improve the performance of server overall. Half full commit buffers will be committed with a timer job running every minute.

Number of commit failures specifies how many commits can fail in a row to trigger recovery mechanism. This mechanism is designed specifically for situations when database goes down for maintenance or temporally unavailable for some other reasons.

Recovery attempts rate specifies how frequently server should try reconnection with the database if there are connection problems.

Number of reconnection attempts specifies how many times to try before giving up on database and switching to a router mode. In the screen shot above, the recovery will run for 30 minutes trying to reconnect every minute. If during this time database comes back, everything will continue as normal. Note that during recovery process, incoming log statements are persisted on local disk and flushed into database when it comes back. When database is unavailable for a long time while application log keeps coming, there could be quite large amount of those backed up records.

Maintenance schedule is a cron expression which will trigger database maintenance job. This parameter is optional and applicable only for SQL databases. By default, maintenance of embedded database will compact local disk storage and optimize indexes. However, it is also possible to write your own set of SQL statements for the server to execute when the maintenance is due. Server will look for a file `/conf/maintenance.sql` file and execute each line in this file as an SQL statement(s), each line will be wrapped in its own transaction scope, one after the other. One line can include more than one SQL statement. If statement must be non-transactional, make sure to do COMMIT statement at the beginning of the line in order to end the transaction server starts automatically for the line. Make sure to properly test those statements before using them in the maintenance job. Also make sure to schedule the job to run off peak hours.

Here are some practical examples how to use the maintenance script:

The following lines used with Oracle database will reclaim unused space, optimize for better performance and rebuild all indexes.

```
COMMIT;VACUUM FULL;
COMMIT;ANALYZE;
COMMIT;REINDEX DATABASE lfs;
```

Note the COMMIT statement we used in each line; as mentioned above this is done in order to avoid the execution of the line statements inside a transaction which is started by default for each line. Basically you instruct the server to commit started transaction without doing anything and then execute the statements in that line. This should always be done when non-transactional statements are at hand.

Here is another example - this will perform a retention task by means of the maintenance job instead of the default built in retention mechanism. In some cases it will perform better especially with larger databases. The syntax used is of MySQL:

```
DELETE FROM lfs_log WHERE loggerTimeStamp < (UNIX_TIMESTAMP() - 7 *
24 * 60 * 60) * 1000;
```

When executed it will remove all records from the table storing all the logs which are older than one week.

Note how you can create very flexible retention rules by using this simple tool. You could choose to get rid of a less important logs to conserve space while keeping more important records forever, etc.

Maximum message length (MongoDB and BigQuery) caps log event message body to prevent flooding the storage with unusually sized log statements. Server will trim messages longer than this value. Trimming is done only on the 'message' part of log events. Value must be specified in kilobytes.

2.9.9.2 Commit criteria

In most cases you will want to specify which log events should persisted into the database and which events to should be discarded. This is done by what we call commit criteria. It's a collection of Boolean rules which you can manipulate to achieve a fine tuned filtering. The example below illustrates how to insure that everything except LFS application gets persisted, however if LFS application emits some warnings or more severe events, they will be persisted as well.

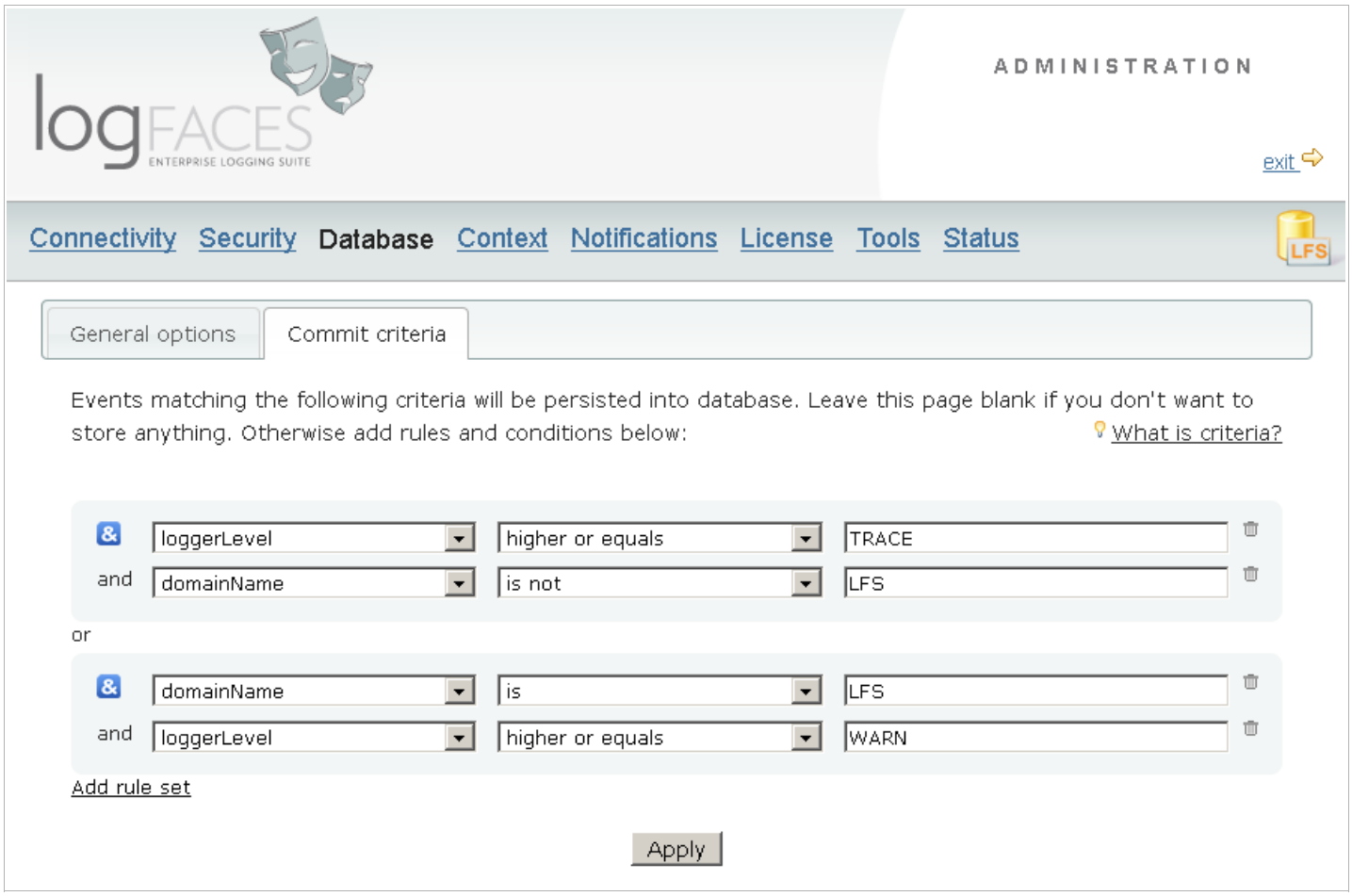


Figure 2.9.19: Commit criteria

2.9.10 Mapped Diagnostic Context

One of the advanced features in many logging systems is diagnostic context attached by the application to logging events. In log4j and its other flavors there is MDC – Mapped Diagnostic Context. You can read more about it here.

To provide convenient integration with MDC, logFaces lets you map your application context variables in such way so that they could later be used in queries and other displays.

With RDBMS you can specify up to 10 different context variables. Those variables are part of database schema thus the limitation. With MongoDB there is no such limitation, you can add as many as you need.

You can modify those names any time during run time, but it's best to setup the mapping as early as possible.

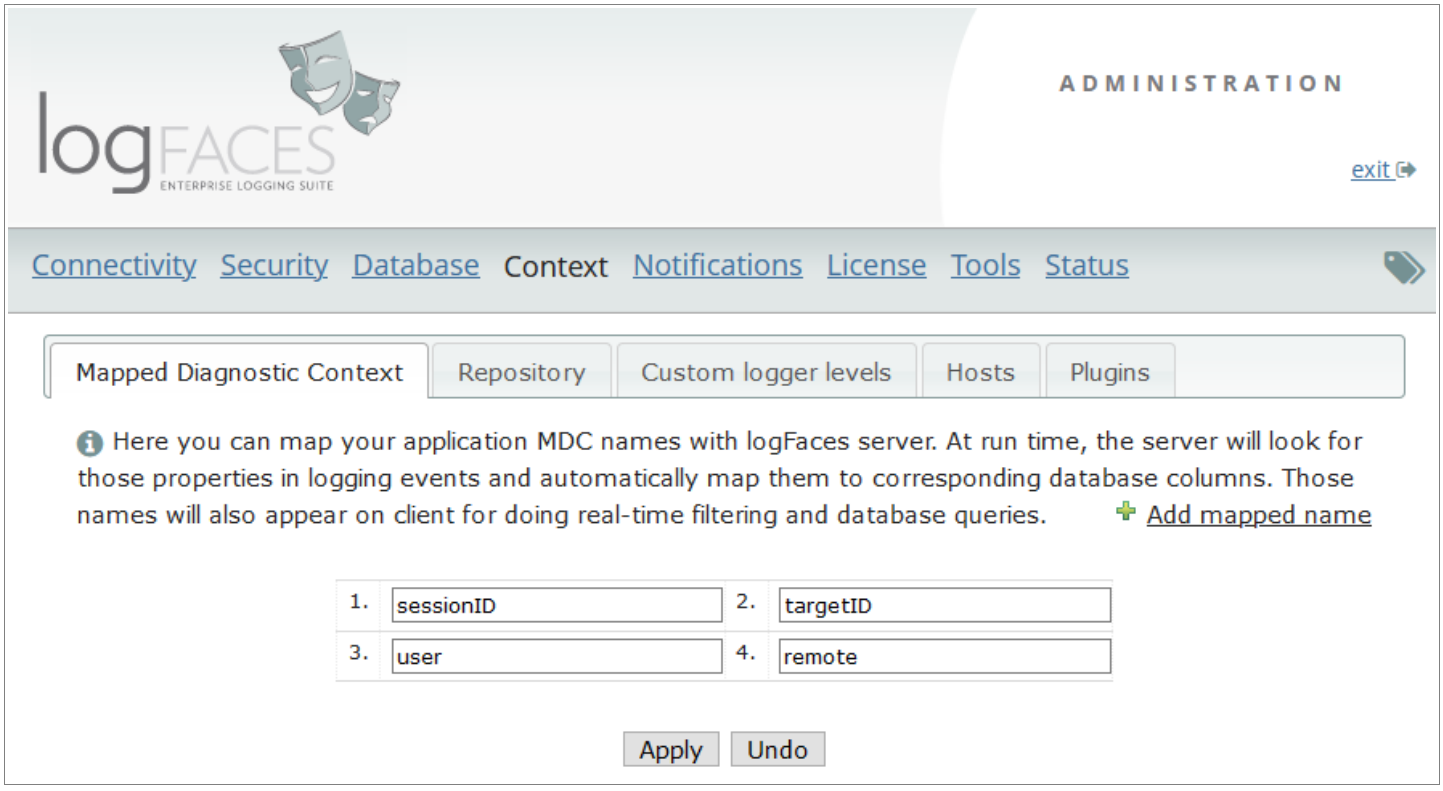


Figure 2.9.20: MDC settings

2.9.11 Repository

Repository holds a description of entire log storage, meta-data would be a good term to describe it. These are names of hosts, applications, loggers and exceptions ever recorded by the log server. Repository becomes more or less static once the system fills up with data and it's used mostly as a helper for getting lists of things. For example, when you need to fill in a query involving a host name, the repository will provide a list of all known host names. Repositories function automatically with any type of database used and there are several parameter to customize its behavior:

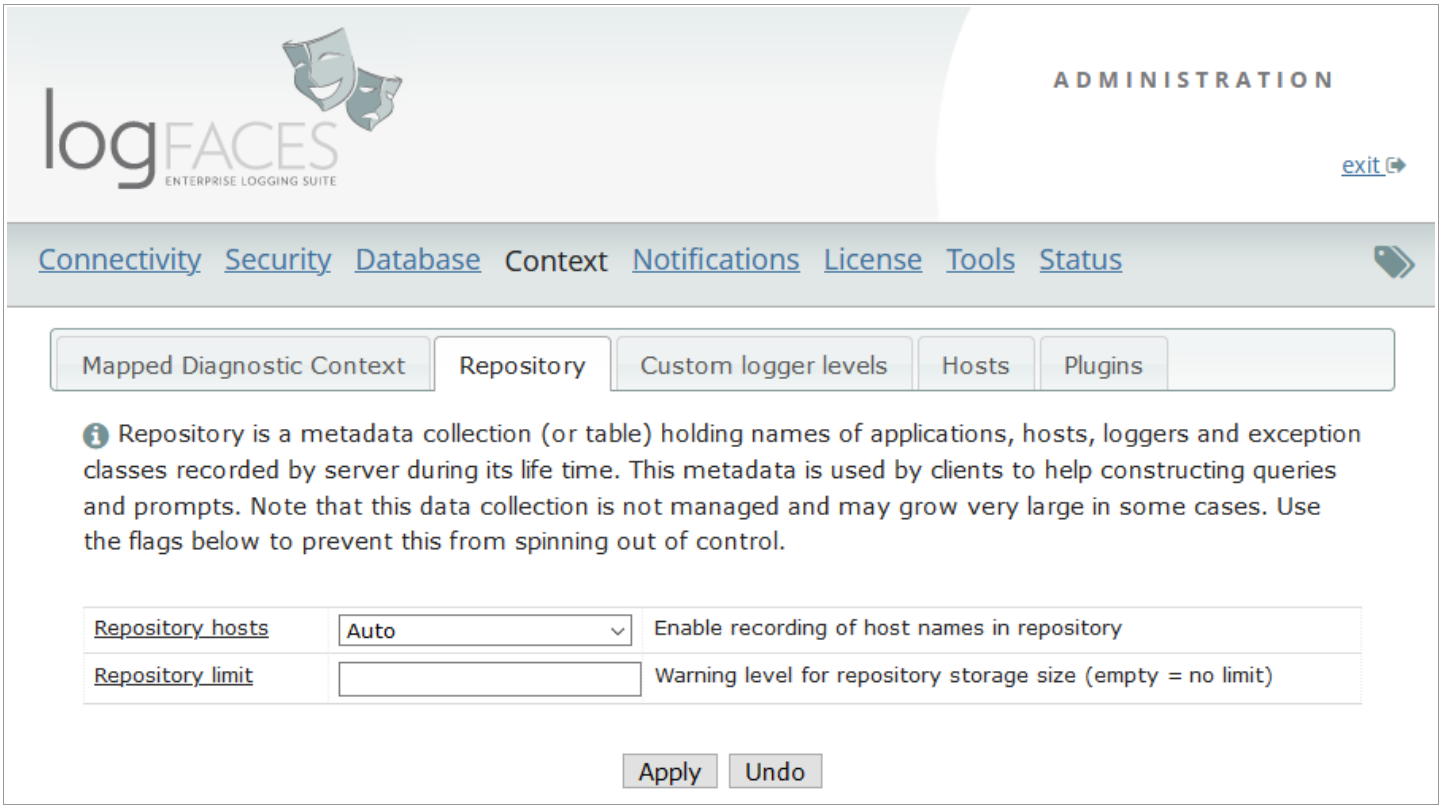


Figure 2.9.21: Repository params

Repository hosts option allows enabling or disabling the recording of host names where logs are originating from. You can disable this option if host names are not applicable to your setup, or when your entire system gets redeployed frequently.

Repository limit caps maximum size of repository collection. When repository grows larger than specified, the server will issue a warning messages to do the cleanup. Server don't have control over the size of repository size and it may grow very large depending on the usage. It is up to the user to keep this collection at a reasonable size and periodically remove stale and unused items. This can be done from client "Domains" view, or by using your database directly.

2.9.12 Custom severity levels

Often there is a need to use custom severity levels in addition (or even instead of) default levels provided by log4j. If this is your case, there is a way to specify custom levels in logFaces server :

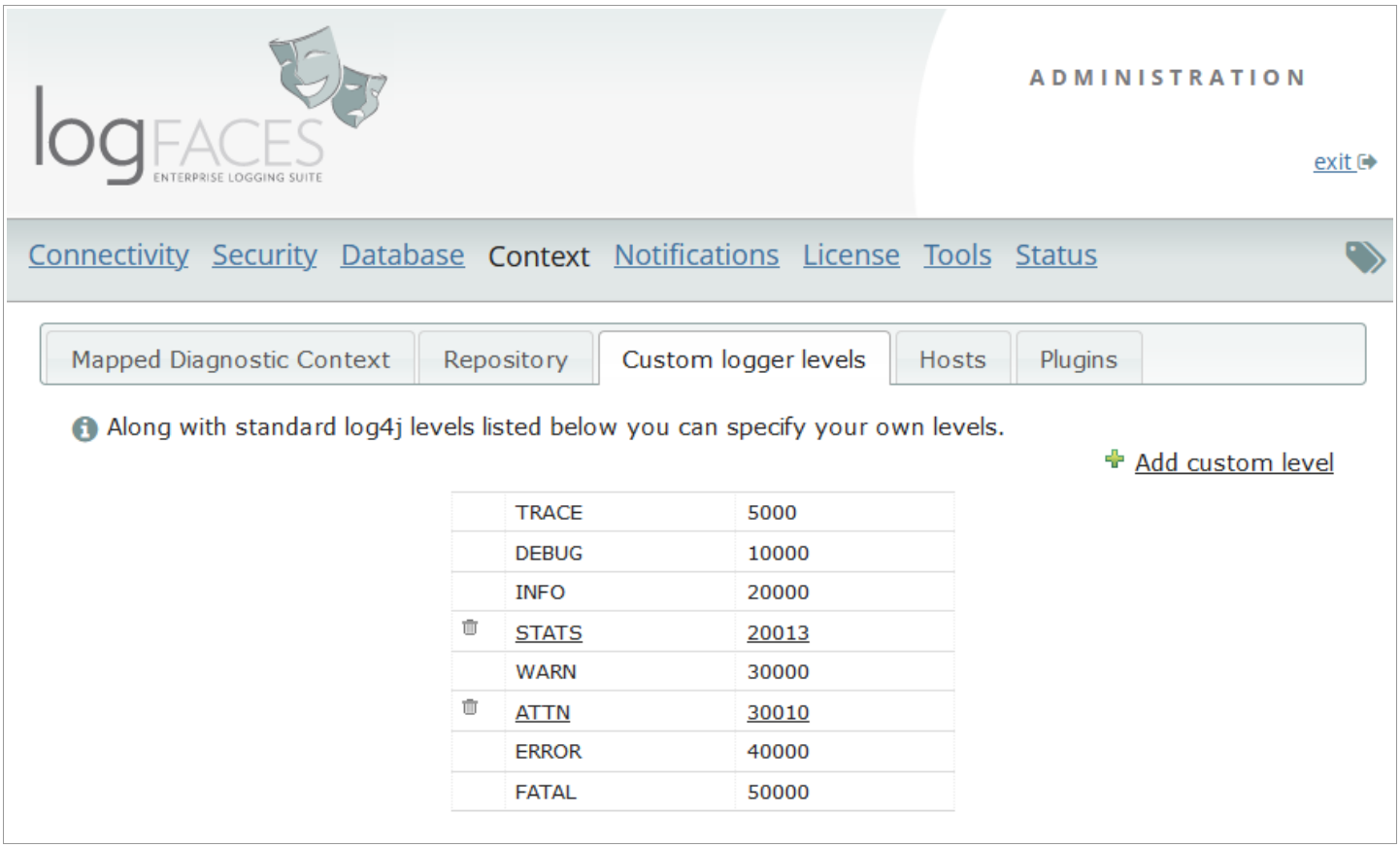


Figure 2.9.22: Custom severity levels

Note how **STATS** and **ATTN** levels are defined – these are custom levels in addition to defaults. Once you add custom levels, they become available throughout the system – in filters, reports, triggers, etc. Those levels will also appear on client side so that users can utilize them in queries and displays.

2.9.13 Hosts mapping

All log events are stamped with the origin host name (or IP address). Usually it's done by appenders but it really depends on the setup and usage pattern. For example, if you are using TCP java appenders, you application will try to obtain the host name from the computer it runs on, which may result in something you may or may not like to see. Sometimes it is just convenient to replace certain host names or addresses with something more meaningful in your environment.

Using hosts mapping you will be able to achieve just that – specify the original names and their substitutes. When log events received, the server will try to replace their origin accordingly. The content of hosts mapping follows the format of standard properties file.

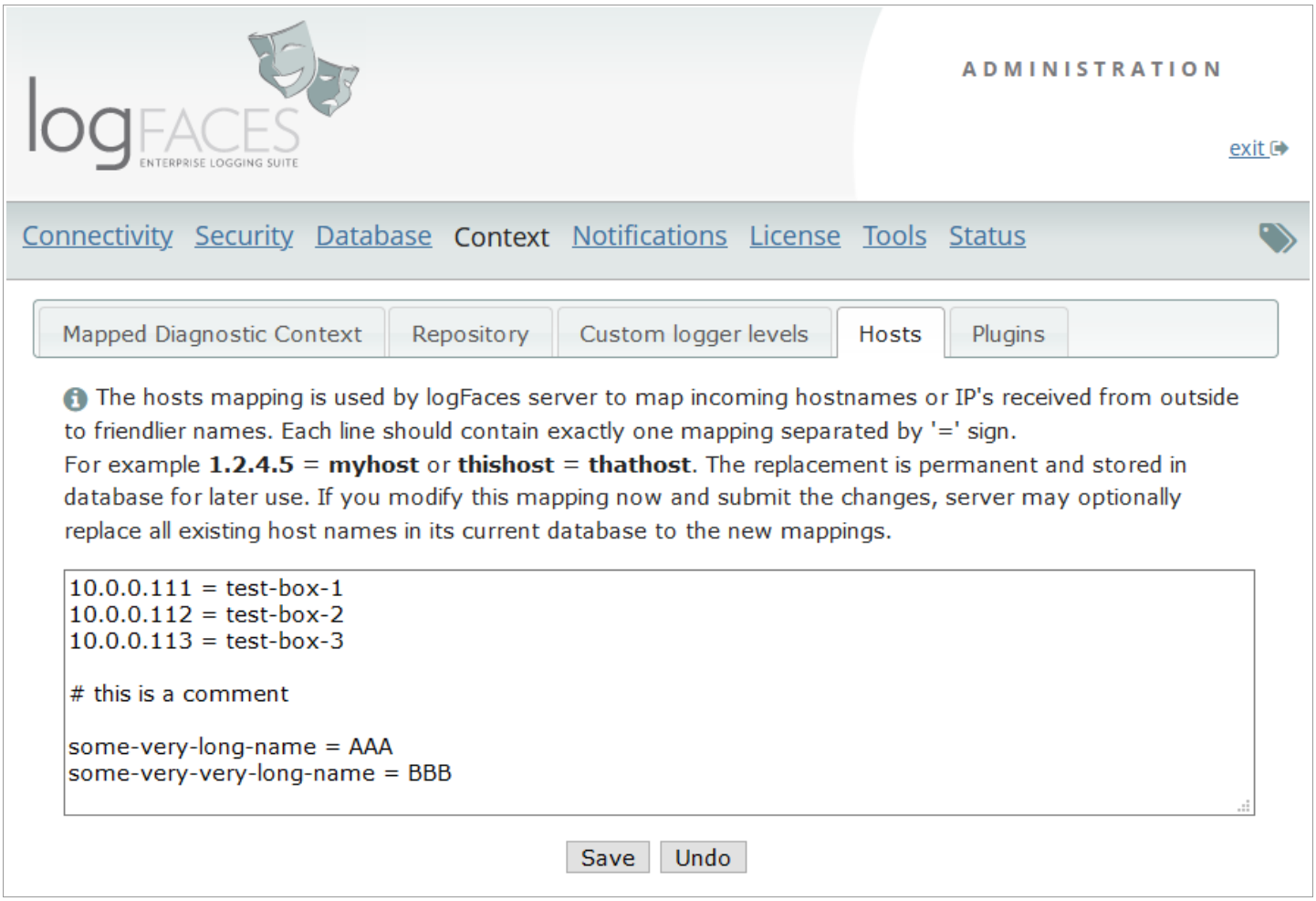


Figure 2.9.23: Host names mapping

2.9.14 Plugin management

Plugins are modules written by users who want to extend logFaces functionality. The process of creating plugins is outlined in this dedicated section and once you have working and tested plugin module, you can upload it to your server instance and let your team members use it in their client application or utilize it in Pivot Chain locally.

This page allows basic management of plugin modules hosted by your server instance. It involves **adding** new plugins, **removing**, **updating** existing plugins and **testing** them.

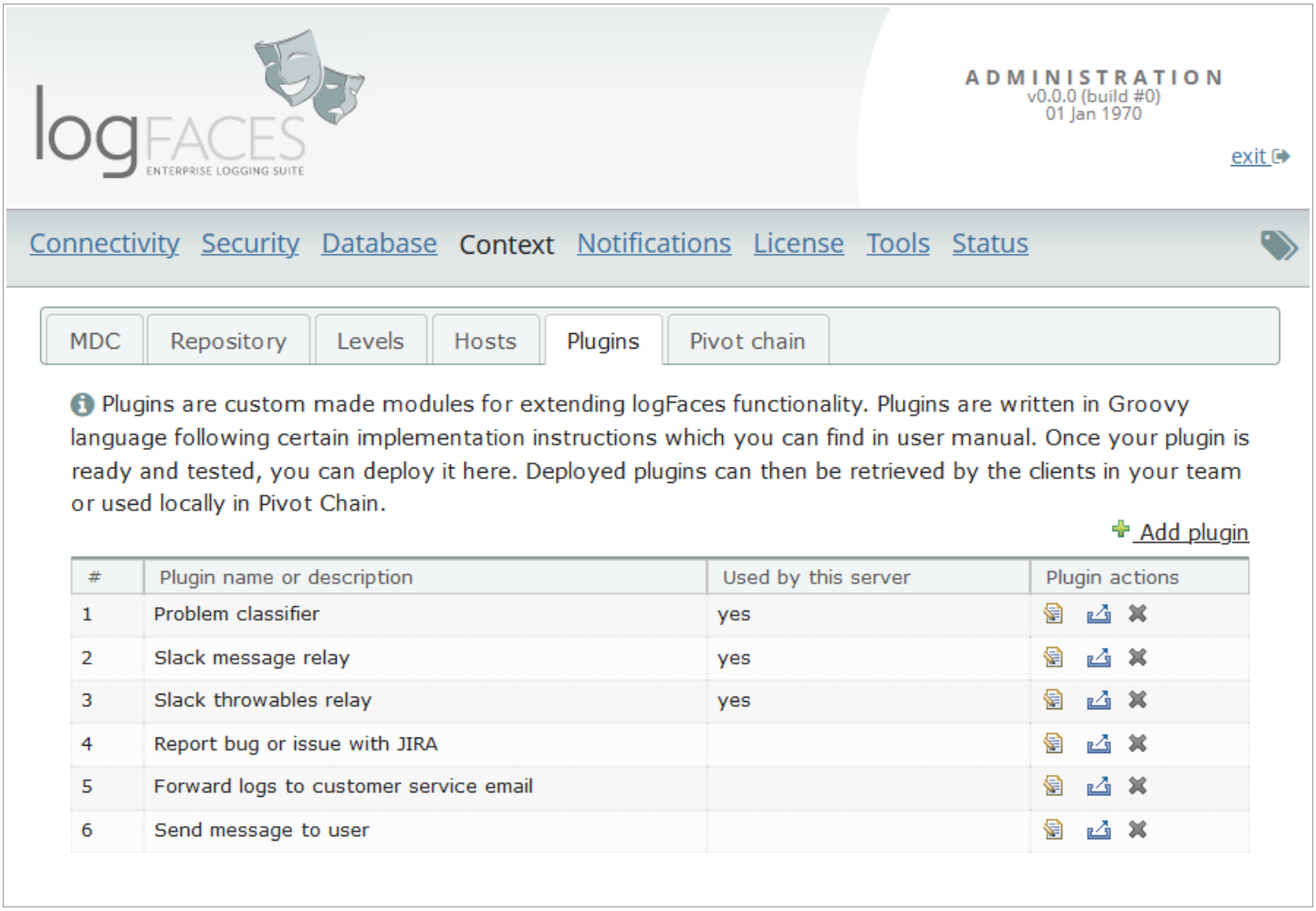


Figure 2.9.24: Plugins management

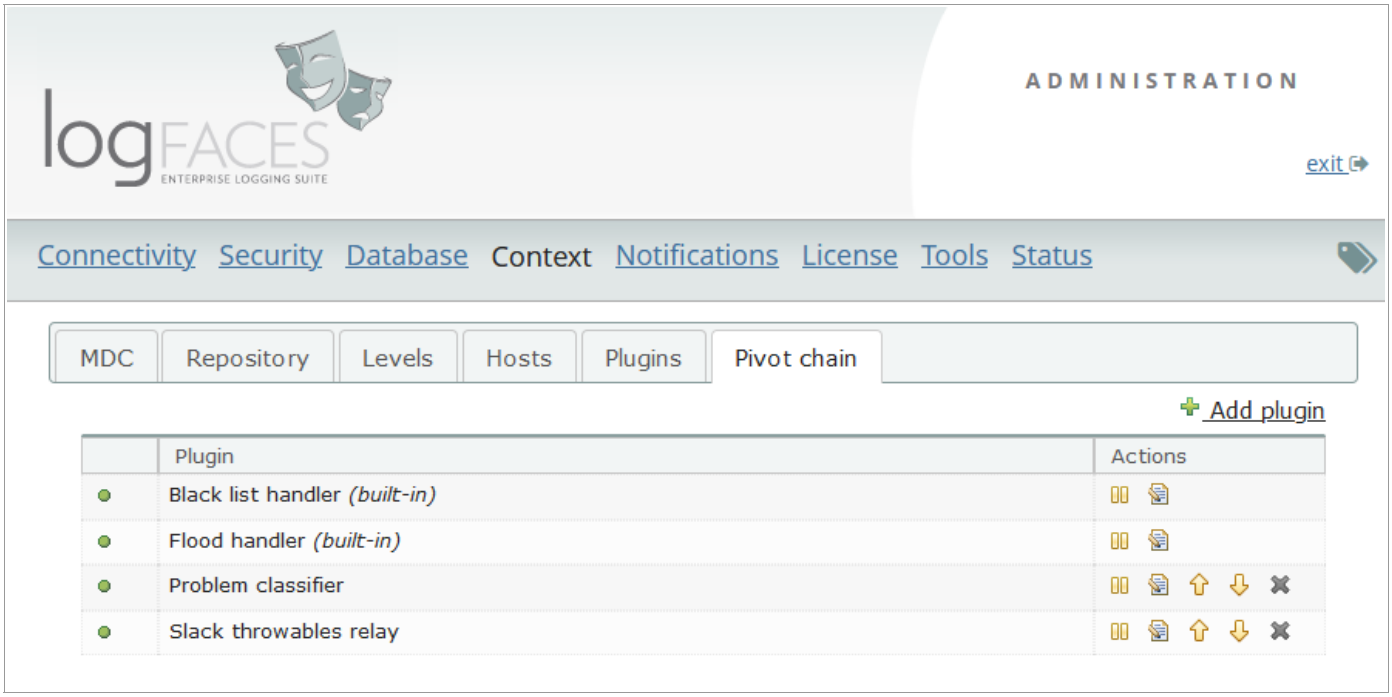
2.9.15 Pivot chain

When log events arrive into logFaces server, they pass through a sequence of modules before they get dispatched to the database, triggers or clients. This sequence, or a **pivot chain**, is designed to allow **pre-processing** of events before they arrive at their destinations. You can design your own modules (in the form of plugins) and use them in the pivot.

A pivot plugin typically morphs an incoming event by adding some properties, modifying content, or discarding the event. Along with the data mutation, the plugin can also act on log events by invoking custom functionality. Keep in mind that this may directly affect the server throughput because the processing in the pivot chain is sequential, the events visit every module in the pivot chain one after the other unless discarded on the way.

The server comes with two built-in pivot modules, the **black list** and the **flood management**. You can enable, disable and customize them to suit your needs, these are the native server modules. Note that the black list module is always first in the pivot chain followed by the flood management. We may add more pivot modules in the future.

Custom plugins will be called after built-in plugins have had their say. In the example below there are two custom plugins, the *problem classifier* determines the type of error (if the event is an error) and the *slack relay* to send the event to Slack channel, if it matches the criteria. Note that you can also re-order the custom modules in the pivot chain and modify plugin arguments.



2.9.16 SMTP settings

Email settings will allow logFaces server to send e-mails when required by **reports** and **triggers**. Here you define outgoing SMTP properties and also can verify that these settings are correct by sending test email to some recipient.

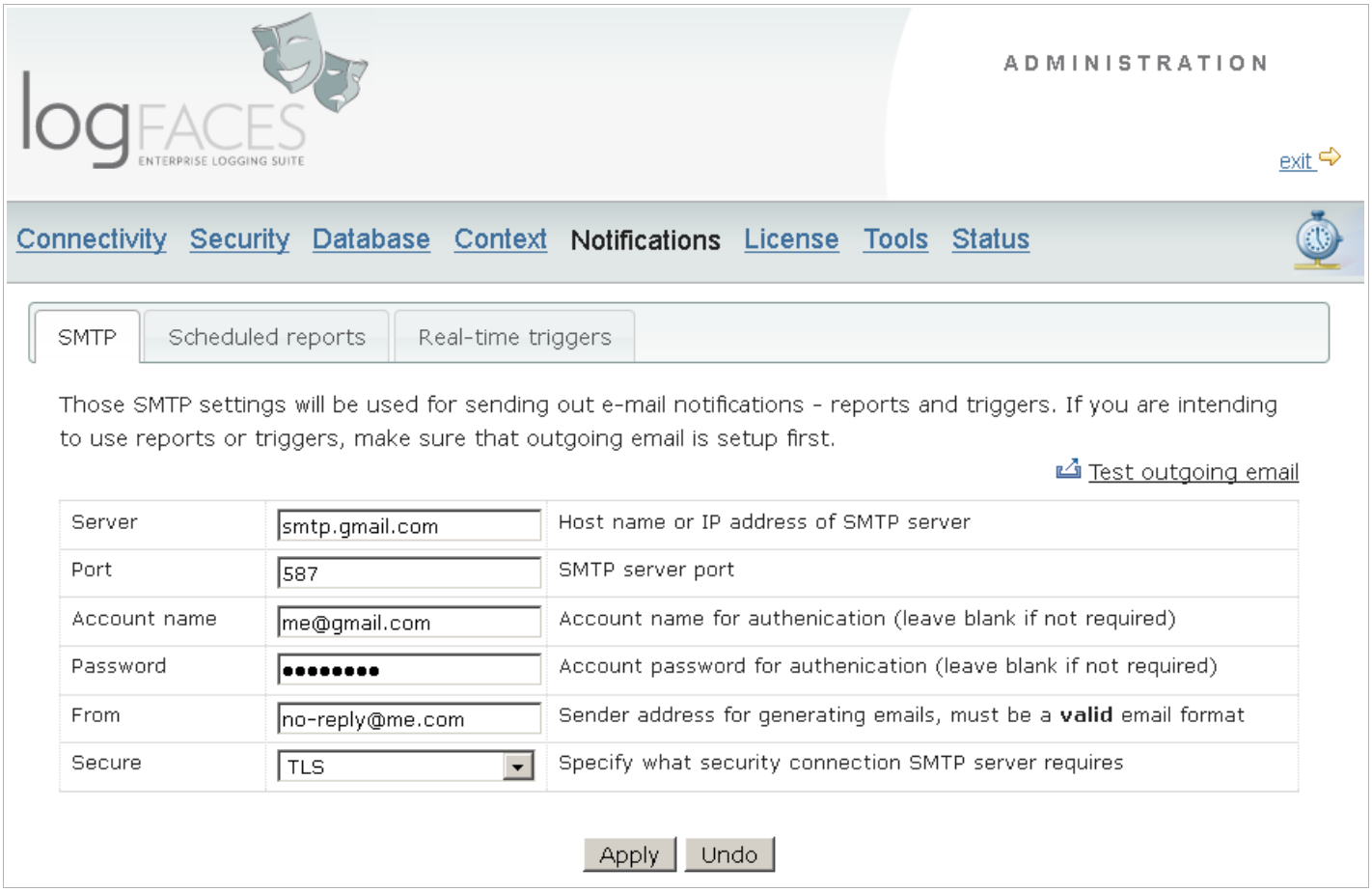


Figure 2.9.25: SMTP settings

Click on "**Test outgoing email**" link to verify that logFaces can send e-mails successfully. Should anything go wrong, you will be shown an error describing the cause. If everything was correct, you will receive an acknowledging email.

2.9.17 Reports

Reports are custom log files that server periodically generates according to the schedule and criteria query. Reports are then emailed to the recipients of your choice. Reports are organized in a list where you can see the overall information. Reports can be enabled or disabled – the rightmost icon indicates that second report is disabled in the example below. Disabled report stays in the system but doesn't actually do anything until you enable it.

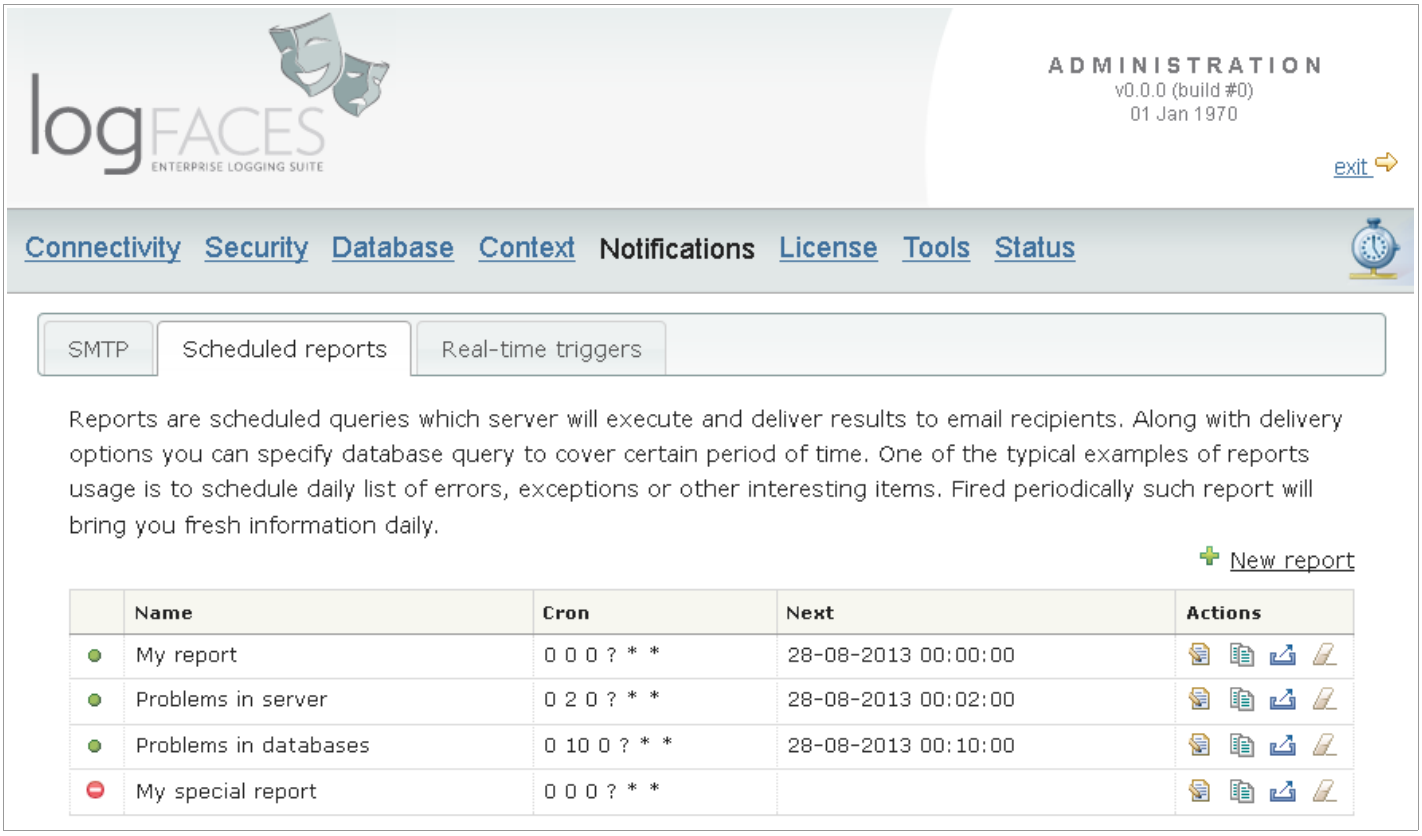


Figure 2.9.26: List of reports

The list also shows the cron expression which drives the report schedule, its closest fire time and links to manage each report individually.

Note that reports can also be individually tested (click on icon), - this is quite useful because it lets you receive real data instantly without waiting for the complex cron expression to trigger the report.

Each report comes with a bunch of parameters explained below as well as its criteria query. The query will be executed when report trigger fires. The results of the query will be then packaged into a log file, zipped if necessary and sent over to your recipients.

The example below illustrates a typical report – it will fire every midnight and if there is anything in the query, it will send an email to our support. It covers past 24 hours and flags high email priority. Look at the query it does – we want WARN+ events coming from com.moonlit.logfaces package.

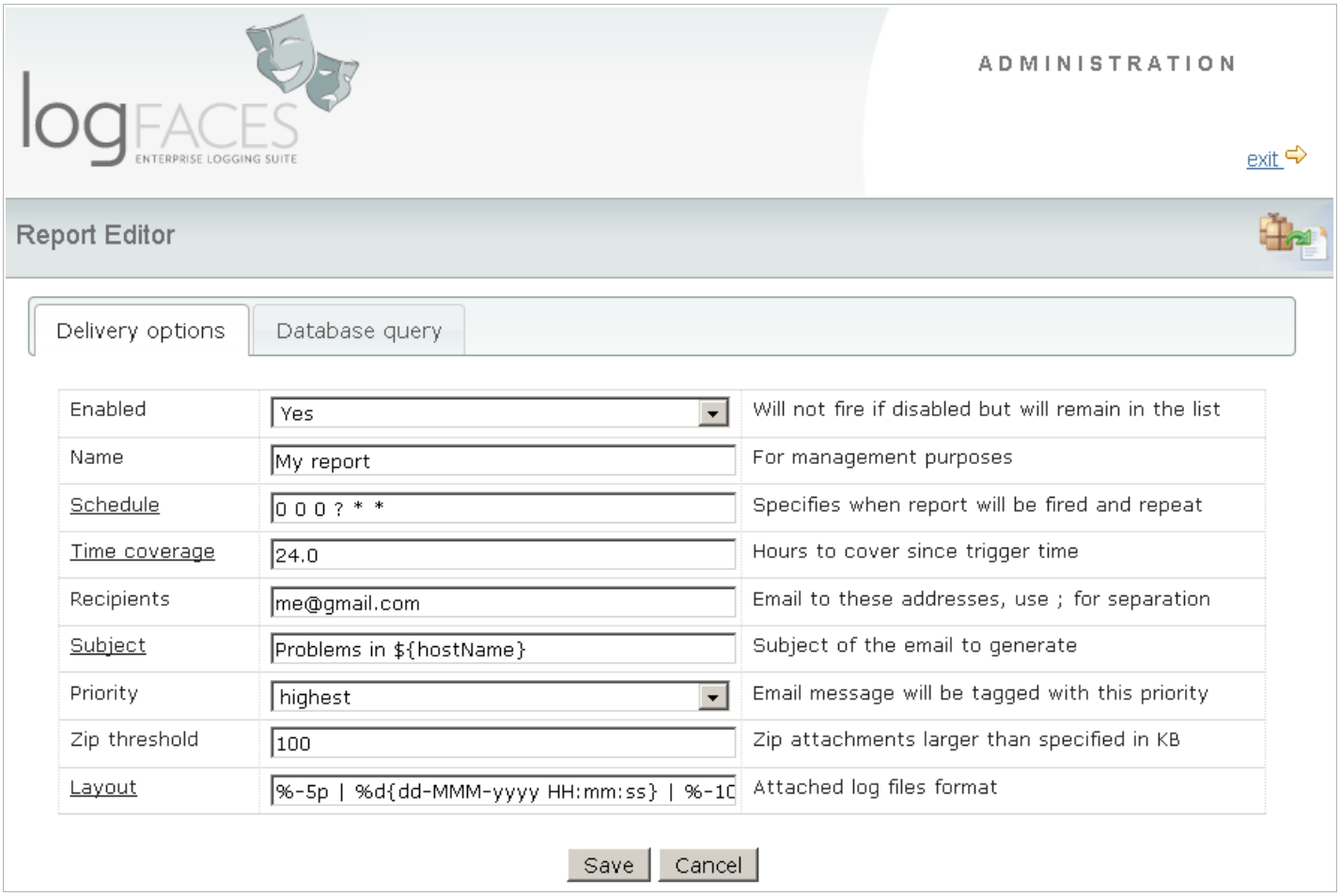


Figure 2.9.27: Report delivery options

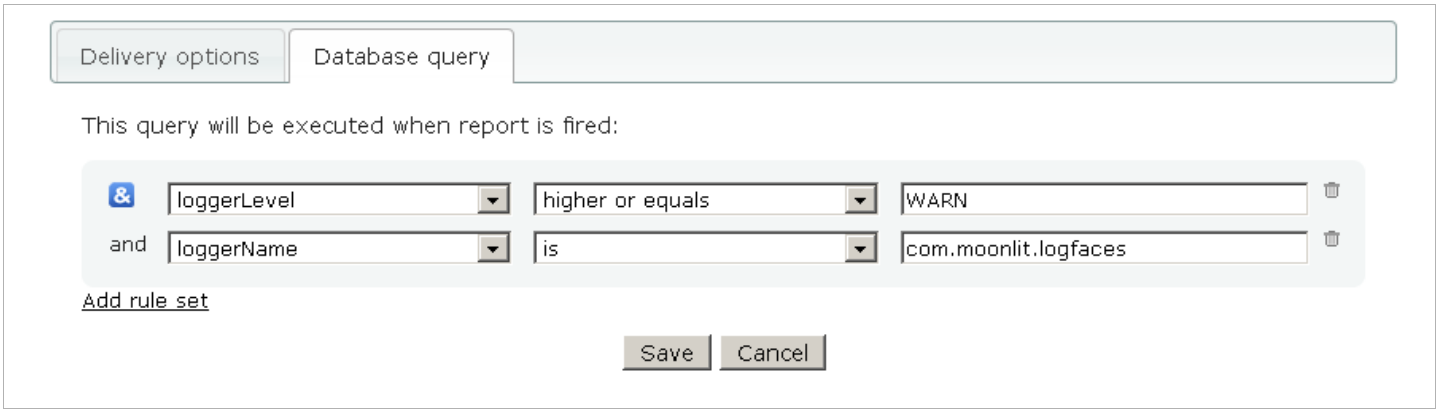


Figure 2.9.28: Report query

Enabled - when unchecked, the report will stay in the system but will never fire. You can enable it any time and it will fire at the next schedule slot.

Report name – this field is used only for the management purposes, give it a friendly name so that you could easily find the report in the list.

Cron expression – is an expression which specifies when and how to fire the report. Cron expressions are very flexible and used to make fairly complex scheduling rules. You can get more information about cron expressions here.

Time range to cover – this specifies the time range which report query will cover, the count begins from the actual trigger time backwards. For example, if you want to cover single day and your report is fired daily, specify 24 hours.

E-mail to - list of recipients to receive the e-mail (use semi column as separator)

E-mail subject – this text will be used in email subject when report is dispatched. Note that you can use ${variable} notation here where variable could be **domainName**, **hostName**, **loggerName**, **message** and any of the mapped MDC names. When report is built, this variable will be substituted with the corresponding value taken from the first log statement in the report.

Mail priority - e-mails can be flagged with standard e-mail priorities (highest, high, normal, low, lowest).

Zip attachments - specify a maximum size of log file in KB; if attachment file will be larger than specified, it will be automatically zipped.

Layout – specifies how to layout the text in the log files. LogFaces is using log4j formatting rules; you can find more details here

2.9.18 Triggers

Triggers are similar to reports except that they are not scheduled but fired immediately when certain conditions are met. Conditions are based on the log data going through the server. By specifying criteria you will be able to detect particular log statements from particular sources. In addition to this, you can also specify how many of such events to capture and within what time span they should be captured in order to fire the trigger.

Like reports, triggers are listed to give you an overall view of what triggers are there, what is enabled and when and how they get fired. Trigger may raise an **Alert** when fired if this option is specified in the delivery settings of the trigger, see below. Alerts are persistent states which are used to visually indicate that trigger **was fired in the past** and may require user attention. When trigger raises an Alert, its firing time label will be emphasized with a brighter color. Alerts can be acknowledged (cleared) by users.

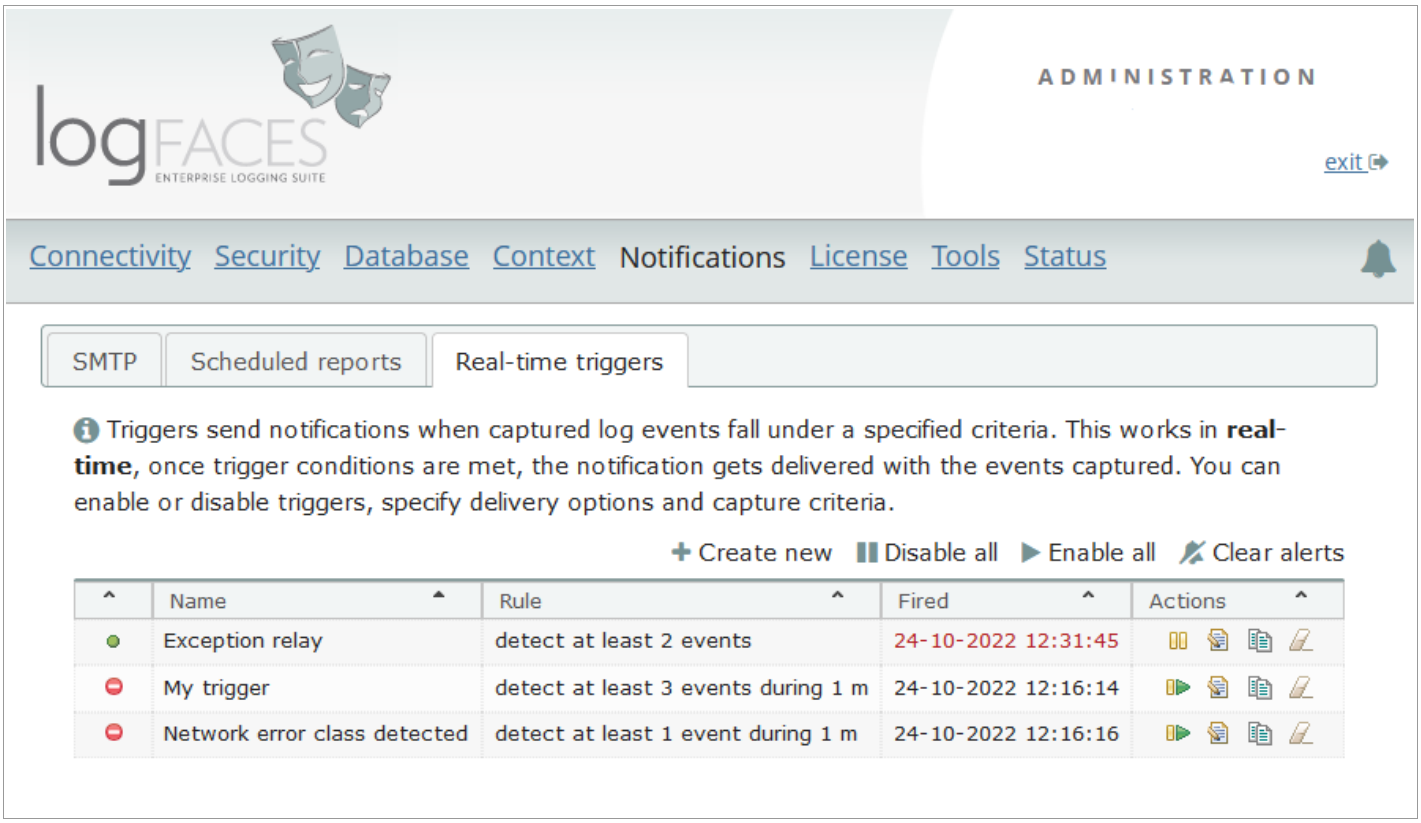


Figure 2.9.29: List of triggers

2.9.18.1 Delivery options

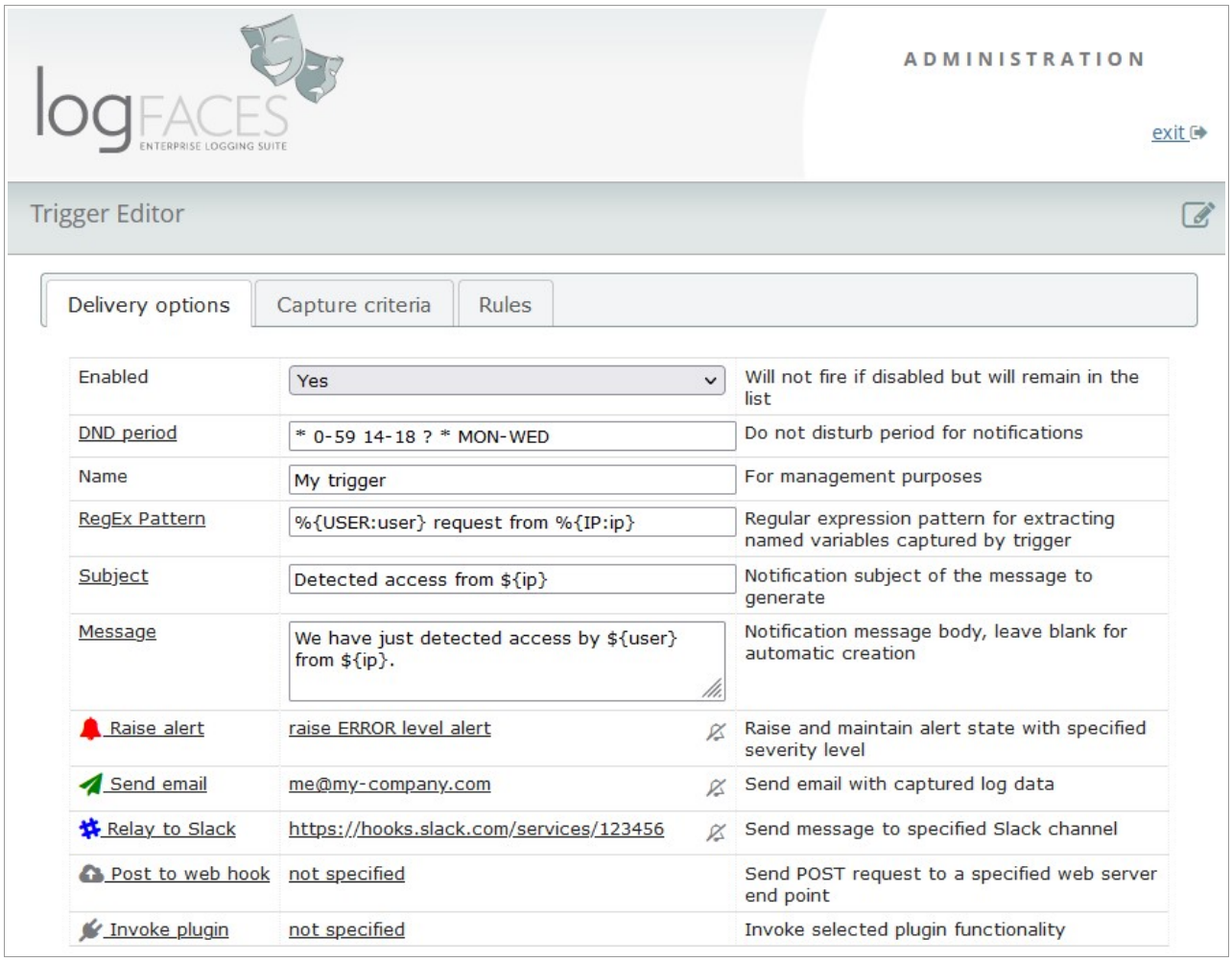


Figure 2.9.30: Trigger delivery options page

Enabled – the trigger will not fire if disabled, but will stay listed.

DND period - “do not disturb” period is when trigger should not send any emails while staying active. This period is specified as cron expression. For example, *** 0-59 14-18 ? * MON** will silence the emails during hours 14:00 - 19.00 each Monday.

Name – for the management purposes

RegEx Pattern – Regular expression pattern with named groups for matching variables in log messages captured by the trigger. These variables can then be referenced in notification subject or message body to make a notification more descriptive with specific context. In the example above, the variables '**ip**' and '**user**' will be extracted from the captured messages and used in subject and message fields using ${variable} notation. This feature is optional. Leave blank if not used.

Subject – will be used with notification when trigger is fired. Note that you can use **${variable}** notation here where variable could be **domainName**, **hostName**, **loggerName**, **message** and any of the mapped MDC names or a split expression (see below). Those variables will be taken from a first log event caused the trigger.

Message – will be used with notification when trigger is fired. If not specified, the server will generate default message. Like with the subject, it is possible to use **${variable}** notations.

Trigger notifications can be further processed with one or more extensions:

- **Alerts** - will raise a server Alert which is a server persistent state reflecting the trigger activation. Alerts are displayed in client UI to raise awareness of past trigger events. Users can then query the Alert related logs and acknowledge them if needed.
- **Emails** - standard SMTP delivery with attachments to one or several recipients. It's possible to customize the priority and layout format of attached log file.
- **Slack** - push notification to Slack channel of your choice. Slack integration involves obtaining the web hook URL from your Slack account. logFaces will post customized messages to this URL. More details can be found here.
- **Web hooks** - logFaces will POST JSON content to any HTTP/s URL of your choice. The content has the following format `{subject: 'string', message : 'string'}` where message and subject are taken from the delivery options.
- **Invoke plugin** - logFaces will call specified plugin when trigger is fired.

2.9.18.2 Capture criteria

Capturing criteria specifies which events should be going through the trigger. Only those events which match specified criteria will participate, others will automatically discarded. It is generally a good idea to have capturing criteria as narrow as possible.

Example below will capture *NullPointerException* from a particular host and application, and only those events may eventually fire the trigger.

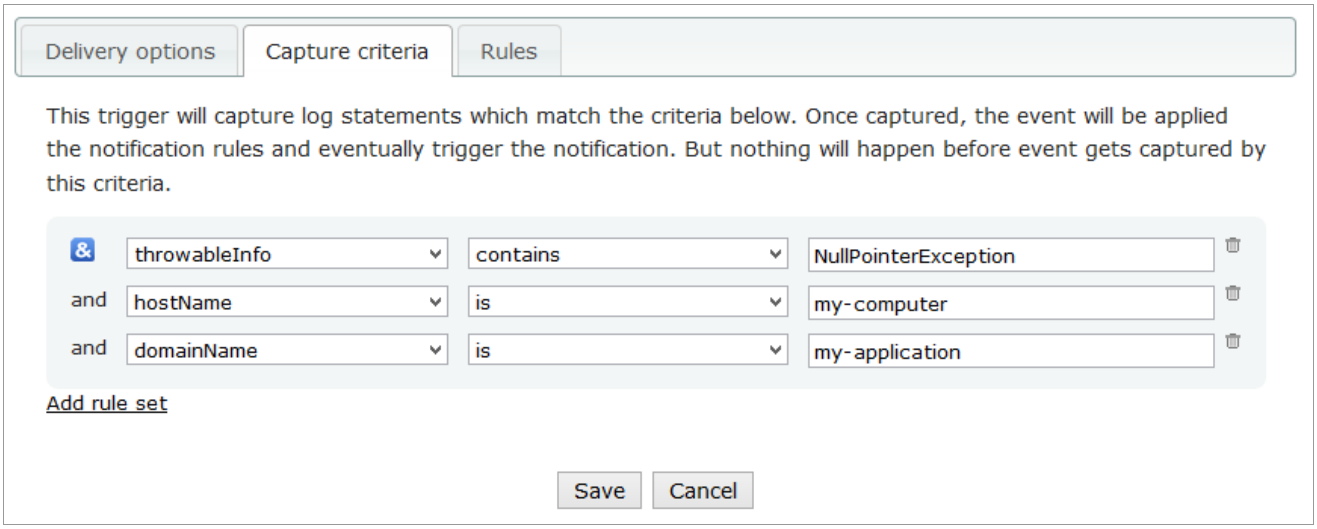


Figure 2.9.31: Trigger capturing criteria

2.9.18.3 Rules

Once log event is captured by trigger criteria, the trigger will apply the following rules and then send email notification according to the Delivery Options specified above.

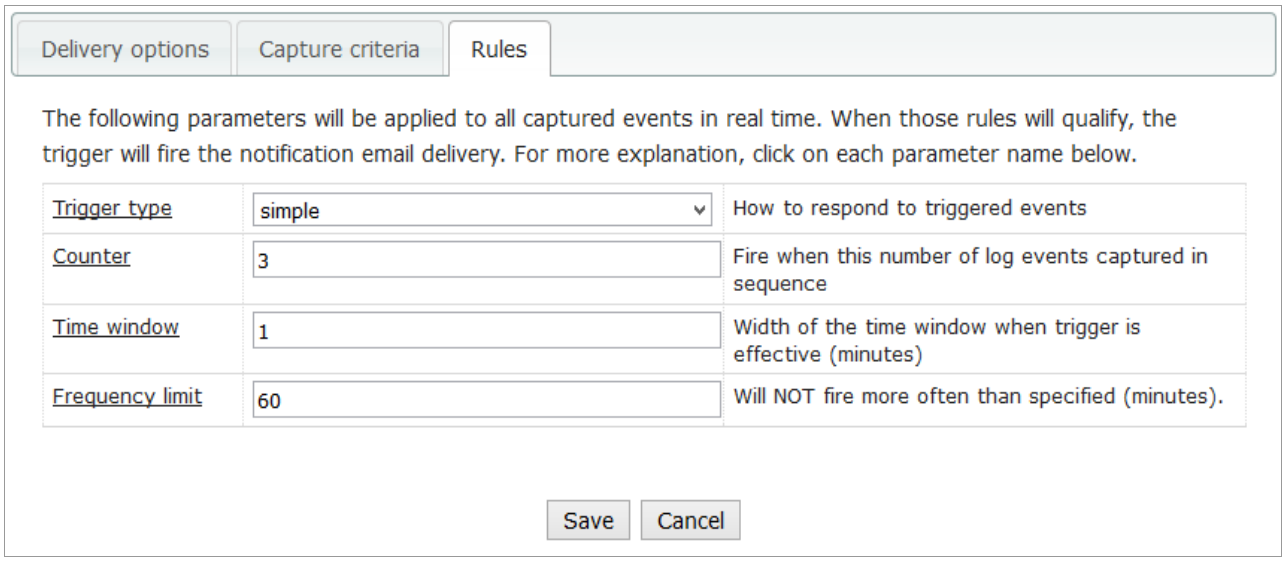


Figure 2.9.32: Trigger rules

Trigger type - depending on the trigger type, the notification behavior will vary. Simple triggers count captured events and fire when certain amount is captured within specified time window. Split triggers are doing the same by using configurable context. Silence detectors do the opposite - they fire when nothing is captured during specified period of time.

Counter - the trigger will fire only when at least this many events are trapped by the criteria.

Time window - the trigger will fire only when events are captured within this time frame (in minutes). To ignore time window and react instantly, set this value to zero.

Frequency limit - the trigger will not fire more often than specified by this value in minutes. This is used to prevent flood of email notifications in case something goes wrong.

2.9.18.4 Split triggers

Split triggers deserve special explanation and real life example. Split triggers not simply counting the occurrence of event but do so in a context you specify. Consider an example when we need to fire a trigger when certain user tries to log-in very often and we want to detect **who** is the user. Assume the following log event: "*User XXX logged in*", where XXX will change depending on a user name.

So, if we want to get notified when **particular** user comes along and **not just any user**, we want to tell the trigger to look in "message" attribute (**split by**) and extract a word from the message using the **regular expression** with group capturing: "**User %{WORD:userName} logged in**".

Named group '**userName**' in this case is called **triggering value** because trigger will fire **only** when certain amount of userName of the **same value** are detected. The same trigger may fire several notifications - each for different userName. Hence the name - **split triggers**.

To use this variable in email body or subject, simply use '**${userName}**' - it will be replaced with actual value when trigger fires. This way you can use very specific email notifications and see right away what happened. For example, email subject "*James Bond is being abusive*" is more helpful than "*There are too many log-in attempts in the past 15 minutes".*

If you need to capture stuff like that – split triggers are good, make sure to familiarize yourself with regular expressions and usage of named groups.

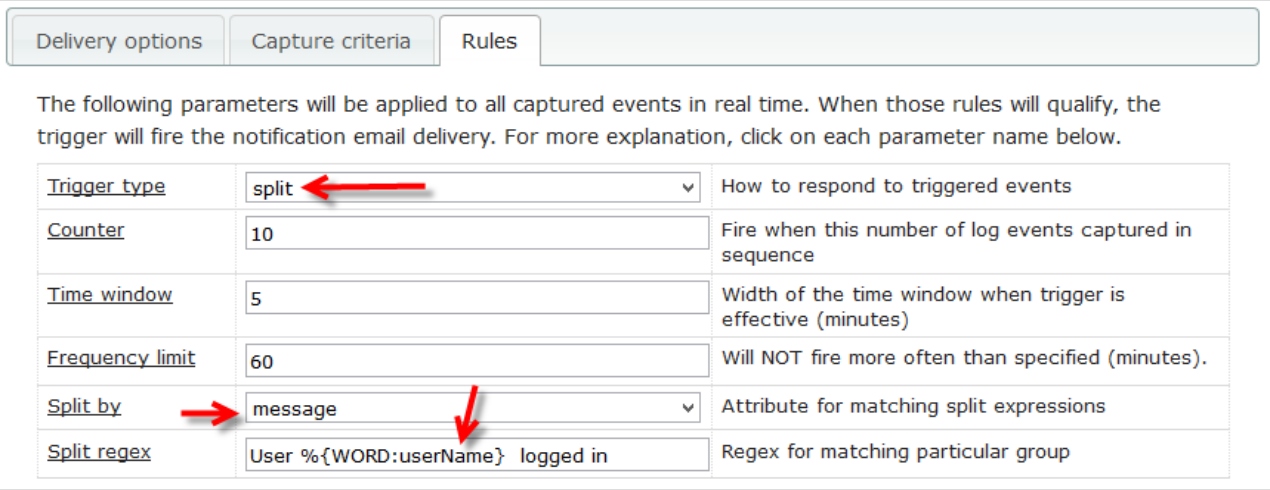


Figure 2.9.33: Split trigger example

2.9.18.5 Silence detectors

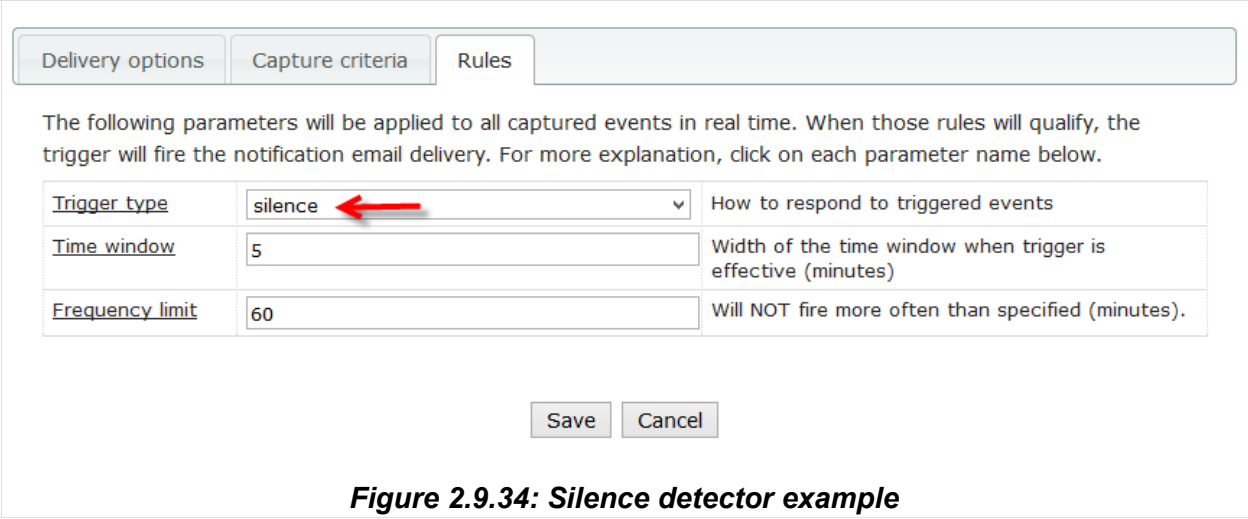


Figure 2.9.34: Silence detector example

Not very hard to guess what this kind of triggers do. Silence detectors go off when nothing is coming along during specified time window. By nothing we mean that **nothing is captured by the capture criteria** specified with this trigger. Use them when you need to detect unusual lack of activity.

2.9.19 License

License tab displays currently installed license information as well as allows you to install new license file. When you install logFaces server for a first time, it automatically activates one time trial evaluation period for 10 days. If you decide to purchase a license, the license file should be submitted through this form:

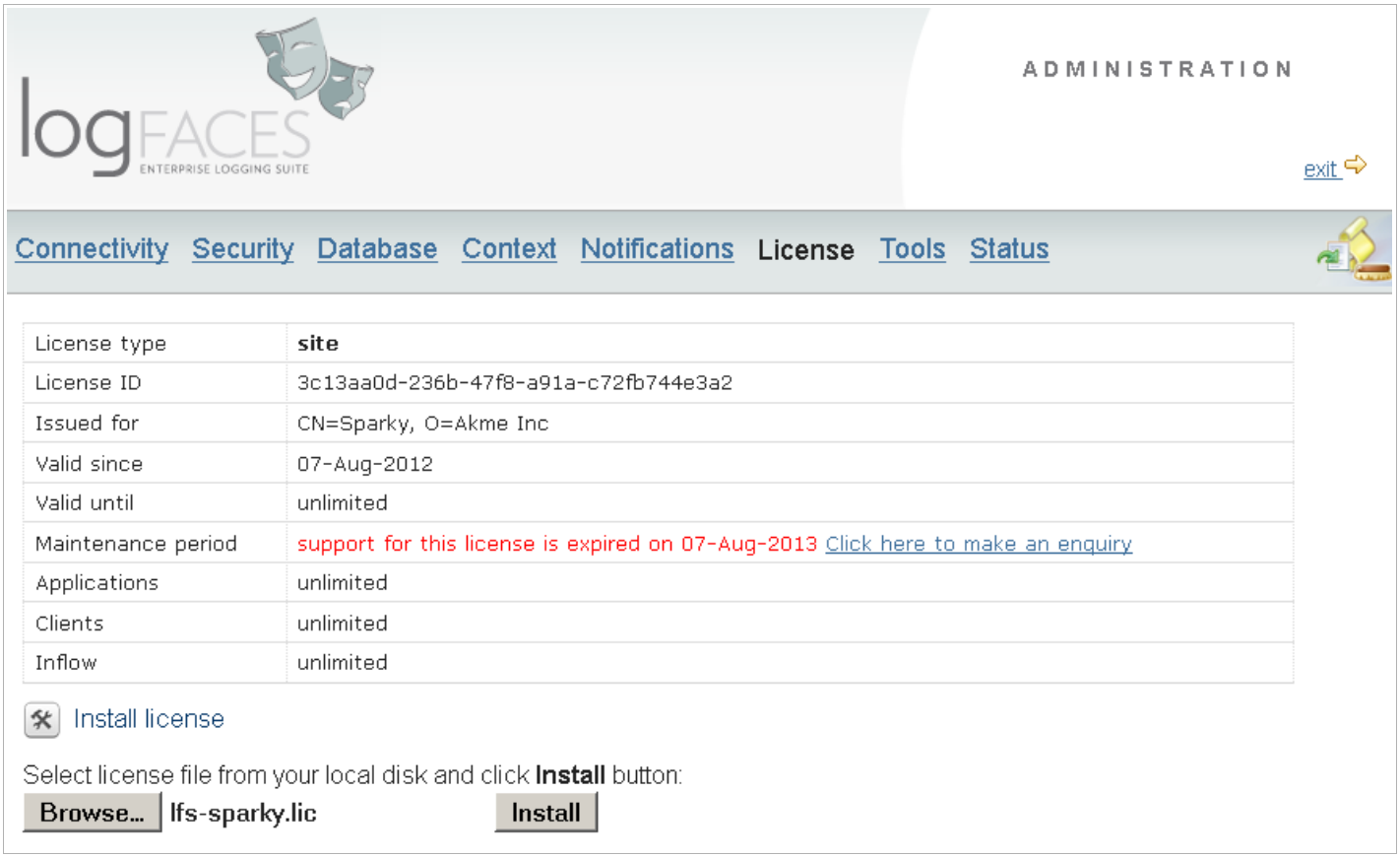


Figure 2.9.35: Licensing

What happens when **evaluation license expires**? logFaces server will shutdown its engine and only allow Administration Console access; applications will not be able to use the server and clients won't be able to connect to it.

What happens when **maintenance plan expires**? logFaces server will continue to function normally. Software updates will not be available until license is extended.

When you install new license, the engine should be started manually. This can be done by simply restarting the service from command prompt or control panel, or from the Status panel link – see the next section.

2.9.20 Status

Status tab contains useful health monitoring information of the server and allows basic instrumentation tasks such as restarting, upgrading, etc.

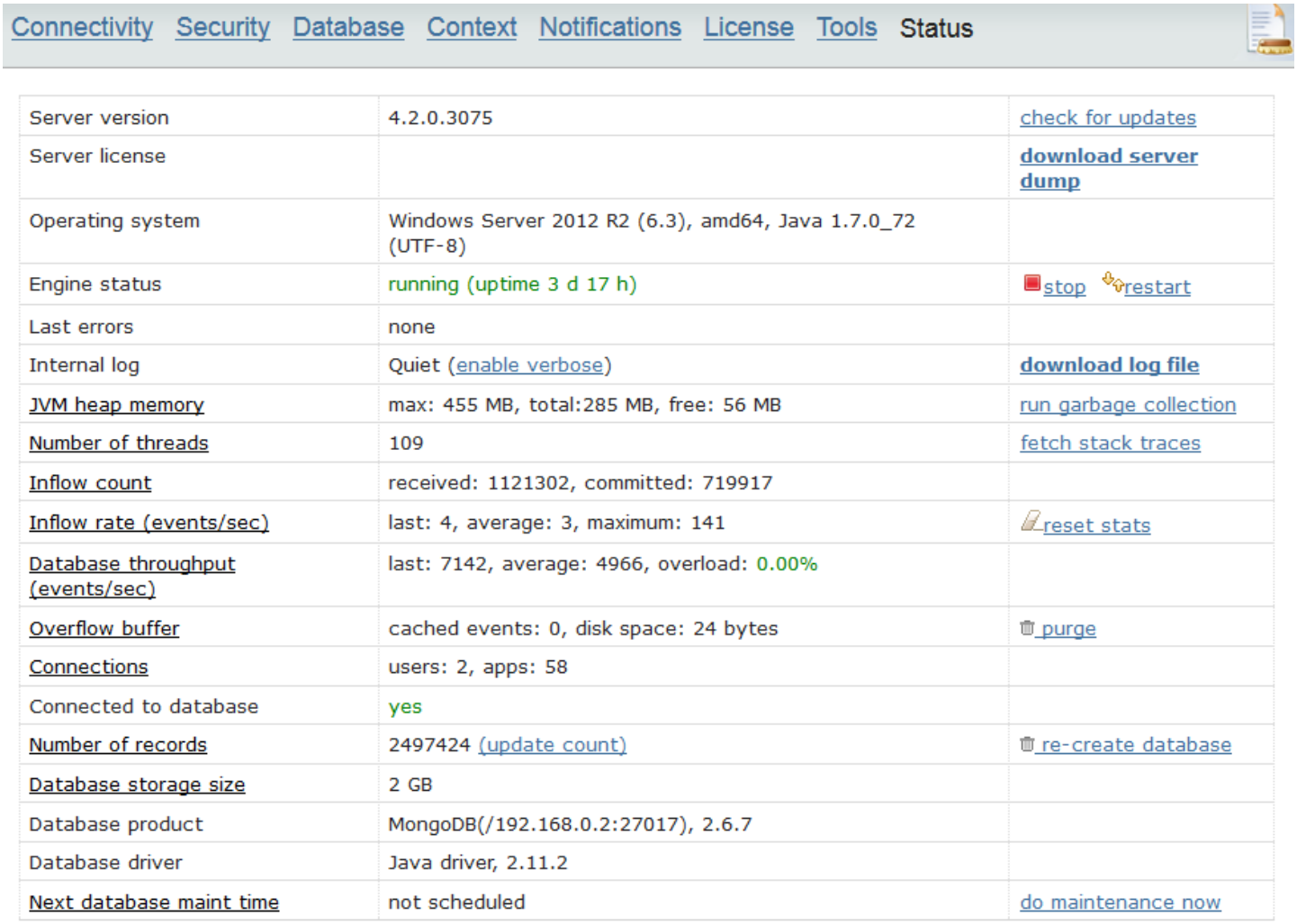


Figure 2.9.36: Server status tab

Check for updates will try to detect updates we regularly post on our web site. If update is available, the new version number and package size will be displayed. Note that this operation requires live internet connection. Updates can also be automatically downloaded and installed – when new version is detected, you will get a link to activate the installer. The process is fully automated but will take your server offline for a few moments.

Download server dump will create a zip file containing your configuration and internal log files, sending this package to our support may often speed up the support process.

Engine start/stop/restart; sometimes it's required to put the server down without actually shutting the process down. One of the typical uses of this option is when trial license expires. In such case, the logFaces Server will start so that you would be able install proper license, but its engine will be down and no logging will be taken from applications.

Run garbage collection; explicitly force garbage collection in server JVM

Fetch stack traces will download full dump of all threads currently running on server.

Last errors is a list of latest errors encountered by server, you can browse through them to see if anything went wrong lately, or simply reset them.

Internal log can be tuned to verbose or silent mode. You can also download an internal log file for inspection. This file can be sent to our support team.

Inflow rate represents the throughput of the server on the network side, it indicates the amount of logs flowing into logFaces from all appenders per second.

Database throughput indicates how much data your database can commit per second. You should keep an eye on this metric to be below **Inflow rate** most of the time. When database throughput is significantly lower than inflow rate for a long time, the data will be stored in local disk storage called **overflow buffer**. Normally this is an expensive operation and may result in higher than usual CPU ans IO use. **Overload** is the percentage ratio of total number of events went through an overflow buffer on local disk to a total committed. This ratio is very important for detecting the database bottle neck. When overload gets too high, it will be emphasized in red color. The default threshold is set to 10% but you can adjust it in environment properties (see paragraph on advanced configuration). You will also see a flag icon indicating that currently server handles its internal overflow cache trying to push it into the database.

Overflow buffer shows how many events are currently pending to be committed due to overload and how large is currently allocated space for overflow mechanism. Overflow is a mechanism designed to guard against inflow spikes and prevent data loss when logs can't be committed to your database. This mechanism gets engaged when database is unavailable, or database is slower than the inflow. Your database may be super capable but when massive spike of inflow takes place, we will buffer the impact to prevent major disruption. When this happens, the overflow mechanism will delegate incoming data to a temporal local storage and then try flushing it whenever database permits. Note that overflow buffer is limited by number of log events it can hold. You specify this in environment configuration file, default is 500K. When this number crossed, logFaces will start loosing data as

unsustainable. Whenever new space disk space is required, the overflow directory gets allocated by 32MB or bigger chunks. Note that total allocated disk space is not returned back to operating system unless explicitly requested to do so. You can manually do it here by clicking on a **Purge** link, make sure that actual cache is empty when you release the disk space.

Number of connection shows how many clients are using the server now and how many TCP appenders are currently working.

Re-create database allows to remove all database records; be careful with this operation, it is not recoverable and can't be undone. Some SQL databases may require special statements for dropping tables, for example Oracle may use recycle bins which prevent releasing the storage space. To customize the dropping sequence we have `/conf/lfs-drop.sql` file where you can specify SQL statements applicable to your database. It must contain one statement per line.

Update records counter will re-count total number of log records stored in your database. Because counting with some databases is very expensive operation, this action is set for explicit user request.

Database maintenance – will manually start database maintenance job whether it is currently scheduled or not.

2.9.21 Getting server info remotely

It is possible to obtain basic server information by doing HTTP call to the following URL:

```
http://your-server-host:8050/rest?method=version
```

Server will respond with its version information in a form of JSON like this:

```
{
  "serverVersion":"4.2.0.3075",
  "osVersion":"Windows Server 2012 R2 (6.3), amd64",
  "javaVersion":"1.7.0_72",
  "dbProduct":"MongoDB(/192.168.0.2:27017)",
  "dbDriverVersion":"2.11.2",
  "dbDriverName":"Java driver",
  "dbVersion":"2.6.7"
}
```

2.10 Advanced settings

There are some settings which can not be configured through the administration web interface but can be configured manually in several configuration files. These settings should only be modified when server is down. Please review the settings carefully before doing any manual change. It's a good idea to keep a backup of the files you are intending to modify.

2.10.1 Server bootstrapping

logFaces server runs in conventional JVM on any operating system with JRE 1.6+ installed. This JVM is wrapped up and monitored by special third party component called Java Service Wrapper. You would never run logFaces in raw JVM, the installation comes with OS specific dependencies which are responsible to run, host and monitor the JVM which runs actual logFaces server software. The most important task of the Wrapper is to provide robust error recovery in case of JVM or hosted software failures. In such cases Wrapper will automatically restart hosted JVM.

Wrapper also provides simple yet powerful bootstrapping configuration for the JVM it hosts. In most cases user doesn't need to know these details, but when it comes to more complicated settings, it is good idea to familiarize yourself with how Wrapper works. Wrapper files are located in `/bin` directory under server installation, this includes OS specific binaries, licenses and configuration files.

The JVM bootstrapping parameters can be found in `/bin/lfs.conf` file, it specifies which JVM to run, how to setup its classpath, what arguments to pass to JVM and how wrapper should manage this JVM when it runs. Refer to wrapper documentation for more details, but from the logFaces point of view, there are only a handful parameters which can be modified:

Wrapper property	Description
`wrapper.java.command`	Full path to JRE java executable, this is the command wrapper will run to start the JVM instance.
`wrapper.java.initmemory`	Corresponds to JVM -Xms argument - minimum or initial memory allowed to be allocated by JVM in MB.
`wrapper.java.maxmemory`	Corresponds to JVM -Xmx argument - maximum memory allowed to be allocated by JVM in MB. Raise this value if your logFaces instance needs more RAM.
`wrapper.java.additional.x`	Can be used to pass additional JVM arguments as -D
`wrapper.app.parameter.1`	An argument passed to logFaces server from command line. This argument must point to logFaces home directory.

2.10.2 Server home

When logFaces server gets bootstrapped, it expects a single argument of its home - the directory where it expects to find its configuration files, licenses and output its runtime artifacts. By default the home directory matches the installation directory, but sometimes it is useful to separate the installation and home directories for deployment management purposes. For example, if you are running several instances of logFaces server and want to manage their settings in one place. Or if you want to swap one set of configurations to the other without too much hassle with admin interface. Or when you run logFaces in virtualized environment.

To setup a non-default home directory look into a bootstrapping wrapper configuration file named `/bin/lfs.conf`. A parameter named `wrapper.app.parameter.1` specifies the argument fed into logFaces JVM during start up. This argument is the home directory where logFaces engine will look for its configuration, license and any other setting. It will also produce all files such as logs relative to the home directory. When home directory is separated from the installation tree, it makes it easier for backups, upgrades, or fast swapping homes during tests.

Note that if home directory is empty during server start up, it will get populated with default set of configuration files which you can further modify.

It is possible to pass environment variables into wrapper configuration file, for example:

```
wrapper.app.parameter.1 = %LFS_HOME%
```

will try to resolve LFS_HOME environment variable and pass it to the server. If server will not be able to resolve this argument into workable home directory, it will fall back to its default home - the installation directory.

2.10.3 Server environment

During server start up the properties from `conf/environment.properties` file are fed into server JVM. The table below gives a detailed description of all those properties. Append `com.moonlit.logfaces` prefix to all names below. Note that changes to this properties will take affect only when server gets restarted. make sure that you modify them when server is not running.

Property	Mandatory	Description
.config.server	yes	Reference to the main configuration file
config.ro	no	When set to **true**, server instance will run in immutable mode where configuration can not be altered
.config.storage	no	Type of data storage to use. Values allowed: **sql**, **mongodb** and **bigquery** Default is sql
.config.mongoprops	no	Reference to internal mongodb configuration file, default is ${lfs.home}/conf/mongodb.properties
.config.hibernate	no(*)	Reference to hibernate configuration file. (*) Mandatory when using SQL database.
.config.schema	no(*)	Reference to schema file which will be created in database. (*)Mandatory when using SQL database.
.config.jobs	yes	Reference to jobs configuration file.
.resources.eventMapping	no	Custom mapping of hibernate event data. Applicable when using SQL databases.
.resources.repoMapping	no	Custom mapping of hibernate repository data. Applicable when using SQL databases.
.resources.mongoIdGenerator	no	Custom implementation of MongoDB ID generator. Applicable when using MongoDB only.
.url.revision	no	URL for checking software updates
.url.downloads	no	URL for update downloads
.url.updates	no	URL to version update descriptor
.url.notes	no	URL to version release notes
.url.support	no	URL to support site
.monitoring.highThreadCount	no	Maximum number of threads the server should be able to sustain,

		higher number will issue an internal warning.
.monitoring.lowMemoryThreshold	no	Minimum of free heap memory specified as a percentage of maximum heap memory. When free memory will go below this value, the server will issue a warning which will appear in Admin. console. Default is 2.
.monitoring.overloadThreshold	no	Threshold for detecting database overload – ratio between overflown and total commits. Default is 10%.
.monitoring.overflowSize	no	Size of the local overflow queue. When reached, the overflow queue will refuse to queue more events and will raise system error. Default is 500000 events.
.monitoring.overflowDisk	no	Maximum disk space in MB allowed for the overflow mechanism. Default is 10240 (e.g. 10GB). Server will not allocate more space for the overflow mechanism than specified by this parameter until purged by user or automatically after the idle timeout specified below.
.monitoring.overflowIdleTimeout	no	The idle timeout in minutes for purging overflow disk space allocation. The space will be returned to OS if overflow is idle for the specified time. Default is 15 minutes.
.dzone.crcCacheSize	no	Maximum size of CRC cache storage used by drop zones
.security.keyStore	no	File name of the key store
.security.trustStore	no	File name of the trust store
.security.keyPass	no	Password for key store (plain or obfuscated)
.security.trustPass	no	Password for trust store (plain or obfuscated)

.security.sni		Enable or disable SNI (Server Name Identification for https calls)
so.rcvbuf	no	Sockets receive buffer size in bytes, equivalent to SO_RCVBUF option for server sockets. Default 65535.
so.maxSize	no	Maximum size in bytes of a single log event expected by socket receivers. Default 262144 (256KB).
http.host	no	When specified, the server will use this name to respond to HTTP requests which **don't** carry **Host** header. If not specified, the 'localhost' will be used. This option is mostly to prevent vulnerabilities in HTTP v1.0 which may leak internal IP when server is hosted behind NAT.

2.10.4 How do I work with external SQL databases?

You have to obtain relevant database driver from your database vendor and place the jars in `/lib/dbdrivers` directory on server. **Our installation only include drivers which permitted by publisher's license**. We do our tests for Oracle, MySQL, SQL Server, DB2 and PostgreSQL, but theoretically there shouldn't be a problem to work with other relational databases as well. It's only a matter of configuration and database driver you wish to use.

Look in `/conf/environment.properties` file – you should see these two properties referencing hibernate configuration file and database schema :

```
..config.hibernate=${lfs.home}/conf/hibernate.properties
..config.schema=${lfs.home}/conf/lfs.sql
```

You can modify these references by pointing to different files, but make sure the files are correct. Our distribution contains both hibernate templates and schema for all databases we support at this moment. The example above will use embedded database driver settings and Derby schema. If, for example, you would like to use MySQL, do the following:

```
..config.hibernate=${lfs.home}/conf/hibernate-mysql.properties
..config.schema=${lfs.home}/conf/lfs-mysql.sql
```

Then modify hibernate properties file to point to your database. logFaces server uses Hibernate ORM framework; it is recommended to have some knowledge of this framework before you decide to make re-configuration.

Once configuration is ready, restart logFaces server. New schema will be published automatically. After the start up, open **Administration Console** and navigate to the **Status** tab. If everything went well, you should see that engine is started, database connection is OK and there are versions of database and driver used. If something goes wrong, you will see red marks and error indications. To see what happened, click on "show last errors" or "download log file" link; the problem is usually related to a configuration error, typo or perhaps your database is not responding as logFaces expects. If you're unable to figure out the problem yourself, submit this log file or an error code to our support site and we will try to help. Another, perhaps easier, option is to run the server in console mode and see that there are no exception thrown during start up. On Windows you can run server in console mode from `/bin/run.bat` on. On Linux do `./lfs console` while in `/bin` directory.

2.10.5 Can I modify SQL database schema?

Yes, you can do that to a certain extent. Database schema file is referenced in `/conf/environment.properties` with `com.moonlit.logfaces.config.schema`.

You can not modify the structure of the tables or column names because they're mapped through hibernate mapping in the code. But you can adjust column size constraints, modify indexes or add some additional statements as long as they don't break the mapping.

After you changed the schema file you need to re-create the database. This can be done either from administration console status tab, or by using some other external tool. Note that when using embedded database, there is no other choice but using the admin. console.

IMPORTANT: If you care for the existing data in the database, make sure to back it up before doing any changes to the schema. One of the options is to use our backup utility described next.

2.10.6 Can I use my own PK generators with SQL databases?

Yes. By default PKs are generated with simple native generator defined in hibernate mapping files which are not exposed for general usage. If you want to generate your own PKs, the mappings need to be altered and server needs to know which mapping to use. The procedure to follow:

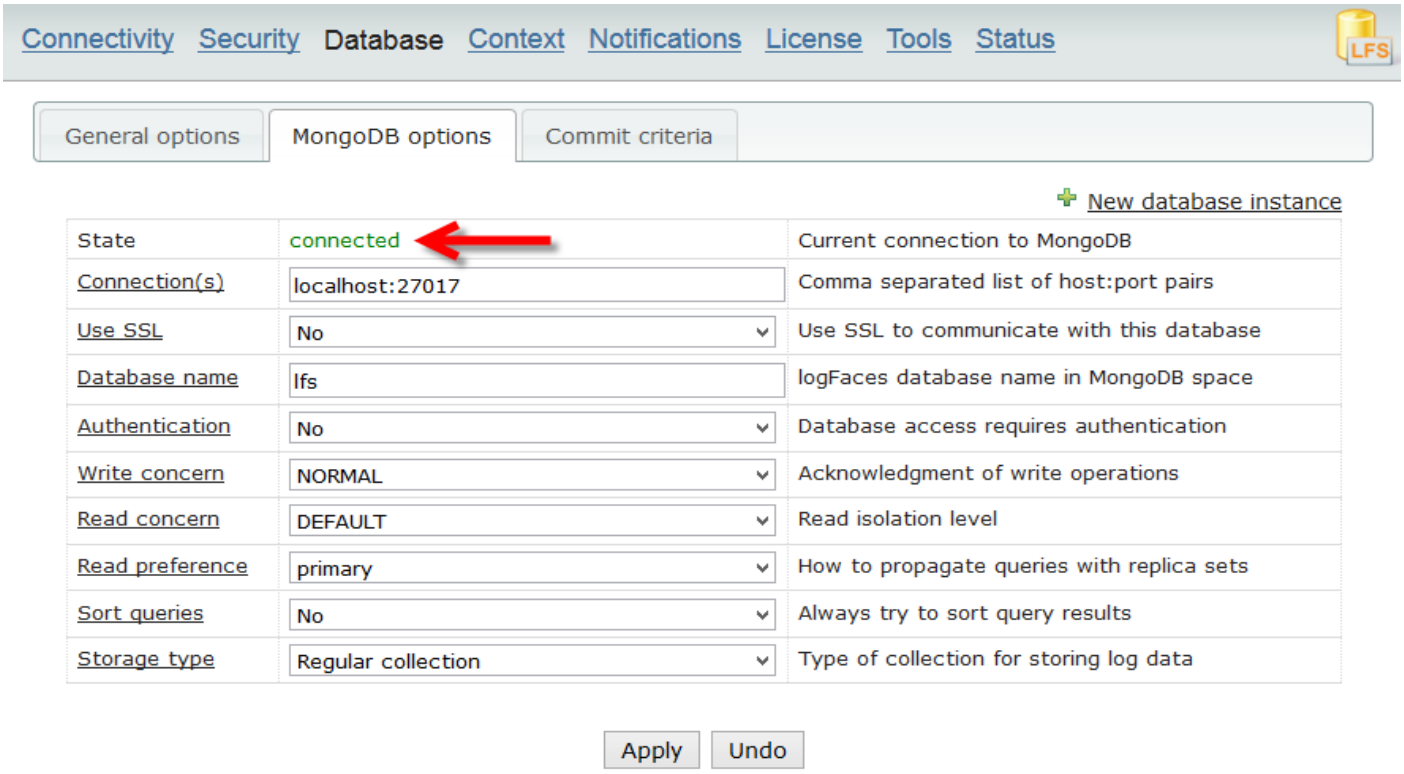
1. extract Event.hbm.xml and Repository.hbm.xml from /lib/lfs.jar
2. modify the 'id' column generator section as you need, this is standard hibernate format
3. create separate jar file and place modified mappings along with your own classes
4. place this jar under /lib directory on server
5. enforce those classpath resources by changing /conf/environment.properties:
   ```
   com.moonlit.logfaces.resources.eventMapping = classpath:/your/Event.hbm.xml
   com.moonlit.logfaces.resources.repoMapping  = classpath:/your/Repository.hbm.xml
   ```
6. run server in console mode (/bin/lfs.bat) and watch the log output. If there is something wrong in the mapping or something is missing, the database layer will not start and there will be many errors.

2.10.7 How do I work with MongoDB?

logFaces can work with traditional RDBMS as well as with NoSQL databases. Our choice for NoSQL database fell on MongoDB and in this section you will learn how to set it up. First - specify the storage type in `/conf/environment.properties` file like this:

```
com.moonlit.logfaces.config.storage = mongodb
```

Next step – get your MongoDB running, start logFaces server and go to its admin/status page. Most likely that logFaces won't find your database right away, so you will see that database is down. This is OK. Go to admin database tab and adjust MongoDB parameters, most importantly the connection end points – host, port and security options if relevant for you database. At this point you should see something like this:



It means that logFaces is now connected to MongoDB daemon on local machine, created its database named `lfs` and ready to work. To verify – open MongoDB console and make sure that requested database is indeed created and it has two collections there - `log` and `repo`. You can always get back to this page later and adjust the settings. In most cases the changes applied instantly.

Note that parameters on this page come from and saved to `/conf/mongodb.properties` file on your server. You will hardly ever need to edit them manually, but it is a good idea to understand

them. Table below is a compete list of properties we use to integrate with MongoDB, it will help you to understand how to use connections, replica sets, indexes and other important features.

Property name	Description	Default
.connection	Comma separated list of host:port pairs for specifying replica sets.	localhost:27017, localhost:27018, localhost:27019
.ssl	When database requires SSL connection this property must be set to true. If your database uses well known root CA, it may work right away, otherwise your certificate and key must be imported into the local trust store as described here.	false
.invalidHostAllowed	What to do when host name of the certificate doesn't match server host name	false
.user	User name for authentication. Leave blank if your database doesn't require security.	
.password	Password for authentication. Leave blank if your database doesn't require security.	
.userdb	When database is secured, this parameter specifies the name of the database where users for authentication are defined.	
.writeConcern	Controls the acknowledgment of write operations with various options. See MongoDB documentation for more details.	NORMAL
.readConcern	Applicable since MongoDB 3.2, allows clients to choose a level of isolation for their reads. See MongoDB documentation for more details.	LOCAL
.readPref	Read preference describes how MongoDB clients route read operations to members of a replica set. See MongoDB documentation for more details.	primary
.dbname	Name of the database logFaces will create	logfaces

.capped	When set to 'true' logFaces will force the capped 'log' collection and will try to convert it to capped when it's not, and will fail if it can't be converted.	false
.cappedSize	Size of capped collection in MB. Affective when 'capped' is set to true	100 MB
.ttl	When set to true logFaces will try to convert the collection to TTL unless it's already TTL.	false
.ttlDays	Number of days for TTL collection. Affective only when TTL collection is enabled	7
.partitioned	When 'true' logFaces will convert its storage into set of partitioned databases.	false
.pdays	When partitioned this parameter specifies the size of each partition in days	1

Note that it is possible to parametrize property values like this:

```
com.moonlit.logfaces.config.mongodb.connection = ${my.host}
```

Server will then looks for JVM system property named `my.host` to resolve the value. Such properties must be injected into server JVM during the bootstrap and should be available during server initialization. Parametrized properties will not be persisted when modified from admin web interface.

2.10.8 MongoDB schema and indexes

When started, the server will automatically create two collections named `log` and `repo`. Log collection stores log events in such way that each event corresponds to exactly one MongoDB document. This collection grows rapidly as data gets committed. Repo collection is a helping repository to keep names of applications, hosts and loggers – this information is used mostly by clients to assist with selection of items for queries and hardly ever grows once the system stabilizes.

Documents stored in **log** collection have attributes corresponding to the fields of actual logging event. In order to save storage space those names are shrank to a minimum, so when you look into collection, you will see extremely short names. Below is the mapping of these names:

```
{
  "_id" : ObjectId("51abeff2e0fd7bab0f2cf20c"),  // object ID (generated by mongodb)
  "t" : ISODate("2013-06-03T01:22:12.794Z"),     // loggerTimeStamp (creation time)
  "q" : NumberLong(3256236),                     // sequenceNumber (generated by lfs)
  "p" : 5000,                                    // loggerLevel (severity level)
  "r" : "355941742@qtp-2017211435-7749",         // threadName ( originating thread name)
  "m" : "This is the message",                   // message (message body)
  "h" : "my host",                               // hostName (originating host name)
  "a" : "my app",                                // domainName (originating app name)
  "w" : false,                                   // thrown (exception true/false)
  "g" : "com.myapp.logger",                      // loggerName (originating logger name)
  "f" : "Myclass.java",                          // locFileName (location file name)
  "e" : "MyMethod",                              // locMethodName (location method name)
  "l" : "489",                                   // locLineNumber (location line)
  "c" : "com.myapp.logger",                      // locClassName (location class)
  "p_targetID" : "1360911352",                   // custom MDC mapped to targetID
  "p_sessionID" : "sid4454.col080"               // custom MDC mapped to sessionID
}
```

Because logFaces allows usage of any of the above attributes in queries, you will need to carefully select compound indexes. One important thing to keep in mind when dealing with compound indexes is that the **order of the attributes** in index is crucial to index performance. For example indexes (t,p) and (p,t) may have completely different performance depending on the data store. First one will seek records in specified time range (t), and then reduce to severity level (p). Second one works in reverse – first seeking level (p) and then reduce by time (t). It is often desirable to have several compound indexes so that you cover as many frequently used queries as possible. Keep in mind that the aim of selecting right indexes is to **compromise** between storage size (indexes are greedy) and the end user satisfaction with the speed of queries. Use MongoDB console tools to examine performance details.

Nearly all queries generated by clients contain `loggerTimeStamp` (t) attribute. It means that in most cases you will want to include this attribute and position it correctly within the index.

2.10.9 MongoDB capped collections

From MongoDB site:

“Capped collections are fixed sized collections that have a very high performance auto-FIFO age-out feature (age out is based on insertion order). They are a bit like the "RRD" (Round Robin Database) concept if you are familiar with that. In addition, capped collections automatically, with high performance, maintain insertion order for the objects in the collection; this is very powerful for certain use cases such as logging.”

RRD doesn't ensure data storage by time (for example 3 days of data), it ensures that storage will not grow bigger in size than specified. However, having such tremendous performance advantage, capped collections could be a preference for some. If you are one of those, this is how you should setup logFaces to utilize this feature – go to admin/database/MongoDB tab, check Capped Collection option and specify size in MB. The server will automatically do the relevant conversion of 'log' collection into capped. It is also possible to convert to a capped collection directly from MongoDB shell. If you do this, make sure to modify `mongodb.properties` file accordingly before restarting the server!

When capped collections used, the database day capacity automatically becomes `unlimited` and logFaces will not manage the data store size. Maintenance of such database is also irrelevant since MongoDB storage size will not grow from the specified. Thus you will not see `capacity` and `maintenance cron` options in admin database page.

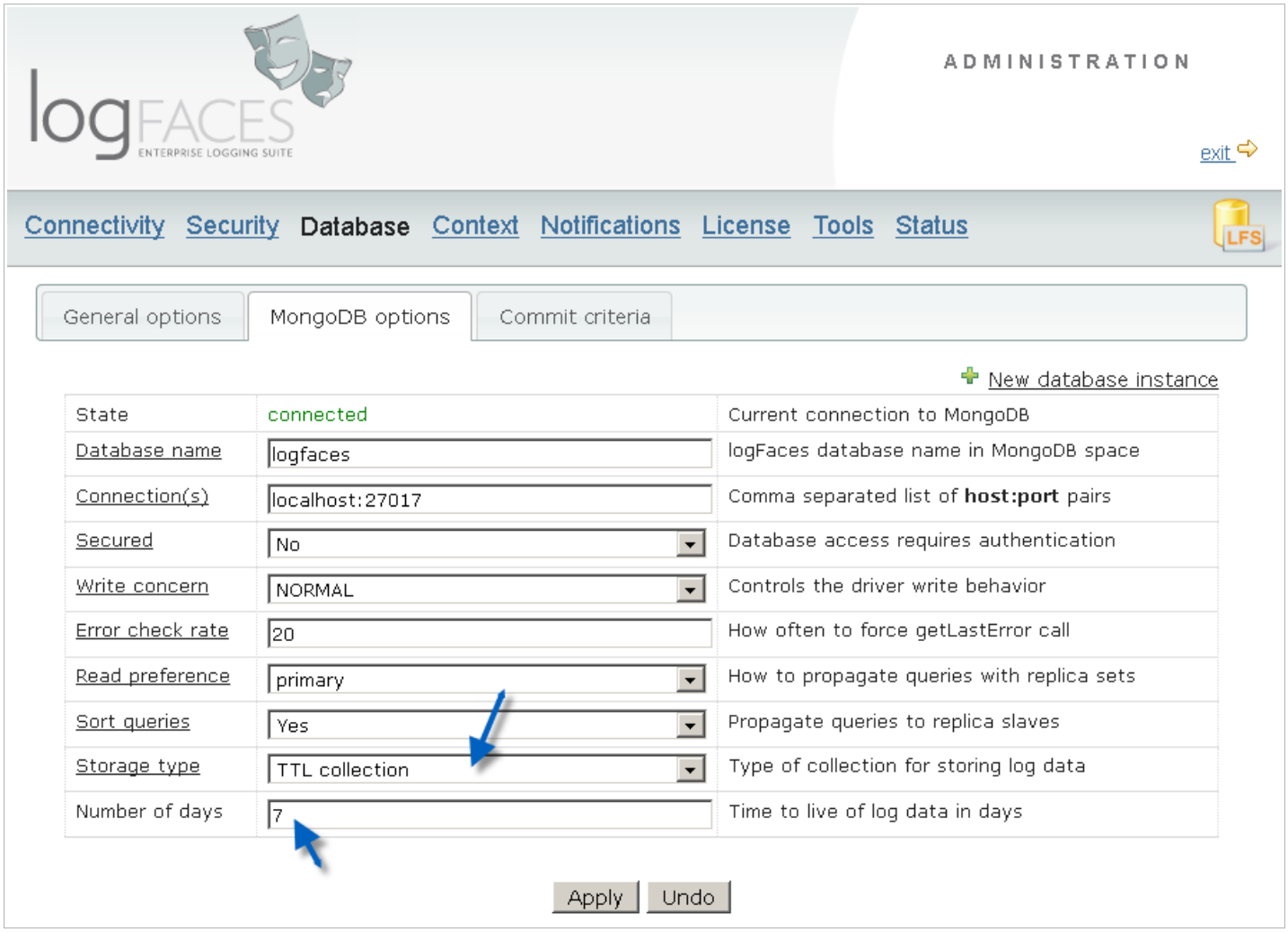
Note that switching from capped collections back to regular collections is not possible without manually backing up your data and re-inserting it back into a newly created regular collection. It is also not possible (as of this writing) to extend the size of capped collections without doing manual re-inserts. Again, make sure to modify `mongodb.properties` file before restarting the server, otherwise it will convert everything back to capped. In other words, `mongodb.properties` must always be adjusted manually **if you manually administer your database** otherwise it may override your changes.

2.10.10 MongoDB TTL collections

TTL stands for “Time To Live”. From MongoDB documentation:

“*Implemented as a special index type, TTL collections make it possible to store data in MongoDB and have the* `mongod` *automatically remove data after a specified period of time. This is ideal for some types of information like machine generated event data, logs, and session information that only need to persist in a database for a limited period of time* ”

To enable this feature with logFaces server, open administration page and modify “**storage type**” option to “**TTL collection**” as shown below. You will also need to specify the “**number of days**” for this collection to operate with.



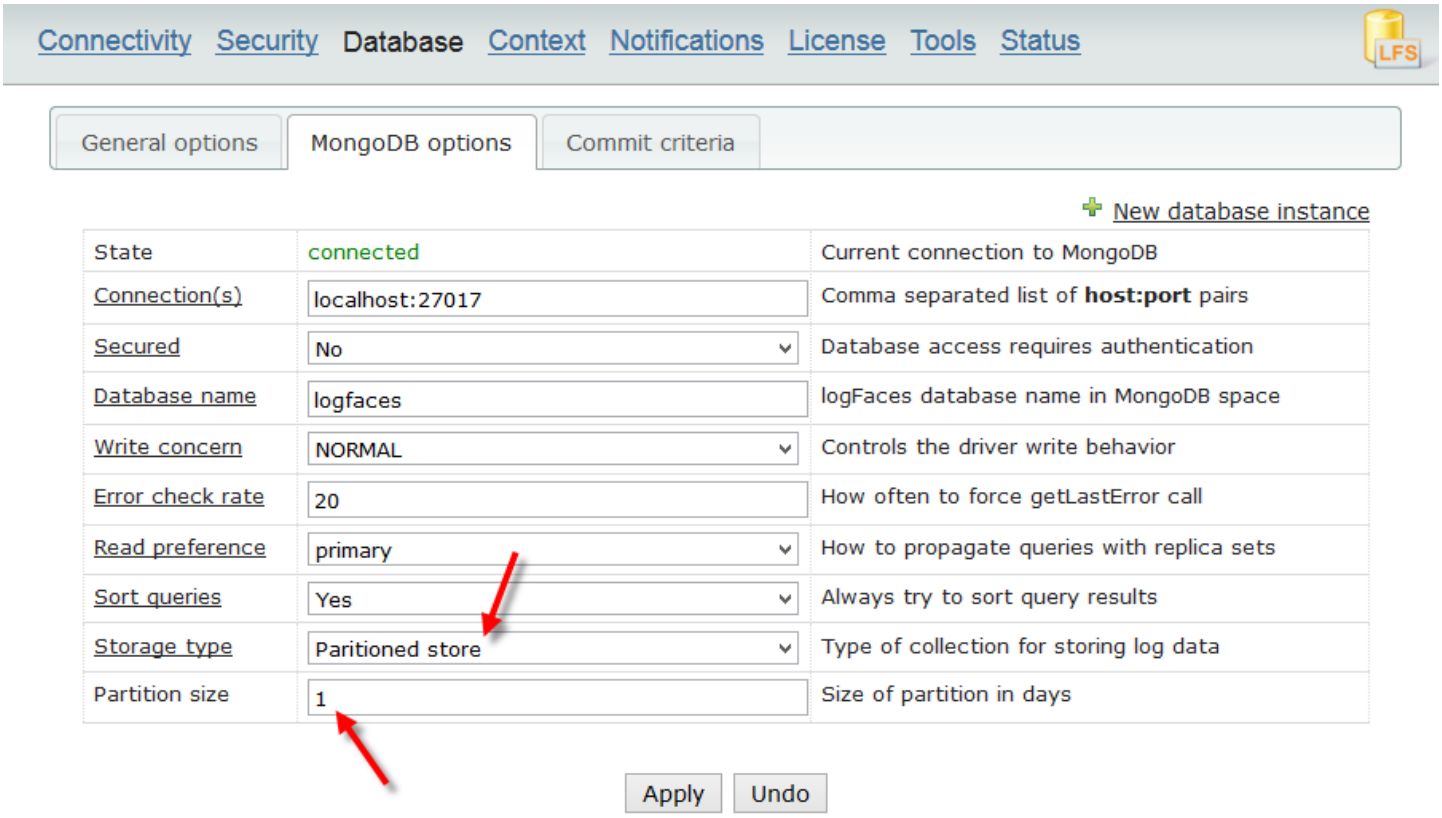
Note that when TTL collection is used, the server will automatically set `Retention` to unlimited and will not remove older records from database on its own – this duty will solely be in hands of MongoDB.

2.10.11 MongoDB partitioned storage

There is a common problem with data storage of any kind – it's much easier to store things than to retrieve them, yet more difficult to delete them. The problem intensifies as storage gets bigger.

To address this problem we have an alternative way for storing very large amounts of log data. Instead of using single collection which is a default setup, you can choose to partition your entire data set by days, or as we call them - *partitions.*

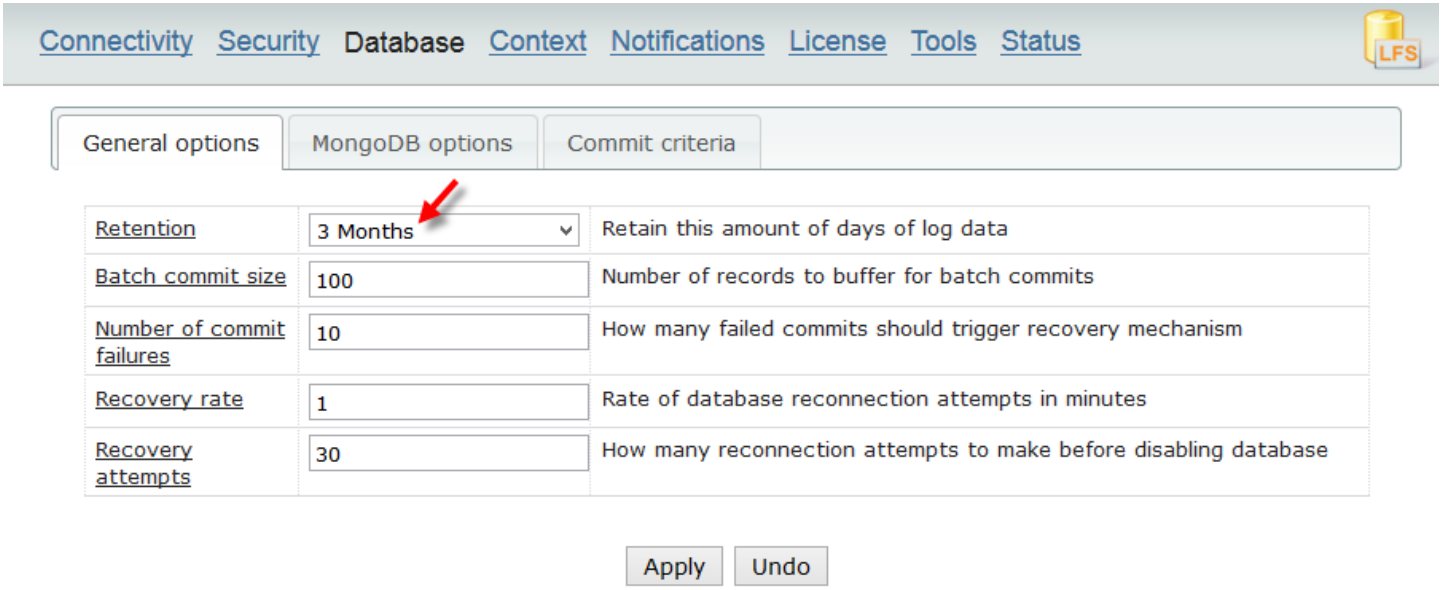
Using partitions is radically different from other types of collections because instead of one database with two collections, your MongoDB instance will have many databases - one per each partition. In other words, we partition the data **by database**, not by collection. And each database will contain exactly one partition. This decision is motivated by performance – it is much easier to drop an entire database than to drop a collection. It gets very noticeable when we deal with terabytes of data and billions of log records.



It is important to decide what would be the size of partitions in days. This greatly depends on your usage patterns. But the tradeoff is based on a very simple rule – the smaller the database the better it will perform in terms of queries. So, how much data you expect daily and how you intend to query this data should be taken into account.

When doing queries from partitioned store, logFaces server may hit more than one partition, depending on the time range user asks for. For example if query spans 3 days and your partitions are sized by 1 day, there will be 3 database hits on the server side. It may sound that things will go three times slower but this is not true. When dealing with billions of log events, it's much faster to hit few smaller collections few times than to hit one big collection once. But you still should consider the partition size, if going after several days of data is something you will be doing very often – consider to span partitions for several days. Keep in mind that once the store is partitioned, it can not be modified.

Once you switch to partitioned storage type, next thing you want to do is to adjust the `retention` – number of days to keep your logs for:



One last thing is indexes. They need need special mentioning. Each partition contains one collection named 'log' where actual data is stored for the given time range. When server allocates new partitions, **it will automatically build indexes for the new partition but it takes indexes information from its previous partition**. So, make sure to select and define your indexes for the current day as soon as you create partitioned store as it will start rolling day after day and doesn't require your attention.

Note that names of databases in MongoDB instance (logFaces partitions) follow strict format, avoid changing them or dropping them manually, this may confuse the server. If you list databases in MongoDB node you will find logFaces partitions named like **lfsp-020414-030414** where numbers denote the start and the end of the partition date, in this case the partition spans from Apr 2 until the end of Apr 3 of year 2014.

2.10.12 MongoDB custom _id creation

By default, logFaces uses the _id generated by MongoDB driver, which is a standard ObjectId. To override this behavior and create your own _id object follow the procedure below:

1. Implement `com.moonlit.logfaces.server.core.OIDGenerator` interface. The interface definition and other dependencies can be found in /lib/lfs-core.jar. Implemented class will be instantiated by reflection, so the default constructor must be present.

2. Place your implementation on server under /lib directory.

3. Tell the server to use your implementation instead of the defaults by changing this property in /conf/environment.properties:
   `com.moonlit.logfaces.resources.mongoIdGenerator = yourGenerator`

4. Run the server, feed some data and see that your implementation works as expected. If server fails to find or instantiate your implementation, it will fall back to the defaults silently.

2.10.13 MongoDB authentication

If your database instance requires authentication for its clients then logFaces server, being the client in this case, must be configured to meet this requirement. This is done by specifying user name, password and the name of the database where this user is defined in MongoDB instance.

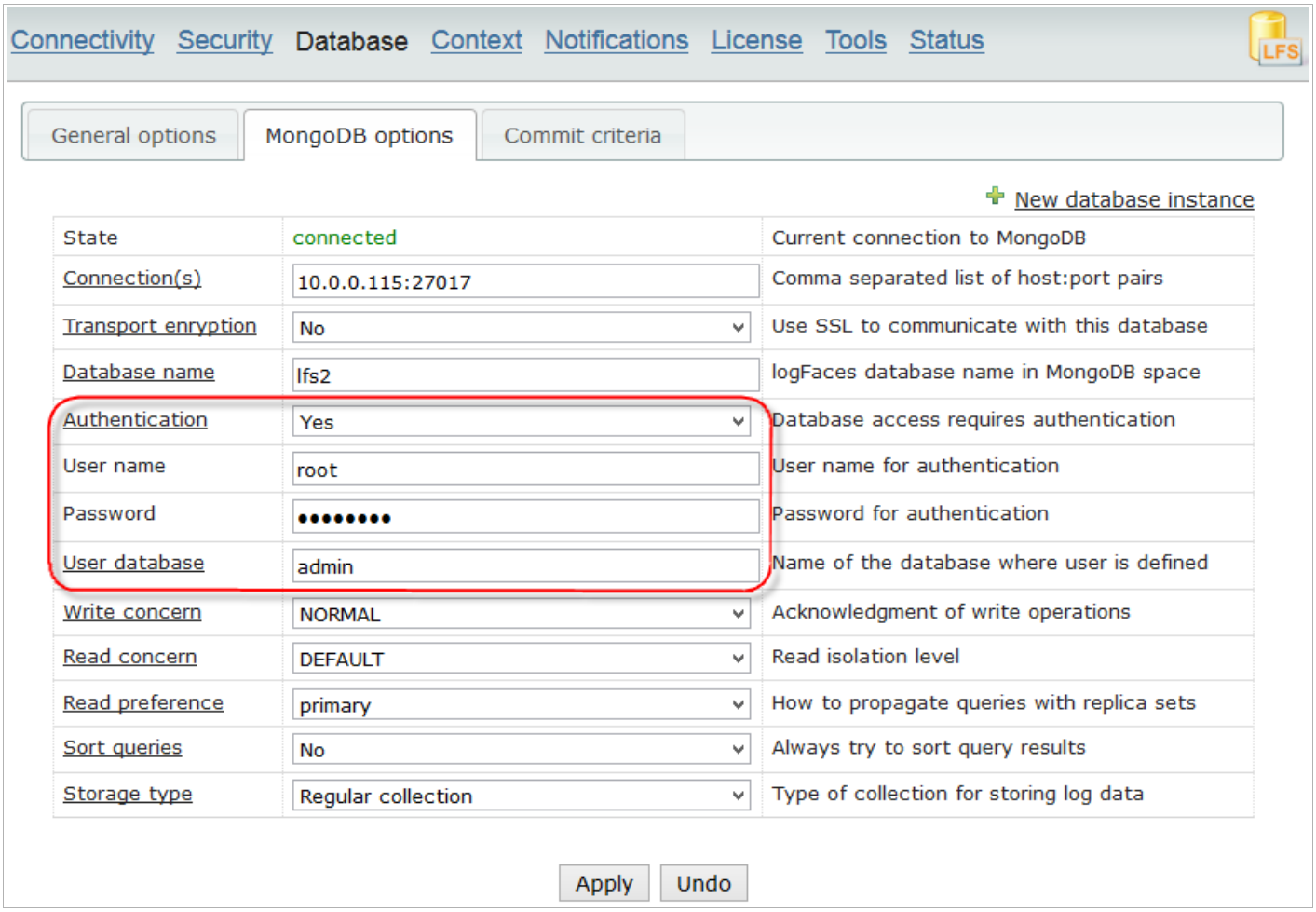


Figure 2.10.1: Setting MongoDB authentication

Starting from MongoDB v3.0 this principal must have the following permission for logFaces server to be able to work with this instance of MongoDB:

- readWriteAnyDatabase
- userAdminAnyDatabase
- dbAdminAnyDatabase
- clusterAdmin

Note that *AnyDatabase* permissions are only required when logFaces is configured to use partitioned

storage. This is because each partition corresponds to a separate database and its name is always changing depending on the time range it covers. As of this writing MongoDB roles can be assigned either fixed database names or use *AnyDatabase* sets of permissions. When partitioned store is not used, it is possible to use corresponding permissions for a particular databases name (in the example above this would be `lfs2`).

2.10.14 How do I work with Google BigQuery?

logFaces can use Google BigQuery cloud storage for the back end. This section describes how this can done presuming that you already have Google Cloud account and BigQuery service enabled and ready to use.

Firstly, we specify that the data storage we want to use with the logFaces instance is the BigQuery. Open `/conf/environment.properties` file and modify the the following property:

```
com.moonlit.logfaces.config.storage = bigquery
```

From now on, the logFaces server acts as a BiqQuery client but it still needs to know where the storage is, which account to use, which project to use and how to authenticate itself with the cloud. We use standard the Google API to integrate with the service, logFaces installation comes with all the dependencies needed for the task. The setup is finalized in `admin/database` section where we integrate logFaces with your BigQuery account:

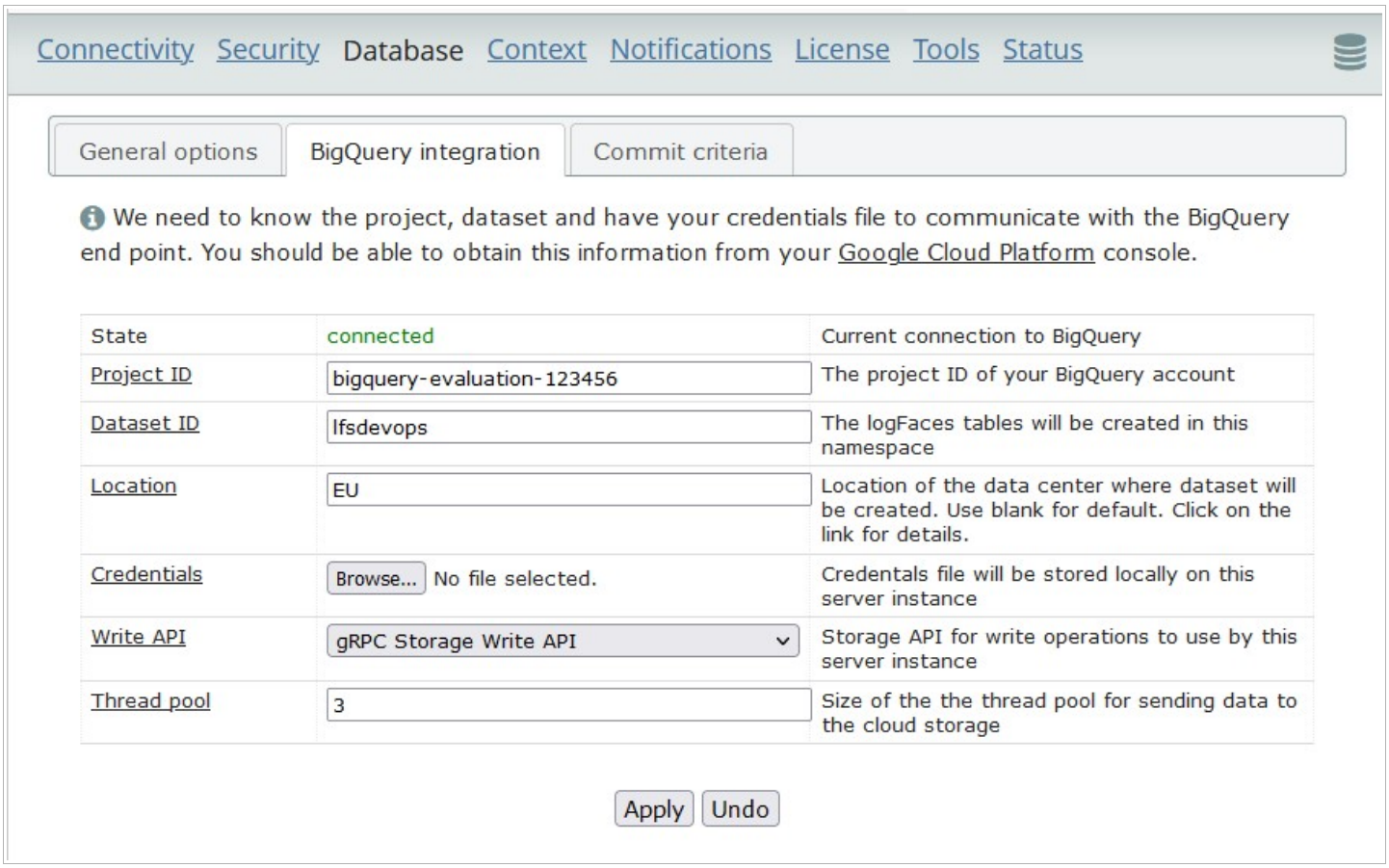


Figure 2.10.2: BigQuery integration settings

Here is a check list of the items needed for the BigQuery integration, all are obtained from your BigQuery project:

1. The **project ID** is obtained from your account when you start the project with Google Cloud. It can be a real or sandbox project. You will find its ID in your project settings.

2. The **dataset ID**, you create dataset in your project. Here is some information about the datasets. logFaces creates its schema tables in the dataset you specify here.

3. **Credentials** are used by logFaces to securely communicate with your storage. The credentials are obtained from your Google service account and come in the form of a **key file** which you request in JSON format. This file is stored in your logFaces /conf directory file named `bq.credentials.json`. Make sure you keep it safe.

   Alternatively, if you setup the hosting computer with Google Cloud tools, you can use Application Default Credentials (ADC) - a strategy used by the Google authentication libraries to automatically find credentials based on the application environment. logFaces server will fall back to this strategy automatically if it won't be able to fine the credentials key file.

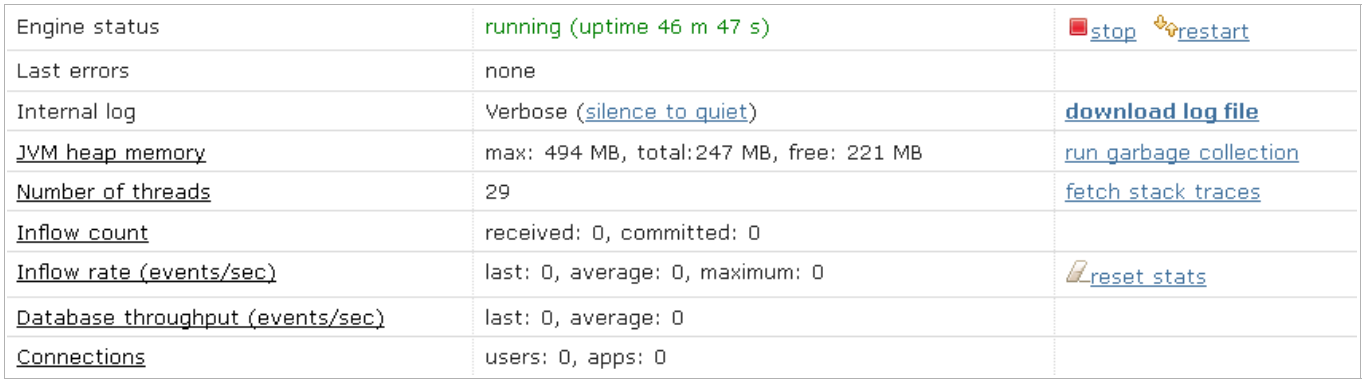
4. Note that your Google service account must have certain permissions. As of this writing the following roles are the bare minimum required for the logFaces to function properly - `bigquery.dataOwner` and `bigquery.jobUser`

Once these parameters are submitted, the logFaces server establishes a connection to the specified project and creates logFaces schema in specified dataset. When the server is connected to your cloud project a green indicator displays connection state. Otherwise, depending on the error, some adjustments will have to be made. Typically the errors are very descriptive, if not, refer to the server internal logs, or contact our support.

The data schema used with BigQuery is very similar to the other storage types we use. Names of the columns are chosen to be as short as possible to reduce the costs.

2.10.15 How do I tune the server for the best performance?

First of all, to determine whether the server under-performs, you need some metrics. You will find them in admin status page, below is an example of the most interesting parameters:



Engine status	running (uptime 46 m 47 s)	stop restart
Last errors	none	
Internal log	Verbose (silence to quiet)	**download log file**
JVM heap memory	max: 494 MB, total:247 MB, free: 221 MB	run garbage collection
Number of threads	29	fetch stack traces
Inflow count	received: 0, committed: 0	
Inflow rate (events/sec)	last: 0, average: 0, maximum: 0	reset stats
Database throughput (events/sec)	last: 0, average: 0	
Connections	users: 0, apps: 0	

The most common performance factor is the memory factor. Note that free heap memory should never get close to about 2-5% of the maximum allocated for the server JVM; when this happens the memory monitor will issue a warning in “last errors”. Once JVM memory gets drained, its behavior gets unpredictable and eventually may result in a complete fault. For the details read section named “How to increase server JVM memory”.

Another common performance factor is the balance between **inflow rate** and **database throughput** rate. The inflow rate is the number of log events arriving to the server per second from appenders. Database throughput is the number of events your database commits per second. You will see these numbers changing quite a lot and depend on many factors. To name just a few - network speed, database performance, log statement sizes, etc. When inflow rate is higher than database throughput, the server will activate an overflow mechanism which delegates residuals to a local disk storage while database is busy committing. Residuals are then flushed into database whenever database permits. In general, this mechanism is designed to prevent intermittent load spikes and can't be effective when the inflow is permanently higher than the database throughput. When local storage grows large, the CPU and IO usage may get much higher than normal. The larger the local storage grows, the harder it becomes to get the data from there. To prevent this from happening on regular basis you should watch the amount of “overload” - the percentage of total inflow which went through the local storage before it reached your database.

Try to adjust commit buffer size in admin database page. Note that larger commit buffer sizes not necessarily improve the database throughput, it very much depends on the database setup. Usually, most databases start getting sluggish when they significantly grow in size – try to control this by reducing day capacity or persisting less information.

Database indexes play very significant role in insert rates, try to tune your indexes to fit best your queries. Removing unnecessary indexes can significantly improve the overall performance – you can modify indexes in schema templates for the database you use, or alternatively – manage them with external database tools.

Of course the amount of server connections and threads play their role too. Each remote client or appender will result in an additional thread. Depending on the hardware and OS, these numbers should be taken into account.

When you see that server obviously under-performs, consider to either reduce the amount of load, or add another logFaces node to split the load.

2.10.16 How do I migrate my SQL database storage to MongoDB?

If your data storage is one of the SQL types and you wish to switch to MongoDB while retaining all the existing data, we have a migration tool built for this specific task. It operates on current logFaces instance using its current configuration which is expected to be for the SQL based storage. Your production instance can be active, but the process may take longer if SQL database is busy. So you may consider taking your server down for the migration purpose, or use off peak hours if you intend to run on live production system.

You need to manually prepare `/conf/mongodb.properties` file with all the settings we need to create the MongoDB storage and then copy the data. There is a template of this file available with the installation, you will want to fill in the details matching your MongoDB profile. More details on each parameter are available here. The MongoDB instance should be running and matching the settings in this configuration file before you proceed, e.g. host name, port, etc.

Run `/bin/mongo-migrate.bat` on Windows or `/bin/mongo-migrate.sh` on Linuxes. You will be asked to confirm your intentions and the process will being. Depending on the amount of data, the process may take some time to complete. Warning: logFaces collections (if exist) will be dropped in MongoDB before the migration.

If there are errors during the process, they will be printed in the terminal window and logged into the internal log file at `/log/lfs_server.log`.

Once migration processes completes successfully, you can switch your data storage in `/conf/environment.properties` file to work with MongoDB instance by modifying the corresponding parameter:

```
com.moonlit.logfaces.config.storage = mongodb
```

Restart your server instance for the changes to take effect.

2.10.17 How can I automate monitoring of logFaces server logs?

logFaces server itself has its own internal log which you can monitor by means of triggers or reports. The application name used by logFaces server is "**LFS**" and root package of server classes is named "**com.moonlit.logfaces**". When specifying criteria you can use either of those parameters to detect problems in real-time by means of triggers, or obtain scheduled summaries by means of reports. Generally it's a good idea to have this set up before you contact our support, we normally will ask for the internal logs before getting into the details.

2.10.18 How do I increase server JVM memory?

Open `\bin\ lfs.conf` file, this is a bootstrap configuration file, read more about bootstrapping here. JVM memory allocation is configured with those attributes:

```
wrapper.java.initmemory=256
wrapper.java.maxmemory=512
```

Those values are default, increase `maxmemory` when required, values are in MB.

2.10.19 How to pass additional arguments to server JVM

Additional arguments (-D like properties) can be passed to server JVM in the following way. Open `\bin\lfs.conf` file (this is a bootstrap configuration file) and look for a properties named like this:

```
wrapper.java.additional.xxx = yyy
```

Note how additional properties are numbered. When adding your own, make sure to increment the number. For example:

```
wrapper.java.additional.1=-Dfile.encoding=utf-8
wrapper.java.additional.2=-Dmy.prop.value=true
```

2.10.20 How do I use NTLM authentication with MS SQL Server on Windows?

1. Download latest jTDS driver from here http://jtds.sourceforge.net/
2. Place `jtds-xxx.jar` and relevant `ntlmauth.dll` under `/lib/dbdrivers` in your server installation
3. Modify `/bin/lfs.conf` so that JVM loads `ntlmauth.dll` by adding the library path property: `wrapper.java.library.path.2=..\lib\dbdrivers`
4. Comment out user name and password in hibernate properties file – this will ensure that NTLM authentication is used. Make sure that connection URL specifies correct domain.
5. If logFaces server runs as windows service, you will have to make sure that LFS service starts with proper user account and not default system account - modify these in service properties. Otherwise the driver will use currently logged in user credentials to connect to database.

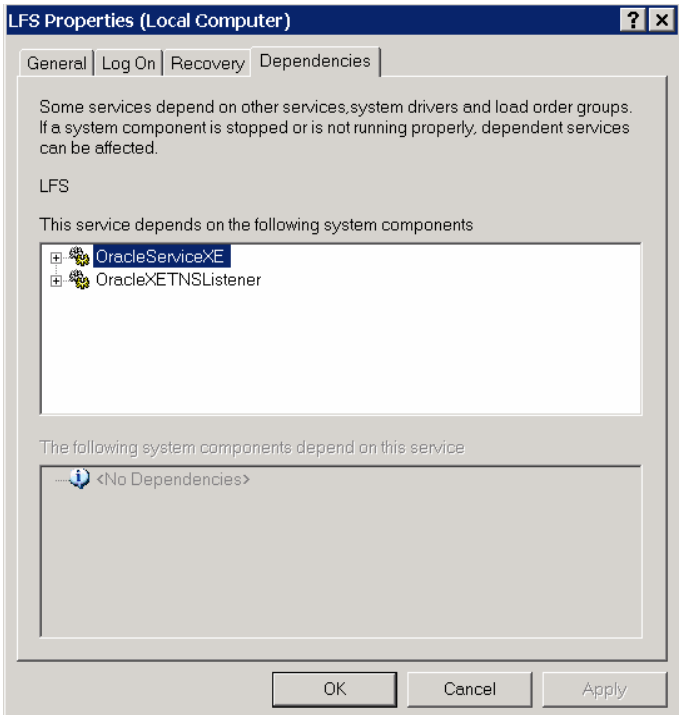
2.10.21 How do I make logFaces Windows service depend on other services?

In Windows it's often required to have logFaces service dependent on other services during start up. For example, if you run database server on the same machine as logFaces server, you may want to make sure that it starts only after database successfully starts. To achieve that, you must specify service dependencies.

Open **/bin/lfs.conf** file and modify `wrapper.ntservice.dependency.xxx` properties accordingly. For example:

```
wrapper.ntservice.dependency.1 = SomeService1
wrapper.ntservice.dependency.2 = SomeService2
```

Make sure to preserve proper numbering order and specify correct names of dependent services. Then you will need to re-register LFS service. Make sure the service is not running; execute `/bin/uninstallservices.bat` and then `/bin/installservices.bat` – this will re-register service configuration in Windows registry. To verify that everything went OK, check out service properties in your Windows administration tools. For example, in case of Oracle, the dependencies should look similar to what is shown below. If everything looks OK, go ahead and restart your computer to verify that dependencies actually work.



2.10.22 How to obfuscate passwords

Some passwords may be kept in configuration files; in order to prevent storing open text passwords, it is possible to obfuscate them and keep them in an encrypted form. To obfuscate your passwords, you can use existing Jetty server utilities located under `/lib/jetty` under your server installation. From the command prompt do the following:

```
java -cp jetty.jar;jetty-util.jar org.mortbay.jetty.security.Password text
```

This command will output an encrypted string which can then be placed in configuration files. On Linux the command should look like this (note the column separator):

```
java -cp jetty.jar:jetty-util.jar org.mortbay.jetty.security.Password text
```

2.10.23 Running in Docker containers

logFaces server can be deployed and run in Docker™ containers. Starting from version 4.2.2 we maintain our own image in a repository named `moonlits/logfaces` in Docker Hub.

Here is the `Dockerfile` used to create this public image:

```
FROM centos
ADD lfs.docker.tar.gz /root
ENV LFS_HOME=/root/lfs/home LFS_XMS=512 LFS_XMX=1024
EXPOSE 8050 55200 55201 55202 55203 1468 1469 514 515
CMD /root/lfs/bin/lfs console
```

This image is based on official version of latest `centos`, it requires `/root/lfs/home` volume mapped to an existing directory on hosting computer. It exposes port 8050 for clients and administration and other ports for appenders.

Note `LFS_XMS` and `LFS_XMX` variables, they specify defaults for initial and maximum memory for logFaces server JVM. These parameters are subject to override during run time in case your deployment has different RAM requirements.

This setup would probably cover most of the situations, but if not - feel free to make your own image from this template. For that you will need logFaces distribution for Docker™ containers - `lfs.docker.tar.gz` which can be found in our downloads. This distribution contains and uses its own headless JRE 11 from OpenJDK distributions, no need to install any other JRE. It also contains preset bootstrapping parameters for the server to know that it runs in Docker™, this is quite important because there are several catches.

As mentioned previously, logFaces server requires external volume mounted for its `home` location where it expects to find its license, configuration and other artifacts produced at run time. Those are logs, overflow files, configuration files, temp files, etc. This mounted volume will be passed to the Docker™ run command as an argument, so make sure to create its directory on hosting computer beforehand.

The following command will start logFaces server using home at local `c:\lfs-home` directory (on Windows), external port 8050 and single appender port 55200:

```
docker run -it -p 8050:8050 -p 55200:55200 -v //c/lfs-home:/root/lfs/home
               moonlits/logfaces
```

When started for the first time, logFaces server will deploy its default configuration into specified home directory on the host computer. This is where the license file should be placed (including the trial or evaluation licenses). If migrating, the existing `/conf` directory from you current setup can be copied to the new home while server is stopped, - when started it will pick it up and continue.

3 Getting started with logFaces client

logFaces client application is available for Windows, Linux and Mac OSX for 32 and 64 bit architectures. Think of logFaces client as a pair of glasses you wear when looking into the log stream pouring through the log server. The client will assist you to make sense out of this stream and convert large amount of log data into a meaningful piece of work.

3.1 Installing and uninstalling

On Windows, download and run the installer which will walk you through the process. Linux distributions come as **tar.gz** archives, just unzip the archive and run **logfaces** executable file. Mac OS X distributions come in a form of Mac OS X folder, open it and deploy to a location of your choice. To uninstall the client on Windows run the **uninstall.exe** in installation directory, on other OS's simply remove the client folder.

3.2 Modes of operation

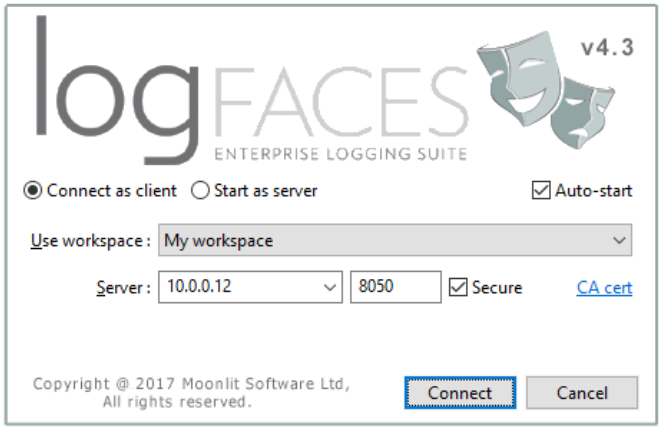


Figure 3.2.1: Client Mode

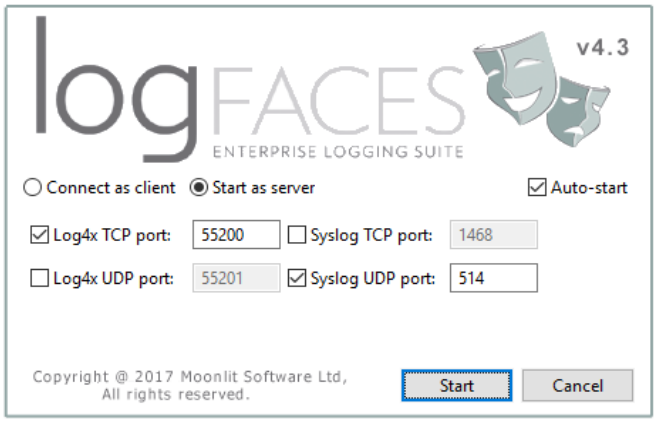


Figure 3.2.2: Server Mode

logFaces client can work in two modes - **Client Mode** or **Server Mode**. You select the mode during application start-up. In Client Mode the application connects to and works with one logFaces server instance. In Server Mode the application acts as actual log server but without database.

In **Client Mode** we specify logFaces server **host** name, **port** number and security option. These options will be remembered for the next time you run the client, but you can also indicate to use those settings automatically in the next time and not asking again. It is possible to modify these settings in the File/Preferences menu later. The communication between logFaces Client and Server is one way (from client to server) and is HTTP(s) based, so normally there shouldn't be a problem with firewalls. Of course, the access through the given port should be allowed by your network

administrator. If your server is configured to work with SSL, you can import server certificates in case the are not available in the local JRE trust store.

In **Server Mode** the application runs with embedded **compact** version of logFaces Server. This is a limited version of server and client combined into single application which provides **only real time viewing** of log data. You can use it when you don't need database and other features available in standalone logFaces Server. In order to run in **Server Mode** we need to specify at least one of the ports which will be used by the application to receive log data from appenders. You can specify either TCP, or UDP or both, just make sure those ports are available and your application appenders are configured accordingly.

Both modes look very much alike from user experience point of view, except that in Server Mode there are no database and querying features.

*Note that in order to run in **Server Mode** you need to install the license on the computer where you run it. Note also that this is not the same license type as installed on the logFaces Server. This license needs to be purchased and installed separately unless you hold OEM or Site license.*

3.3 Workspace

When using several server nodes, you often need to switch quickly from one system to the other. This is done by means of Workspace which stores connection parameters, real-time perspectives, queries, counters, and all other settings. Workspace can be exported to and imported from text files. Additionally, Workspace can be stored on server and shared amongst team members, this way you do the setup only once and let others to import your Workspace with all the settings. File menu contains all workspace related actions :

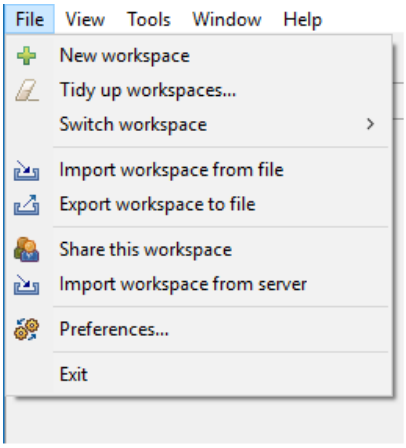


Figure 3.3.1: Workspace actions

3.4 User interface concept

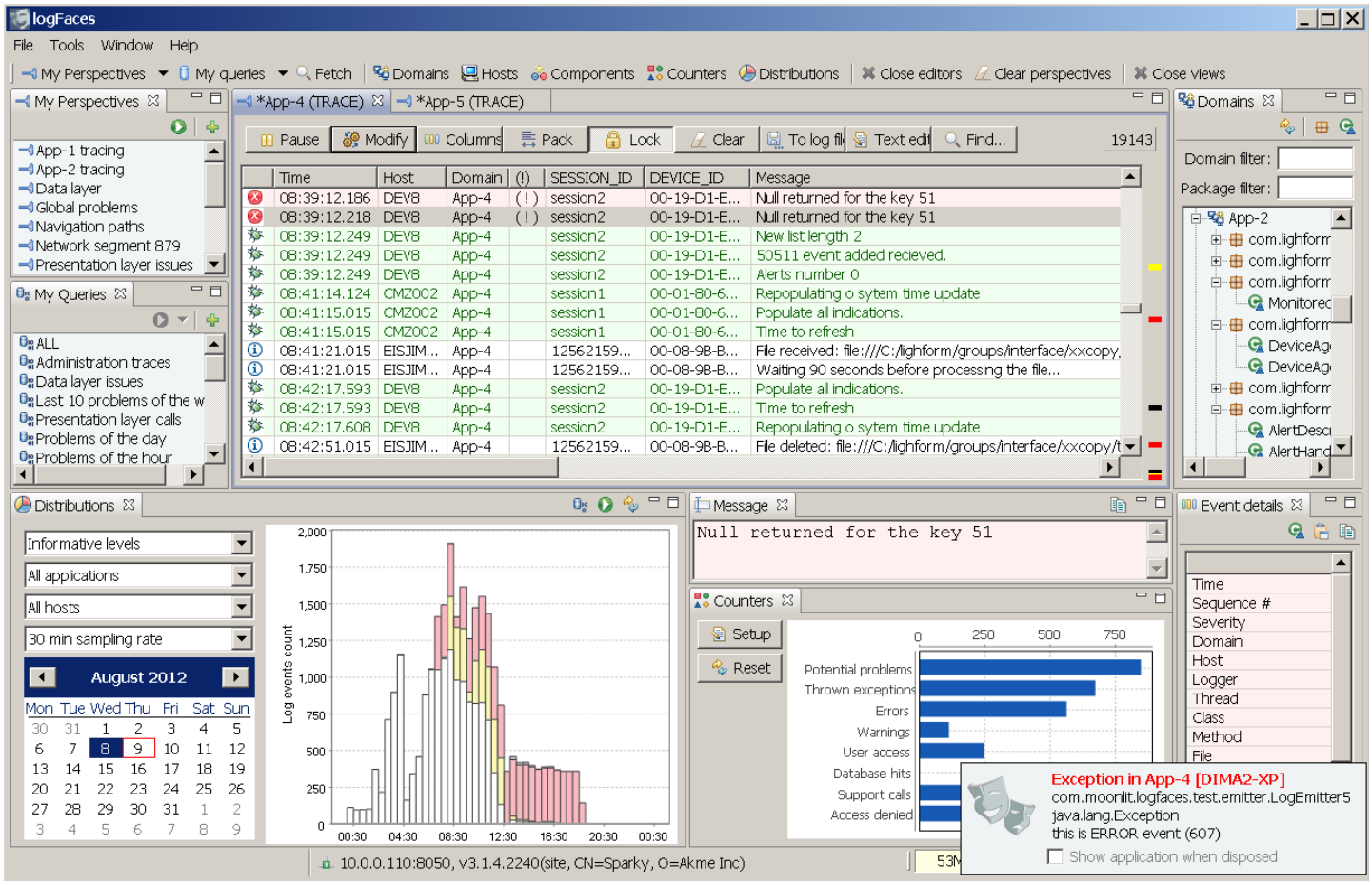


Figure 3.4.1: logFaces client user interface

logFaces is designed with two major goals in mind – real-time **monitoring** and **drilling** into the log history. The user interface you see above is built for achieving those tasks. The client is an Eclipse RCP application and retains most of the Eclipse paradigms – there is a working area in a middle and it's surrounded by auxiliary views which drive the work flow.

Real time data and query results are displayed in a middle part within data tables and each piece of information has its own tab. You can move those tabs around to have several views displayed together.

Along with the data, there are structural pieces of information like names of loggers, applications, hosts, exceptions, event details – they occupy axillary views around the main area. Axillary views also can be hidden, shown or re-arranged as you see fit.

The following sections explain how to do the monitoring and drilling as well as the usage of auxiliary views.

3.5 Data tables

Log data is displayed in tables - flexible to manipulate and easy to navigate with. Once you have data (real time or historical) you will want to customize the way it looks or navigate from it somewhere else. Table headers are special – they allow additional filters on existing table content. For example, if you want to focus on particular Thread, click on the corresponding header and select that thread. Several column filters can be combined. Tables can also be displayed with specified columns so that you only see relevant details. The content can be searched and annotated.

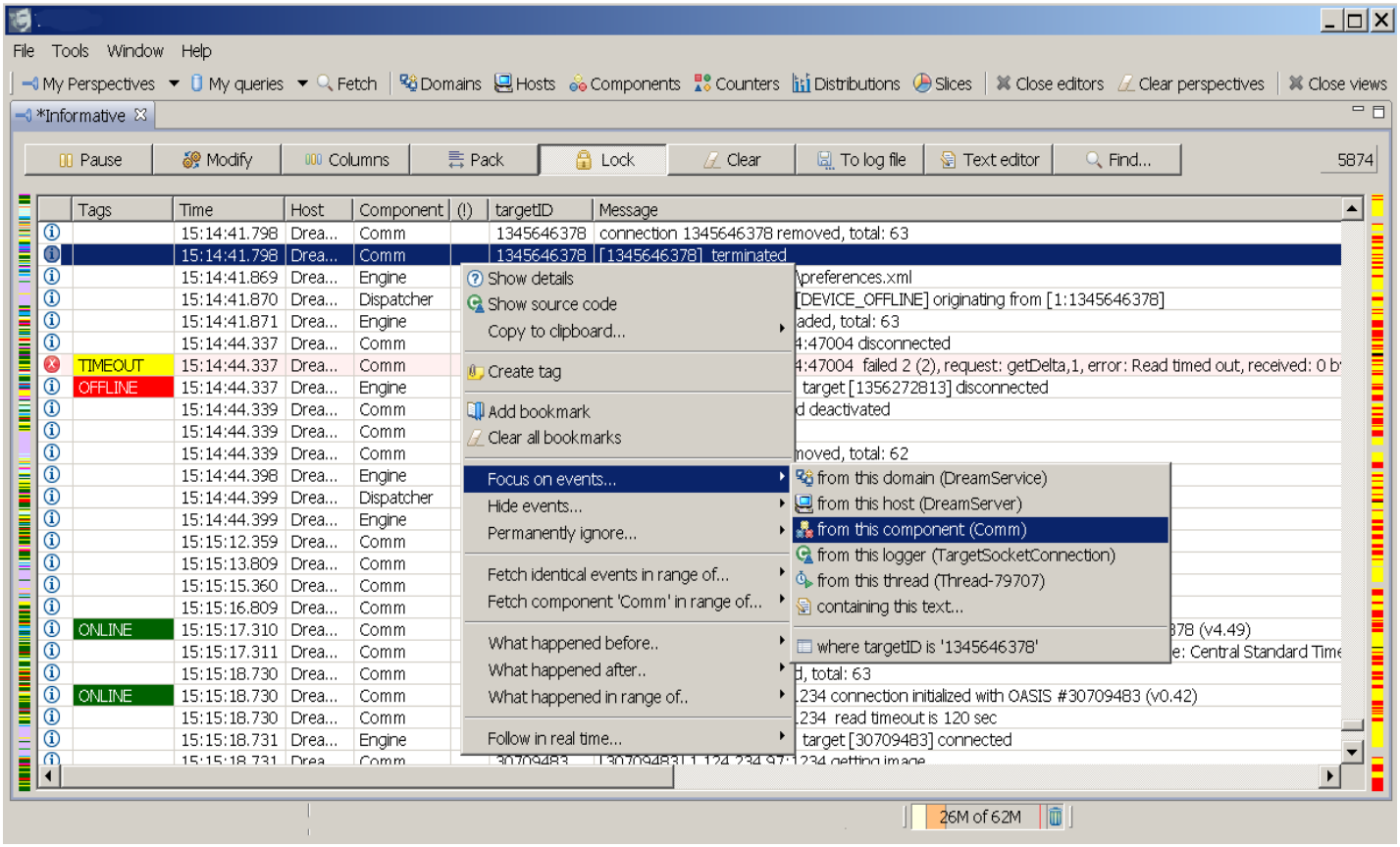


Figure 3.5.1: Data tables

Note the colored bars on the sides – **problematic** events are color-mapped on right side bar while **tagged** events are color-mapped on the left side bar. Hovering the mouse over those bars gives a quick jump to a particular event without too much scrolling around.

The content of data tables can be instantly converted into textual log files – click on “To log file” button or even open the text in an editor by clicking on “Text editor”. The text layout can be specified in preferences.

3.6 Monitoring tasks

Real time monitoring is a unique feature in logFaces as it was designed for incredible volumes of data going through the server. To serve many users, logFaces server pushes logs to each client separately and according to its very own criteria filter. Those filters you will find throughout the system - including queries, administration tasks like reports, trigger and database. This is how real time perspective is defined:

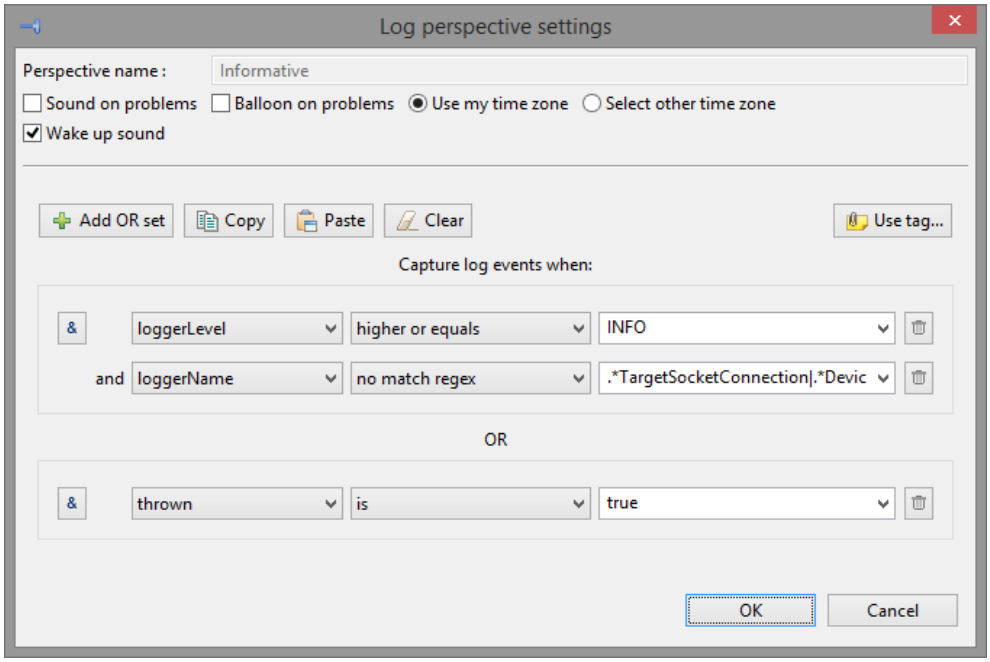


Figure 3.6.1: Perspective settings

Along with criteria filters there are other options – alarm and wake up sounds, balloon pop-ups and time zone presentation. Note that perspectives are named and can be stored for later use in “My perspectives” axillary view. Perspectives can be created from several contexts:

- From “My perspectives” view where you will specify the criteria yourself
- From most other axillary views where perspectives can be launched instantly using the selected context. For example, right clicking on a host name within “Hosts” view you can create a perspective which will listen for selected host logs.
- From data tables (query results, or other perspectives) where you can “Follow” some interesting item by right clicking in the table rows.

Perspectives tool bar gives you full control of each perspective separately where you can pause, resume, setup columns, lock, clear or do text searching.

3.7 Data mining tasks

Real-time monitoring alone is not good enough unless accompanied with data mining. You will find numerous ways of getting the historical data – there is an instant fetch, context drilling, range look ups to peer into all sorts of time spans – all made simple and easy to use. Database queries are based on the same idea of criteria filters:

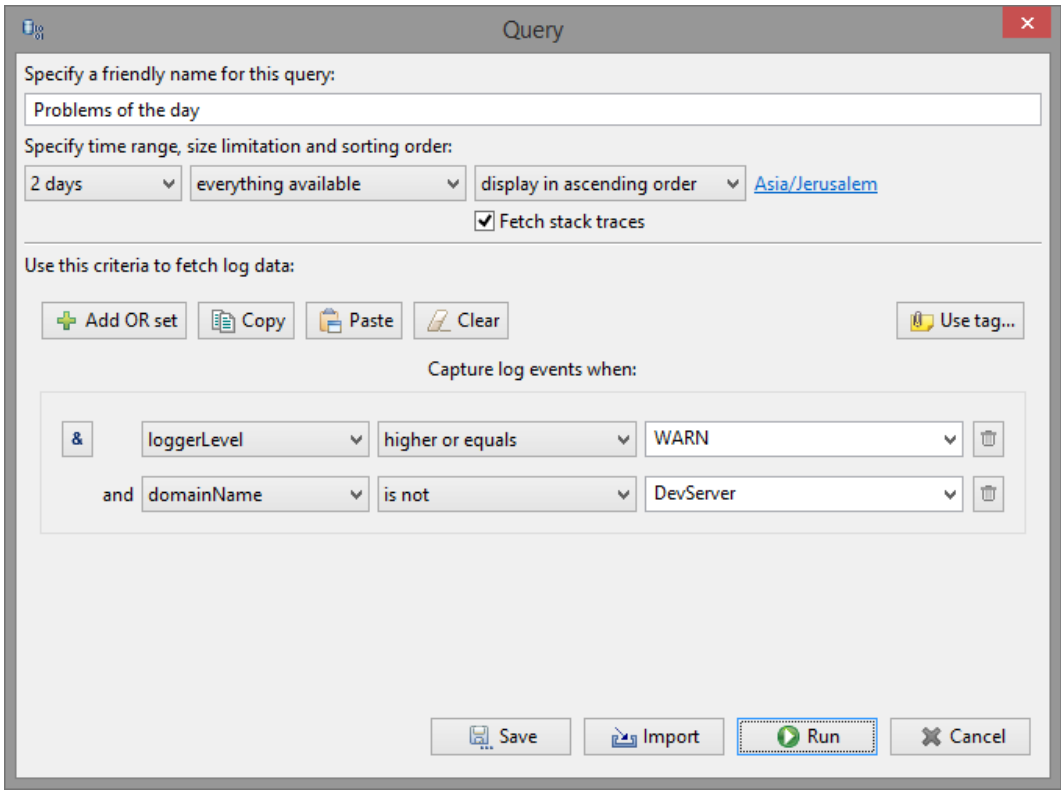


Figure 3.7.1: Database query

Along with criteria you specify a time range to cover, result set limitation, order display and time zone for presentation. Note that queries can be named – you may want to keep some of the queries you use often in “My Queries” view and spin them out any time. Since queries are based on predefined time ranges, they will always work no matter when you run them. For example query with “1 day” time range will give you data spanning from this moment and back 24 hours.

Queries can be launched from various contexts in data tables and most axillary views. Those queries which you decide to keep in “My Queries” can be instantly launched from main tool bar drop down menu.

3.8 Criteria with parameters

It is often useful to run the same query or real-time perspective with slightly different inputs. This can be achieved by setting up **${param-name}** variables in criteria fields. Those variables are then substituted at run time before criteria is applied.

For example, the query below will require user input for a “serial-number” and whenever query is executed the user will be prompted to input that value.

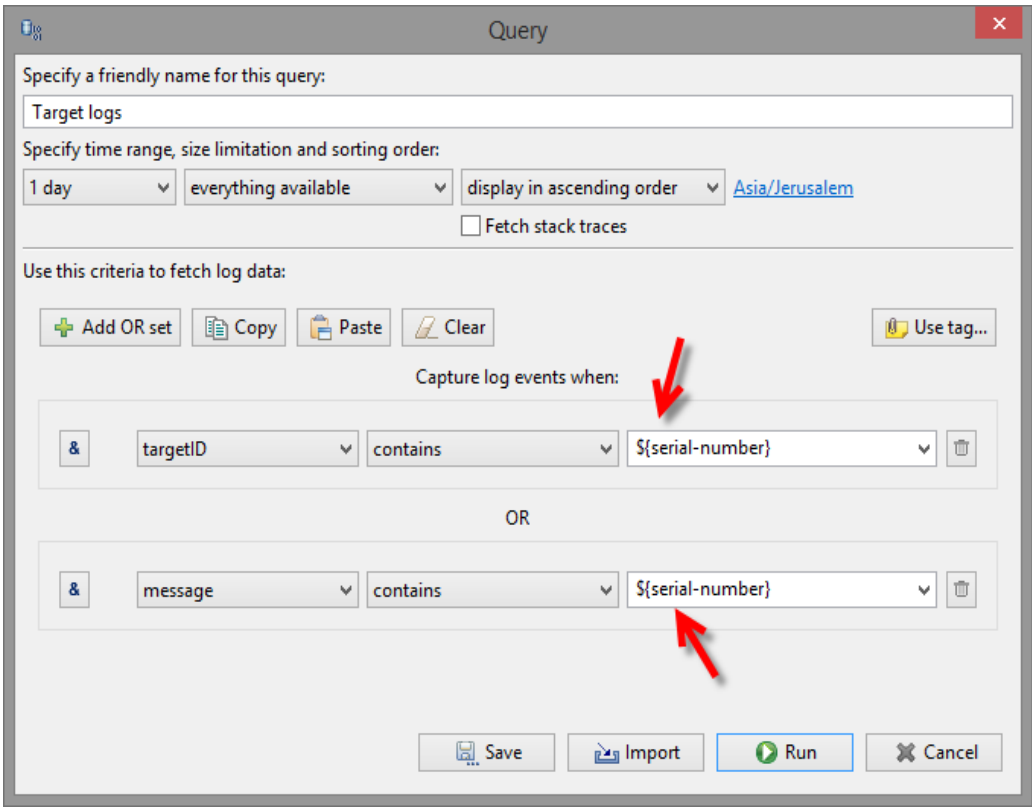
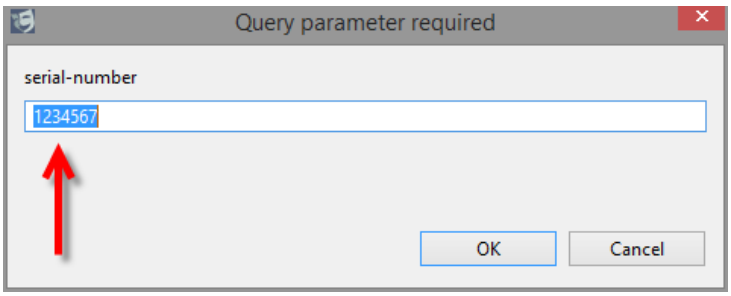


Figure 3.8.1: Query with parameters



3.9 Analytical tasks

Few clicks away you will find convenient tools for visualizing and analyzing log data. There are Dashboards, Distributions, Slices and Counters. Combined use of those tools will get you better understanding of the underlying logs. This is particularly true when dealing with large volumes of log data.

3.9.1 Dashboards

Dashboards are containers for widgets each of which displays certain aspect of the log data. Once you design your dashboards, they will instantly visualize the story going on behind the logs.

For example, the dashboard below illustrates some “user activities”. It shows the recent 5 minutes of the story which can be stretched and shrank as you need it. As you can see, there is a **list** of currently active users, **counters** of page hits and sessions, some activities **sliced** and grouped into pie charts, and **spread** in stacked time bars.

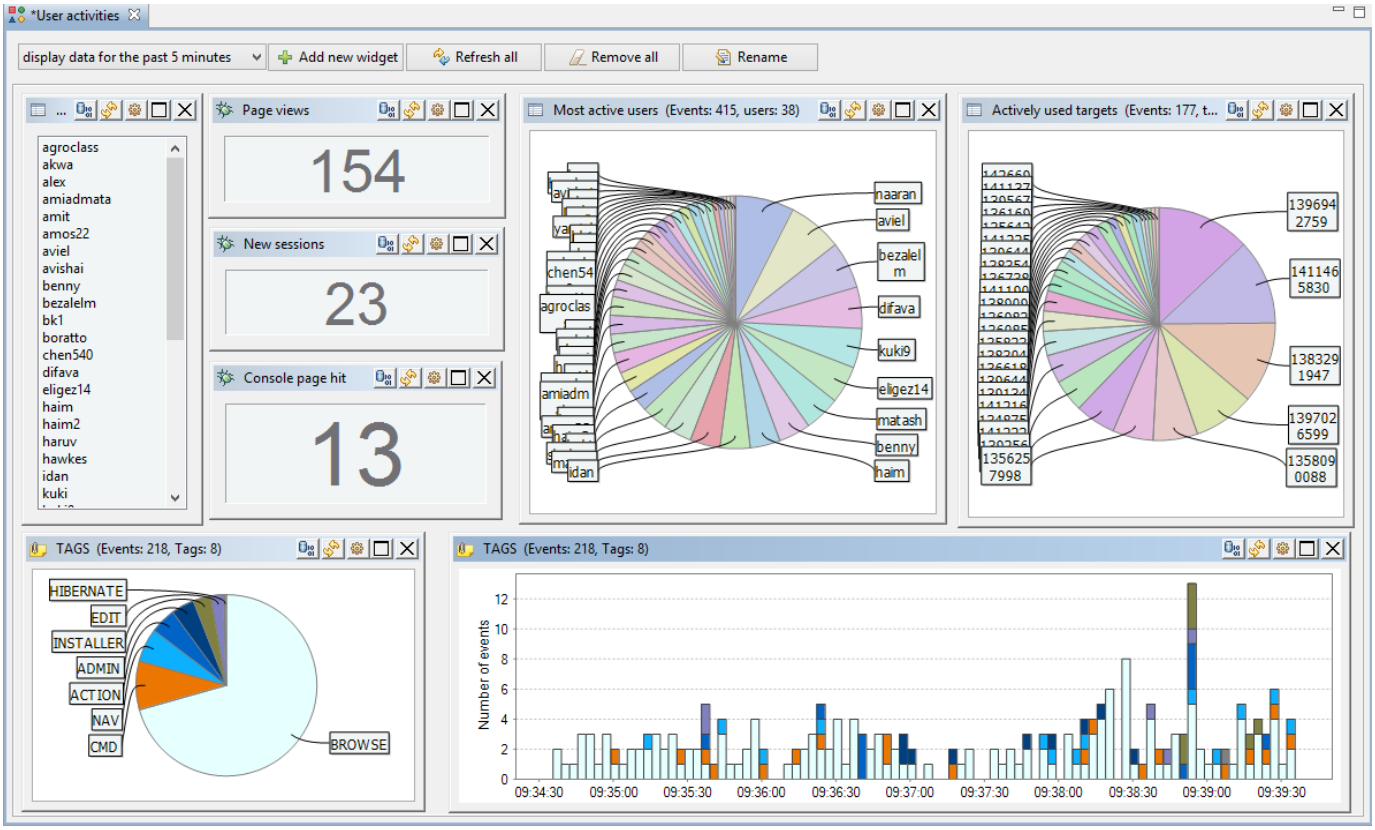


Figure 3.9.1: Dashboard example

You can re-arrange widgets on the dashboard to your liking. The dashboard state is persisted so that next time you open the dashboard, it will show up in exactly the same state as you left it. All your dashboards are stacked under the main toolbar menu so that you can instantly bring any (or all of them) to the front view. When displayed, the dashboard will automatically refresh its content every once in a while. There are predefined time ranges for fast time traveling.

As you have seen there are several types of widgets supported and each type renders itself differently on the dashboard. Widget window has a button bar on its top right corner to allow the access to actual log data, settings, manual refresh and maximizing. Widget window can be re-sized and dragged around the dashboard.

The content of the widget window is very active, clicking on its part will perform drill down and extract actual log statements. For example, clicking on the slice will deliver the log data for that particular slice. Clicking on the bar will bring logs from that particular bar and time. And so forth..

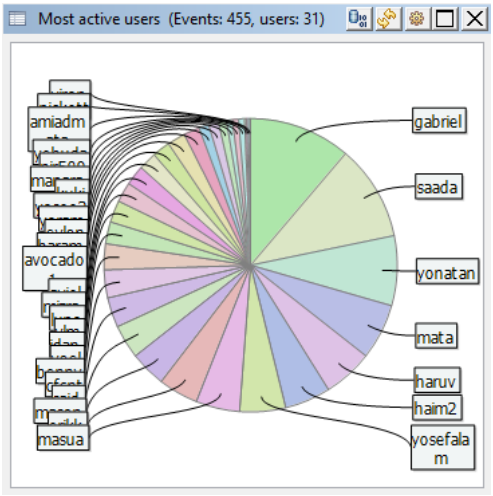


Figure 3.9.2: Slice widget

Figure 3.9.3: Counter widget

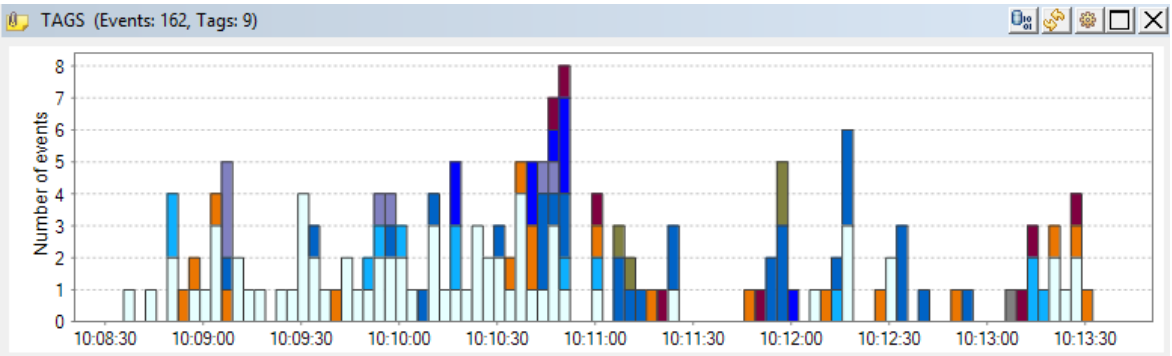


Figure 3.9.4: Time line widget

The widget itself is a query with some additional parameters needed for its display. The example below illustrates a widget named “Most active users” - this name will be shown on the widget title bar.

This widget is of a type “Pie chart” where we aggregate all data and slice it by category “**user**” which actually is an MDC name in our system.

Now, the data we will be dealing with is a query result based on logger level, domain name and host name. Everything this query produces will be sliced by “user” and rendered as a pie chart we have seen in the previous example.

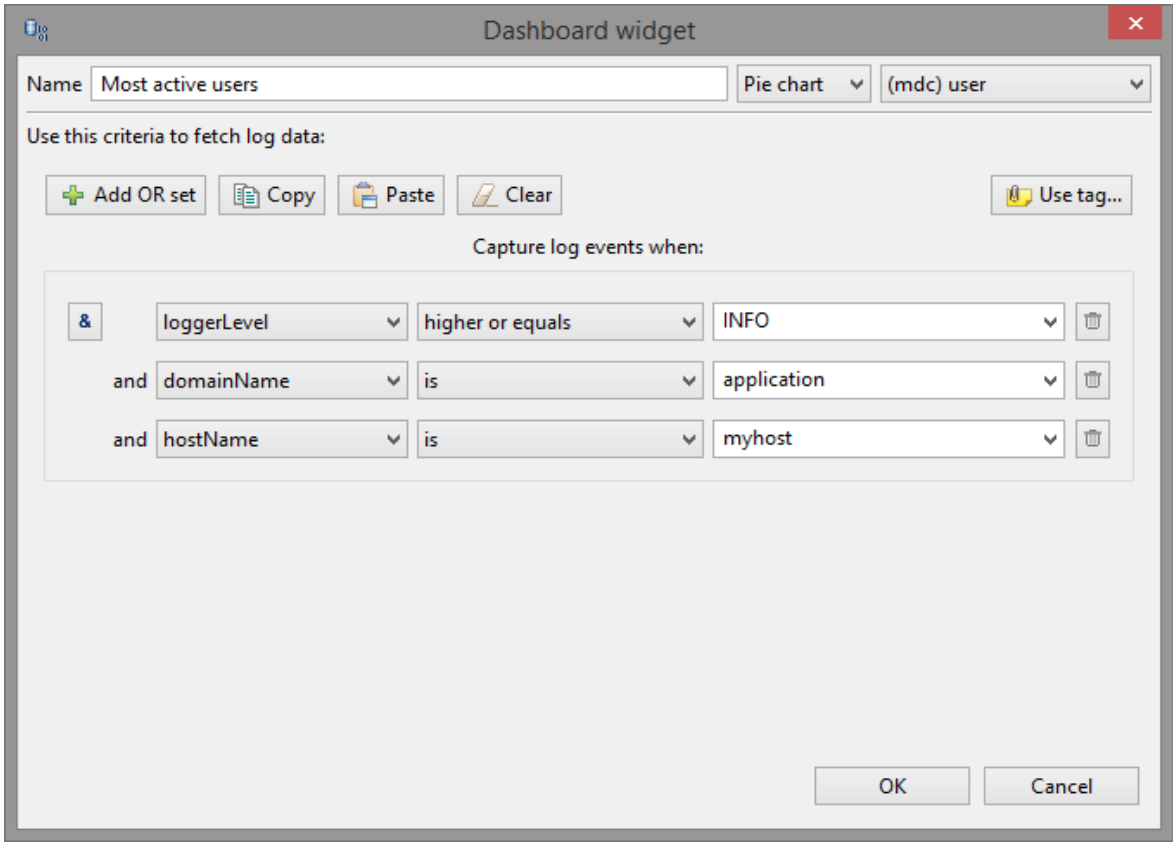


Figure 3.9.5: Widget setup dialog

3.9.2 Distributions

Distributions is an axillary view for visually spreading log data in time by categories. Distributions work in the context of some data display like real-time perspective or query results. The example below illustrates how query results are visualized by distributions.

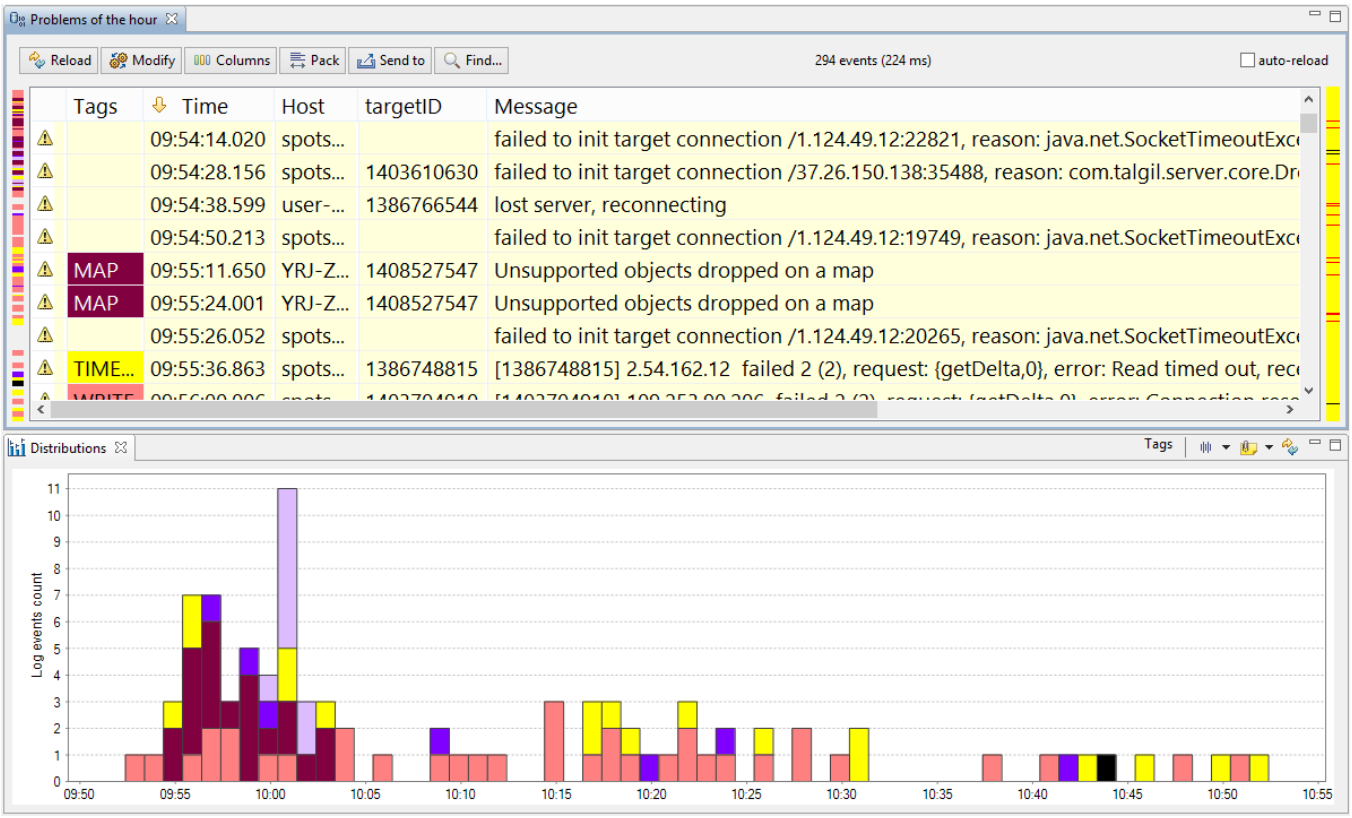


Figure 3.9.6: Distributions example

Distributions chart aggregates logs on time axis using some category for stacking the data. This way you see the activity in the context of the time which is covered by actual result set.

There are built-in categories to choose from - severity levels, host and application names, exceptions, tags, MDC names, etc. But there are also ways to define your own categories. See "Named regular expression" section for more details.

While distribution chart is displayed, you can modify its category and time rate on the top right tool bar of the view. Left clicking on the bar will and get you the underlying log data of given time. Right clicking on the bar will fetch events of clicked category from entire time range.

3.9.3 Slices

Unlike distributions which show time line of logs, **Slices** view displays categorized quantities and their correlations. Slices view is also linked to currently displayed data table and will automatically update when data table gets changed or switched. Below are several examples:

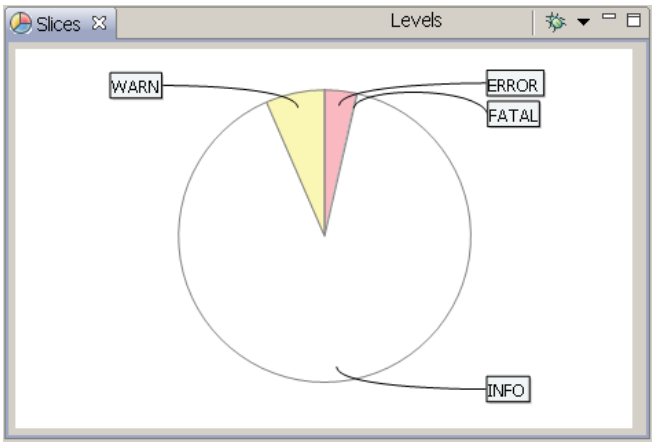


Figure 3.9.8: Sliced by severity levels

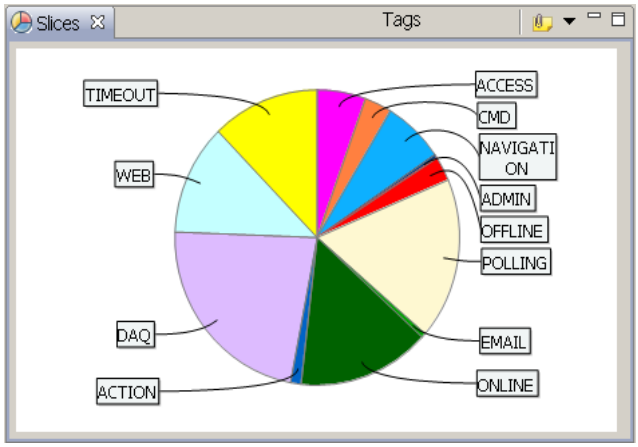


Figure 3.9.7: Sliced by tags

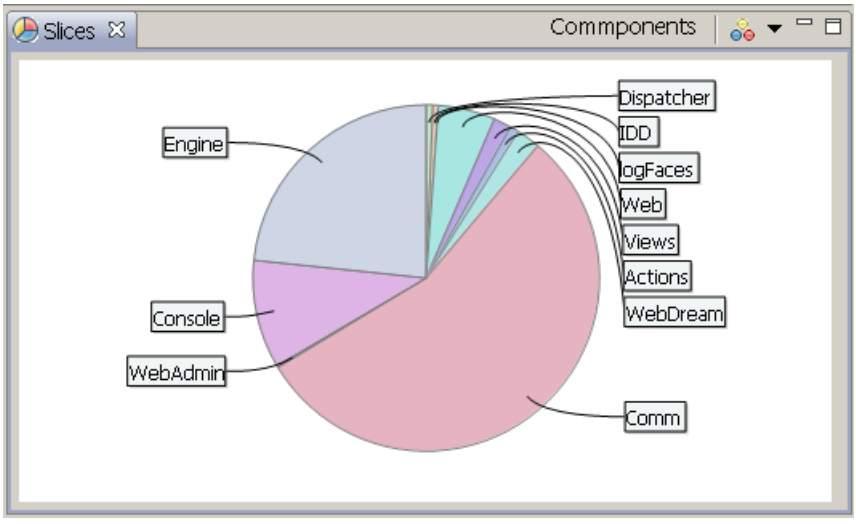


Figure 3.9.9: Sliced by components

3.9.4 Named regular expressions

As you may have noticed from previous sections, the analytical views are based on categories which derived directly from logging event structure - names of applications, host, loggers, exception, etc. To extend this list it is also possible to extract data from messages and use those data as category for further analysis. If you have log messages of repetitive structure but variable in terms of content, some of this content may be used to create custom categories. For example, consider the following log messages:

```
program started, env: Windows 7 (6.1), x86, JRE 1.6.0_54
program started, env: Windows 7 (6.1), x86, JRE 1.7.0_64
program started, env: Windows 7 (6.1), x86, JRE 1.7.0_71
```

To capture and plot the information about JRE version we use regular expressions which we also name so that we can easily find them later. Once defined, named regular expressions will show up in a list of **categories** in Distributions and Slices views. To define and test those expressions, open file File/Preferences and select **Named regex** section, it looks like this:

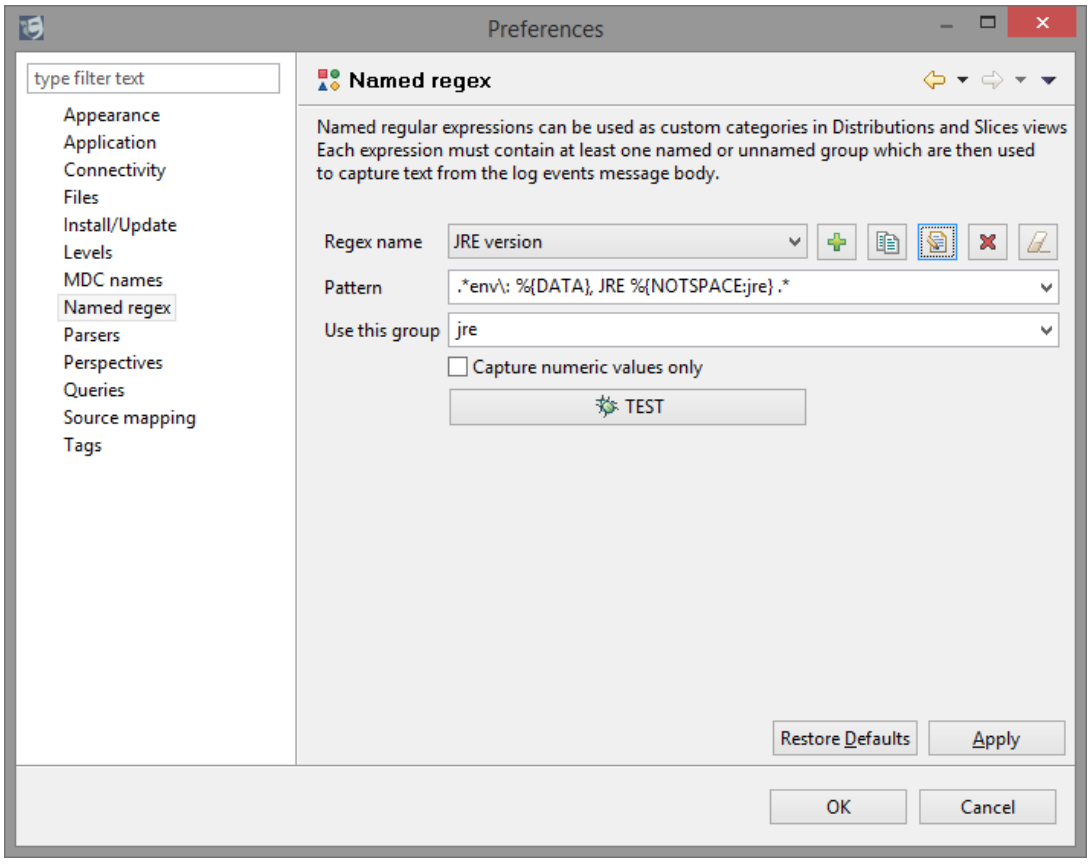


Figure 3.9.10: Named regular expressions

To be used along with other categories, regular expressions are named, hence the term.

The regular expression pattern must include at least one **named group** which will be used for capturing data from text. If you specify more than one group, you should also select which of the groups to use for data extraction. It is possible to use expressions from your patterns library, the notation of **%{pattern-name}** still applies.

When named regular expression is applied to a collection of logs, it will scan each event message body to match the regular expression at hand. When match, the group data will be extracted from the text and used as a series for display in analysis charts. For example, a slice with applied above example may look like this:

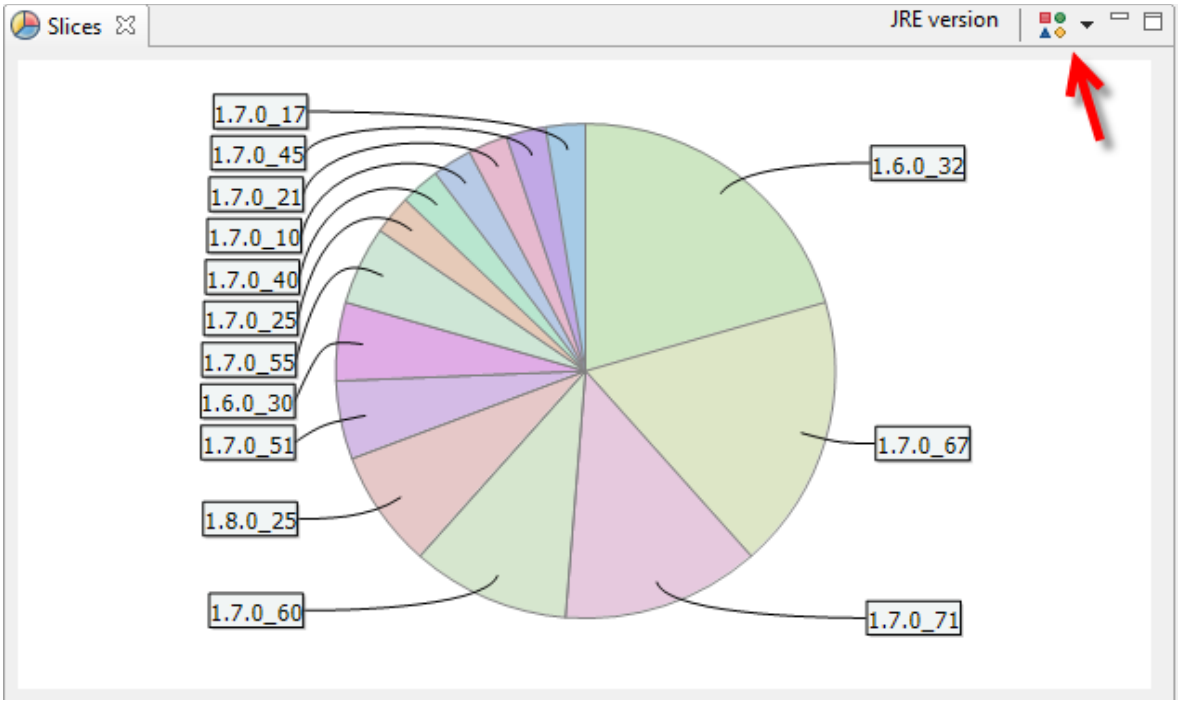


Figure 3.9.11: Named regular expression applied

As you can see, the most used JRE version in this example is 1.6.0_32. But you can also see that there are 14 different versions used altogether. That may be a valuable information.

If you click on either of the slices, logFaces will drill down and extract from its database the log events contributed to the picture we see.

Another interesting feature of named expressions can be seen with Distributions where aspect of time comes into play. If your logs contain numeric information, we can extract it and instead of counting the amount of log events matching, we can display extracted numbers.

Consider the following example, these logs reveal some data being collected:

```
collected 132 records from ....
collected 596 records from ....
collected 28 records from ....
collected 24 records from ....
collected 30 records from ....
collected 16 records from ....
collected 42 records from ....
collected 45 records from ....
collected 2 records from ....
```

Named regular expression for this type of data should indicate that we want numbers extracted if possible. Select "**Capture numeric value only**" check box in expression editor panel.

A pattern like "**collected %{NUMBER:num} records .*"** will extract and accumulate the numbers along the time line and using specified rate.

Finally, when applied to real log data our named expressions will produce this:

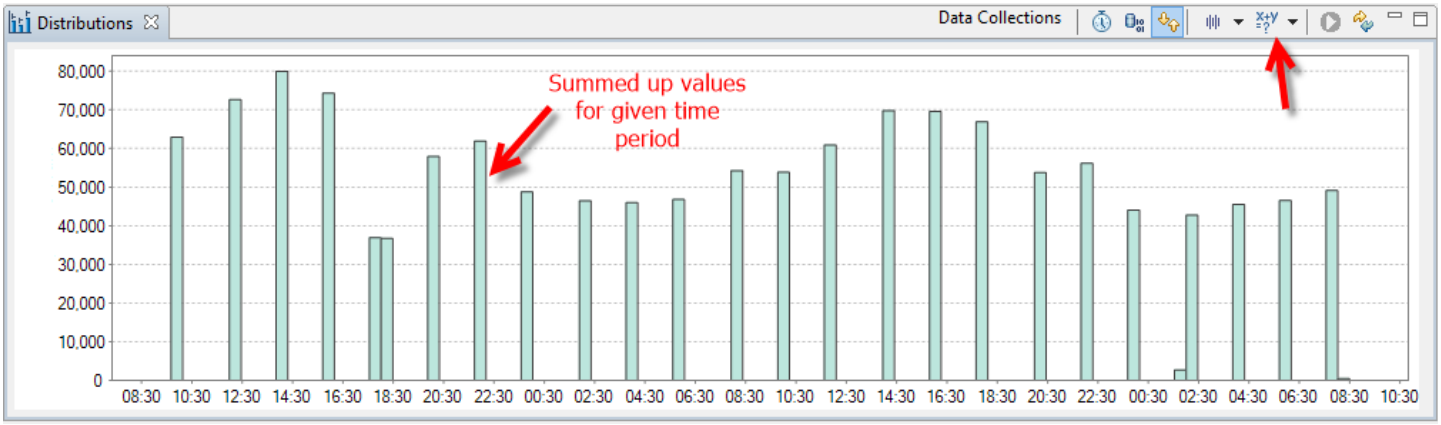


Figure 3.9.12: Numeric distributions

Note how Y axis for numeric expressions shows the **sum of extracted numbers** for a given period of time. In this example, it's the number of "collected records". And again, clicking on an individual bar will display the contributing log events which produced this picture.

3.9.5 Counters

Counters is another form of looking at logs – by counting certain events in correlation to each other. Counting is done in **real time** and can be setup separately per client and reset at any time. From server point of view Counters are yet another log perspectives but they displayed differently:

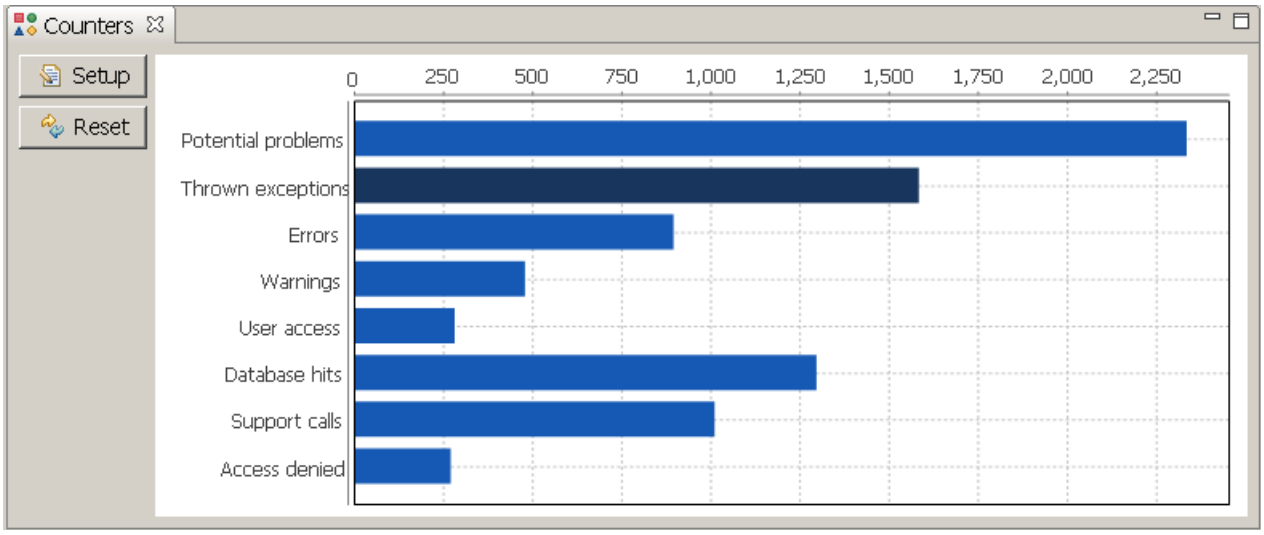


Figure 3.9.13: Counters view

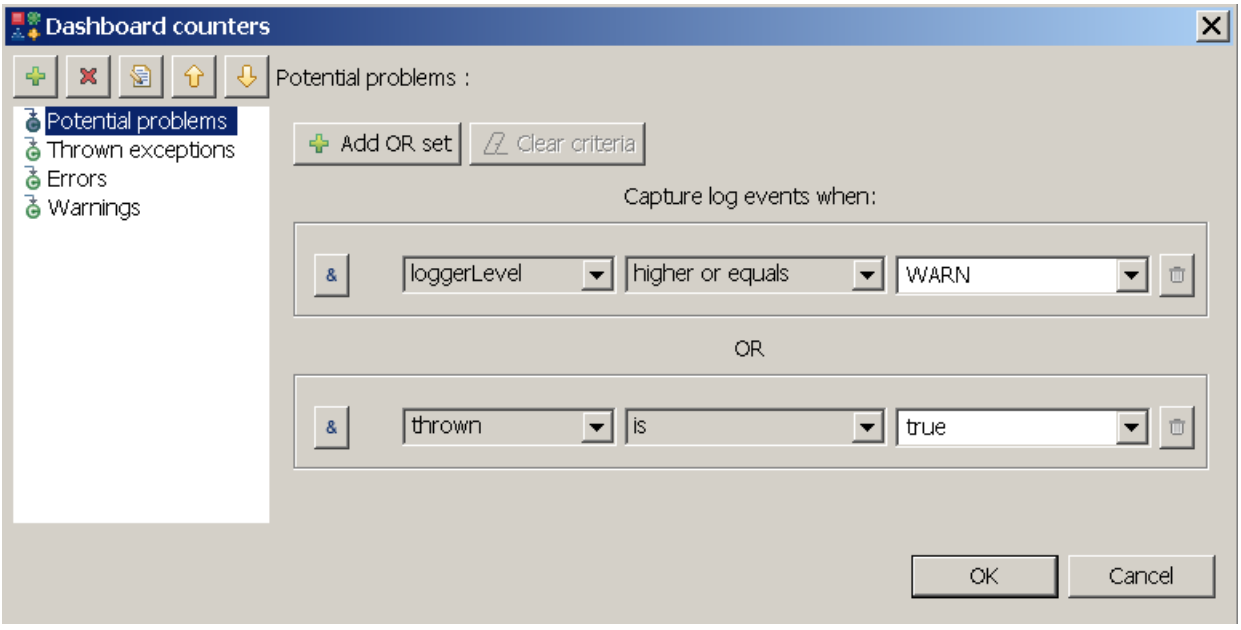


Figure 3.9.14: Counter setup

Each counter has its own criteria filter and name. Whenever criteria is met the counter gets incremented and its bar grows horizontally. Clicking on the bars will fetch the contributing log statements into ordinary data table where you can further work with the logs.

3.10 Repositories

Domains, hosts and components are jump starts into real-time monitoring and data mining tasks. Those views provide structural information about your log data and underlying system. These are actual names of applications, hosts, packages and classes. This information is accumulated by server continuously and never gets removed. Use refresh button to update the content of the views, they aren't updated automatically.

Use right click context menu to spin off real time perspectives or queries. Selected items will be used to construct criteria filter automatically.

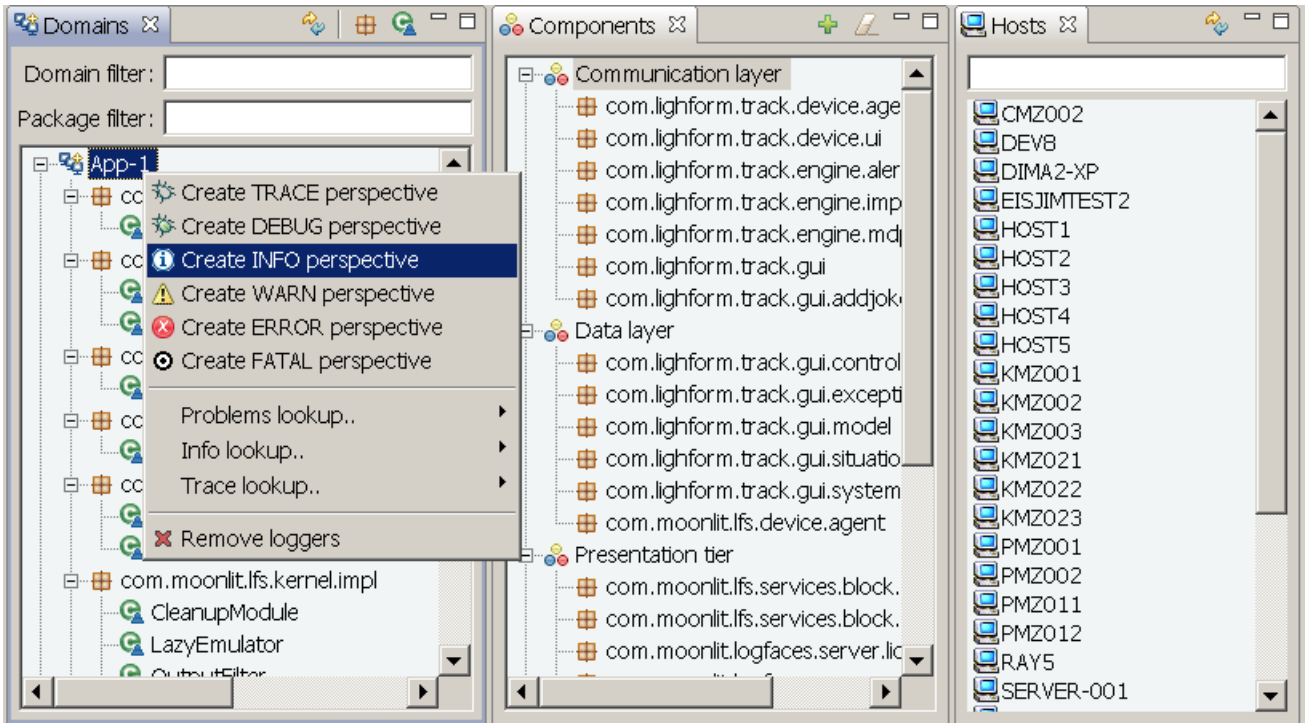


Figure 3.10.1: Repositories

One note about **Components**. This view is special because components are artifacts, server doesn't have a knowledge of them. Component is a collection of packages grouped together to allow you additional abstraction level of the underlying system. For example, you can make a “Presentation layer” component holding dozens of packages and since components are named, you will be able to use those names in filtering and queries.

3.11 Messages, exceptions and details

Whenever something is selected in data tables, the additional details are also shown in separate views for convenience. Every log event has a message text; when it's too large or spans several lines the “**Message**” view will show it nicely so that you could select and copy text into a clipboard.

When event carries an exception, the “**Exception**” view will be displayed with full stack trace where you further jump into sources.

“**Event Details**” view has a summary of all information known about the log statement. Everything is colored by event severity so that you will easily know what it's all about. For example, below we have an error selected in data table, so the “Message”, “Exception” and “Details” views are painted with the red background.

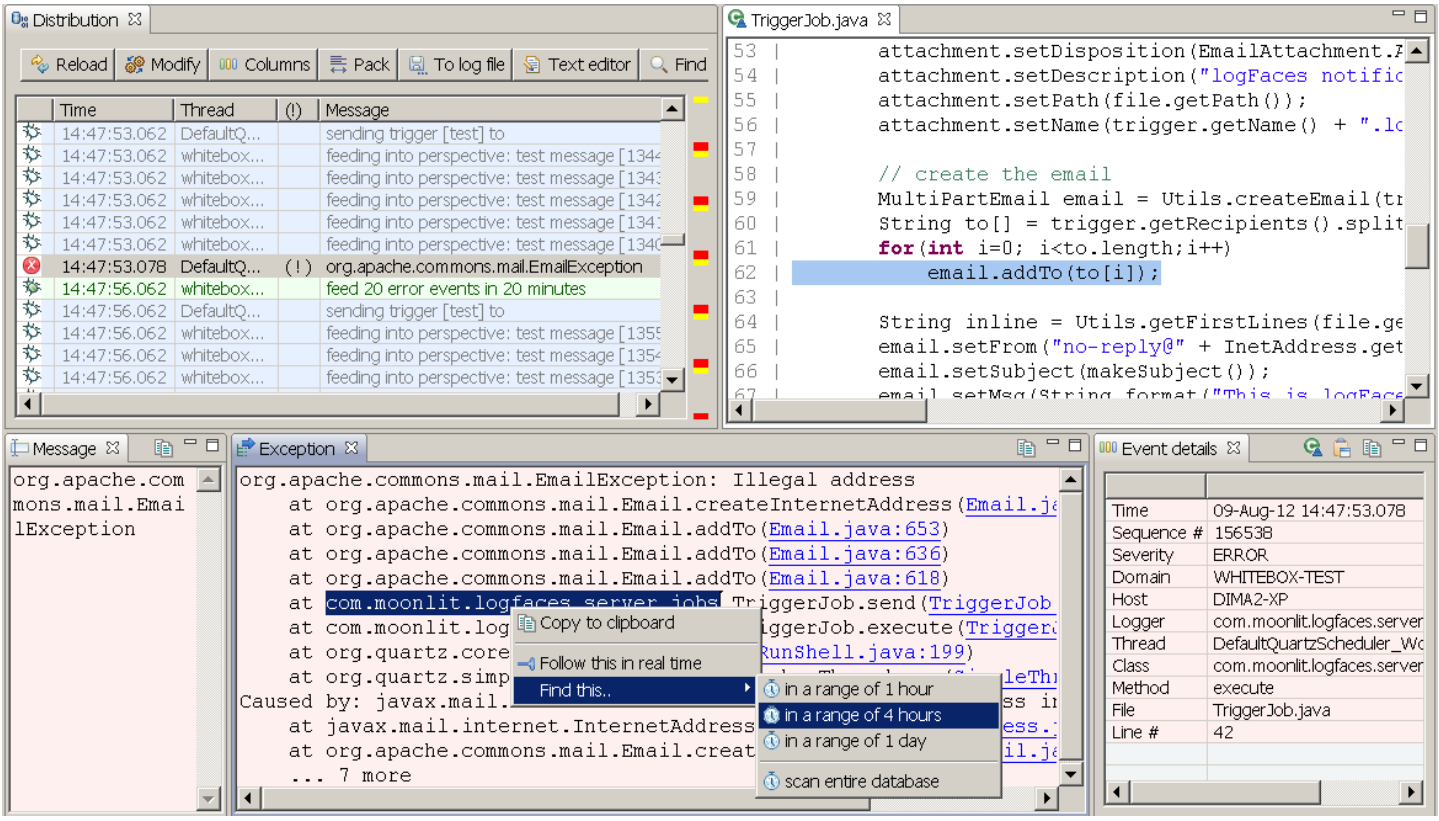


Figure 3.11.1: Using auxiliary views

You can also spin queries from selections and even real-time perspectives to follow particular patterns – use the mouse right click wherever possible.

3.12 Source code mapping

Having appenders to transmit location information allows instant access to the origin of the log statements. Most appenders have a property named “**locationInfo**”; when emabled, the appender transmits files name, class name, method name and the line number of the originating log statement.

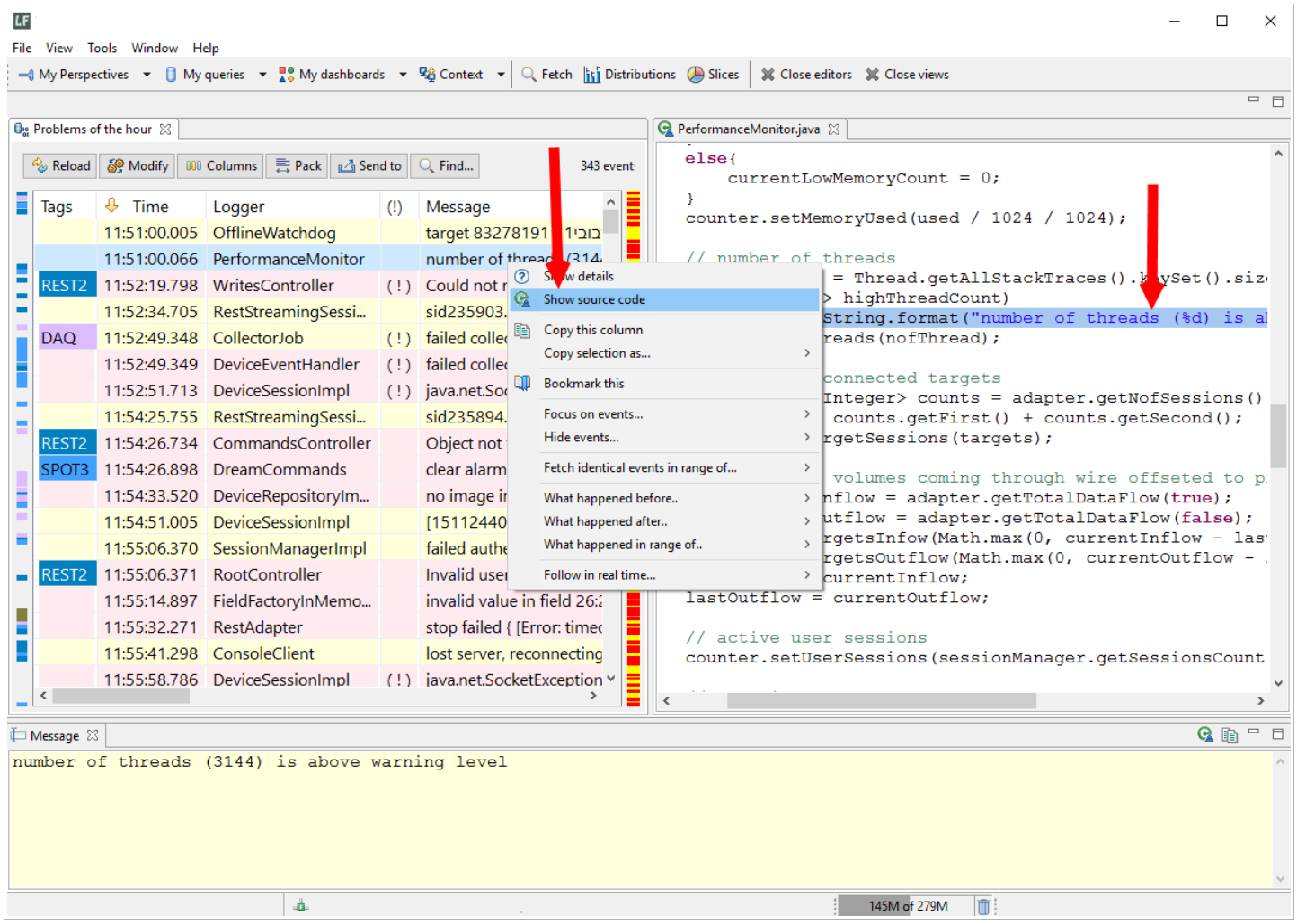
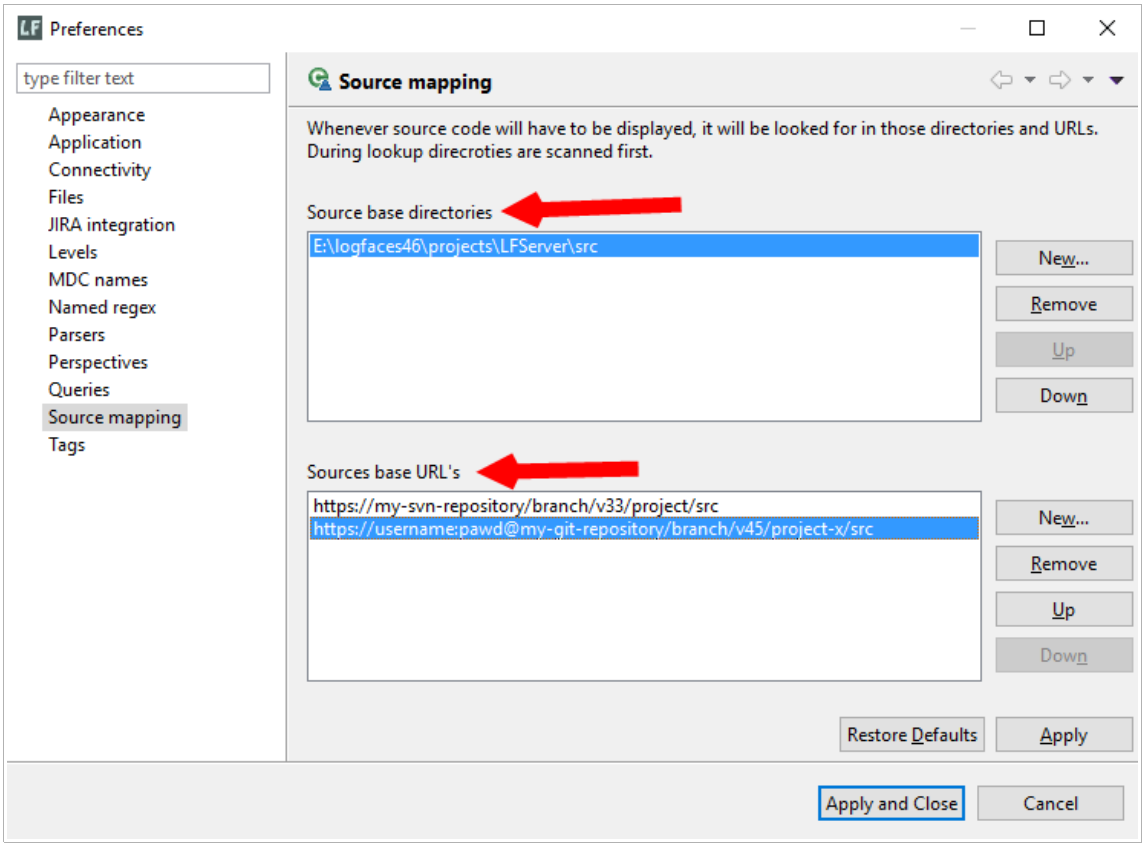


Figure 3.12.1: Accessing the source code

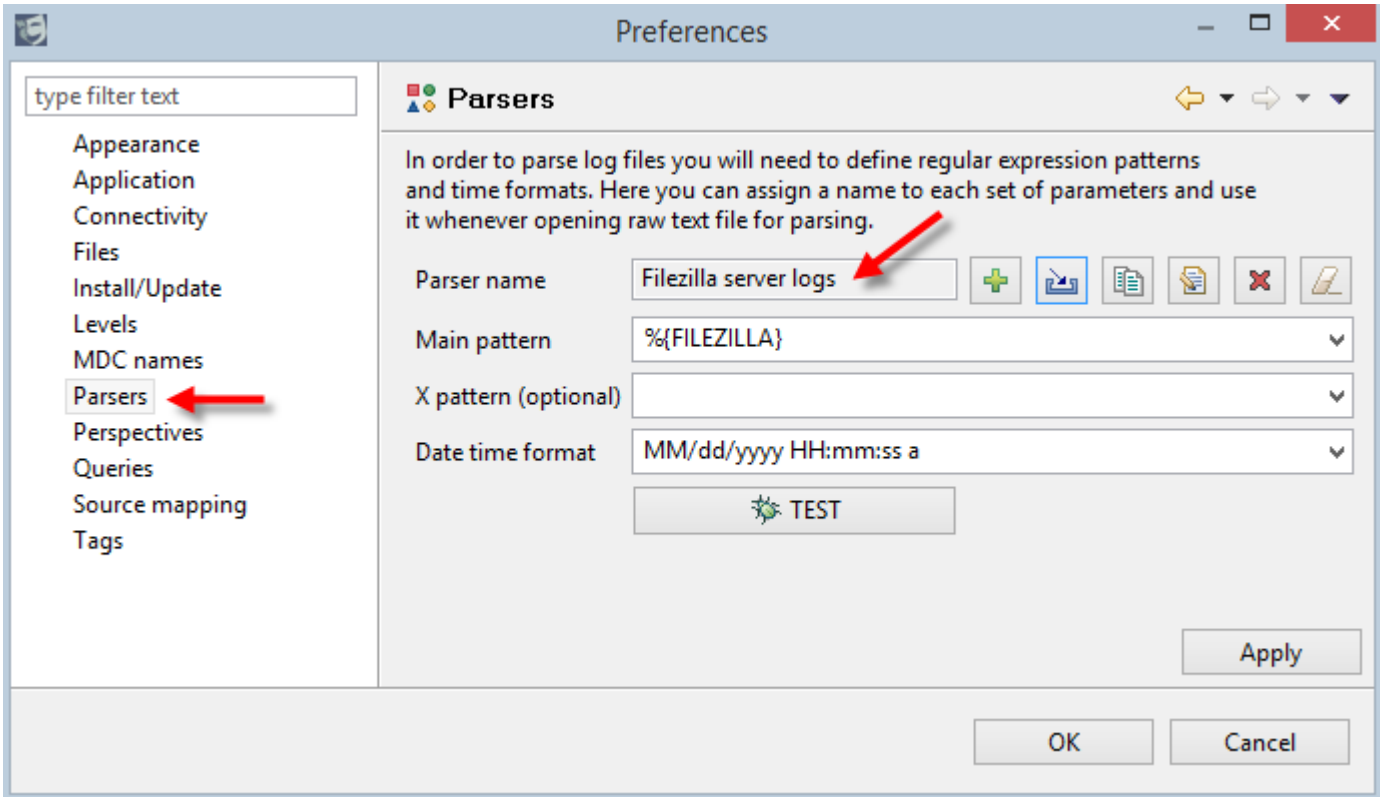
In order to be able to use this information we need to map the locations of source files – this can be done in **File/Preferences/Source mapping**. Whenever the source file is requested, the mapped locations are scanned to find the corresponding source file. If source file is found, it will be displayed in a separate window pointing to exact location in the code where the log statement originated.



Note that remote source repositories are specified as HTTP URL's. Depending on your repository you may want to include authentication parameters in URL itself. If required, client may prompt you to enter user name and password for accessing the remote storage. Credentials are cached and stored as part of the current workspace. Normally you will be prompted only for the first time when remote location is accessed.

3.13 Viewing raw log files

It is possible to parse log files and use client analytical features to inspect the content. This is done by means of parsers which you can define in your workspace and use them whenever log file needs to be opened. Parsers are defined in preferences:



Each parser defines main regular expression pattern, exception pattern and date time format specific to the log files you will be using this parser with. Each parser is also named so that you can easily recognize it. Each parser can also be tested by challenging it with real log data.

Once you are done defining your parsers, they will be listed under **"File / Open log file.."** menu. Processed file will be displayed in a conventional data table where you will further apply filtering, do the search, spread and slice by categories and use the rest of the analytical tools available in client interface.

To learn how to use regular expressions to parse raw text please refer to "Working with regular expressions" section. Note that clients can share pattern libraries through the server and use the same set of regular expression patterns.

3.14 Managing patterns library

Each client maintains its own copy of regular expressions patterns library and using it for parsing log files when needed. To see the library content go to **Tools** menu and select "**Open Patterns Library**", the local copy of patterns library will be displayed.

Patterns can be modified locally, imported from the server, tested and exported to server for sharing with other users. To learn about patterns library please refer to "Working with regular expressions" section.

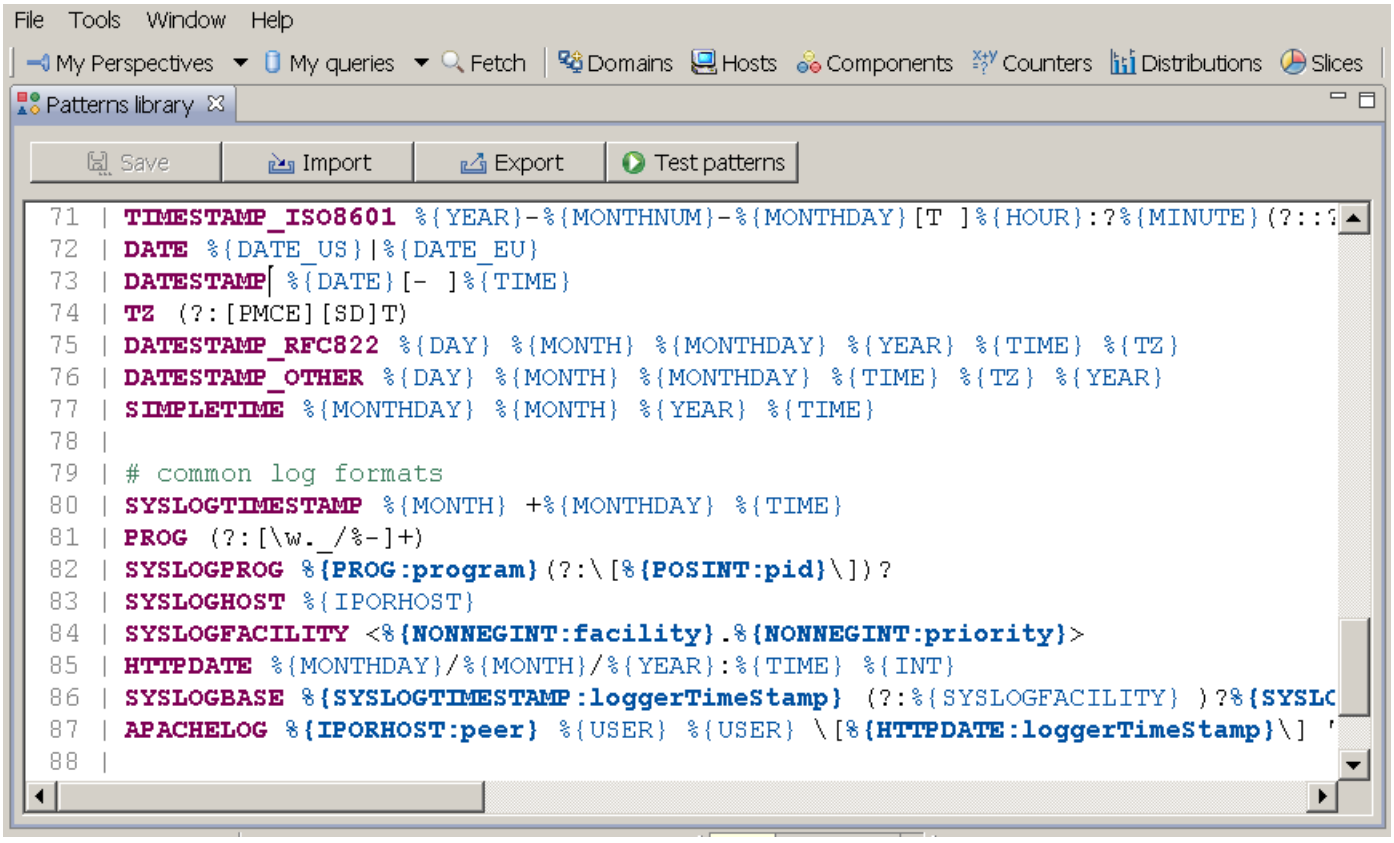


Figure 3.14.1: Patterns library

3.15 Importing and export logs

Log data can be exported from one server database and imported into the other, we use binary data image for those operations. You can export query results directly into binary file and import it later on somewhere else.

To export all data from the database, go to Tools menu and select "**Export data from database**". You will be prompted for the file name. This operation is identical to what is done by Backup utility.

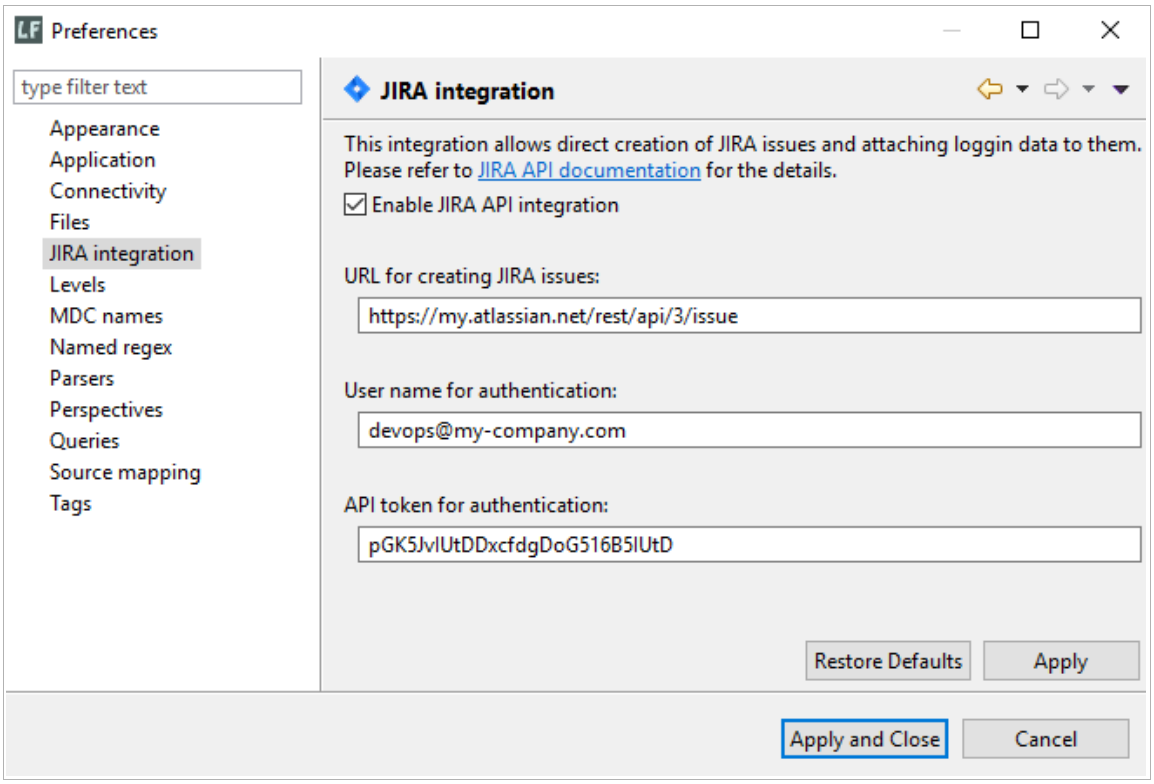
To import data into database, go to Tools menu and select "**Import data into database**". You will be prompted to select a file exported previously from logFaces. Before the import actually starts, you will be asked to specify a Domain name to which imported data should be associated:

If you leave this field blank, the data will be imported without any modifications. Otherwise, the Domain name of all events will be replaced to the value provided. This is very convenient when you exchange data between different logFaces servers.

Please take into account, that depending on the data set size and the speed of the database, those operations may take up to several minutes while taking considerable amount of CPU and database I/O.

3.16 JIRA integration

Integration with bug tracking systems can save plenty of time when you spot and report issues directly from the data you see in logFaces. JIRA is one of the popular issue tracking platforms and logFaces can directly create issues using its REST API. To enable JIRA integration, go to File/Preferences and specify the URL, user name and API token:



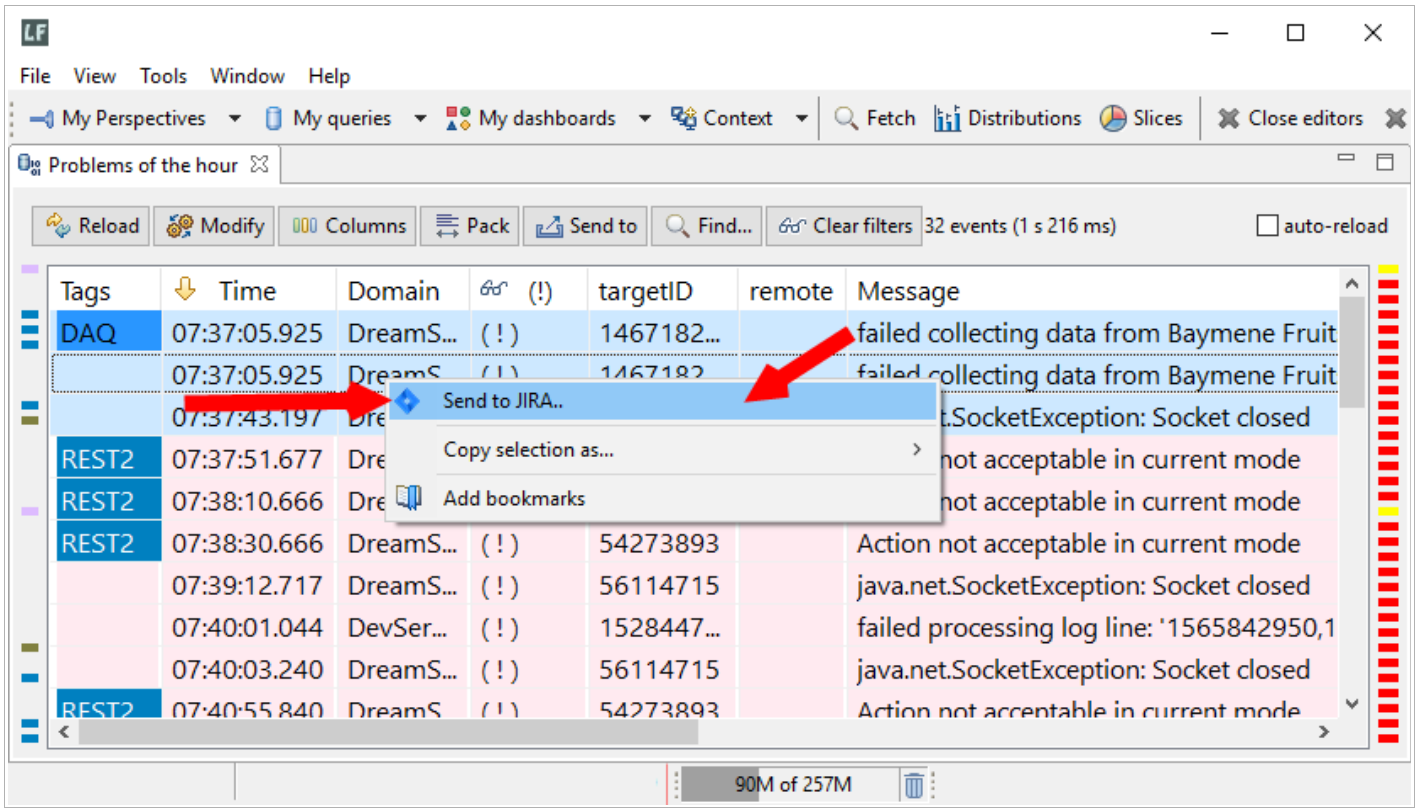
As of this writing the issue creating end point is currently available at the following URL:

```
/rest/api/3/issue
```

Note that JIRA expects user names in the form of actual **email** addresses.

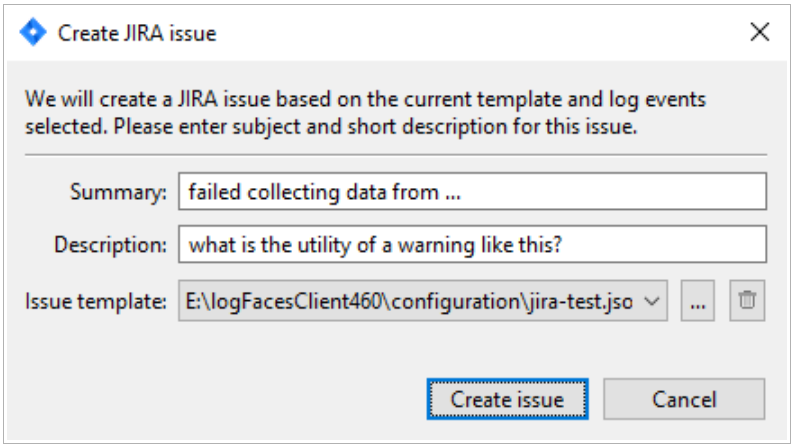
The API token can be obtained for individual user from JIRA administration site, note that the token must match the user name used above.

Once JIRA integration is enabled, you can start creating issues directly from logFaces data tables, either queries or real-time. When you see something of importance, just select the events and using the right click choose “Send to JIRA” action.



You can select single log statement or multiple. Whatever is selected gets converted into a log file and attached to the JIRA issue. The format of the log files can be configured in preferences **File** section. Same format is used with `${logs}` variable in the template explained below.

JIRA issue creation is a complex API, but we simplify it greatly by requesting issue Summary and Description only. The rest is defined in re-usable issue template(s):



The issue template is a JSON body sent to JIRA servers with the issue creation request. You will create template(s) in accordance with JIRA settings of the project where issues directed at. For that you will need to use JIRA documentation and have knowledge of your project structure.

Below is an example of an issue template. You can have few templates if there is a need to create issues in different projects.

```
{
  "update": {},
  "fields": {
    "summary": "${summary}",
    "issuetype": {
      "id": "10010"
    },
    "project": {
      "id": "10012"
    },
    "description": {
      "type": "doc",
      "version": 1,
      "content": [
        {
          "type": "paragraph",
          "content": [
            {
              "text": "${description}",
              "type": "text"
            }
          ]
        },
        {
          "type": "paragraph",
          "content": [
            {
              "text": "${logs}",
              "type": "text"
            }
          ]
        }
      ]
    },
    "priority": {
      "id": "3"
    }
  }
}
```

The template file must be a **valid** JSON structure defined by JIRA API and contain references to **real** JIRA identifiers of projects, types, priorities, etc. Note how we use `${variable}` placeholders - these are getting evaluated by logFaces before the JIRA call. The `${summary}` and `${description}` get populated from the prompt dialog above. Optionally the `${logs}` variable can be used to append **first few lines** of the selected log statements into the body of the issue, sometimes it's a convenient option for laying out the issue view.

3.17 Command Line Arguments

3.17.1 logfaces.ini file

Client applications use a set of arguments to bootstrap their JVM. Those arguments are located in a file named **logfaces.ini**. Its location is platform dependent. On Windows and Linuxes it will be found under client installation directory. On Mac OS it will be located inside the application archive: /Logfaces/Contents/eclipse/logfaces.ini.

Do not remove any of the default settings because it may break the bootstrap sequence. Below are commonly used arguments which can may be modified or appended.

Using non-default JVM is possible by specifying **-vm** argument (note that argument name and value are passed in separate lines!)

```
-vm
path-to-javaw.exe
```

The following parameters specify JVM heap memory sizes. Increase the Xmx parameter for more memory intense scenarios.

```
-vmargs
-Xms64m
-Xmx256m
-XX:MaxPermSize=128m
```

In case your desktop computer is using proxy to connect to the outside world, add the following two lines with your own settings for proxy host and port.

```
-Dhttp.proxyHost=host
-Dhttp.proxyPort=90
```

If you use SSL to connect to logFaces server and your CA is not in the default trust store of client JVM, you can instruct client JVM to use another trust store containing your certificates by specifying its path and password using these properties:

```
-Djavax.net.ssl.trustStore=path-to-trust-store-file
-Djavax.net.ssl.trustStorePassword=your-password
```

3.17.2 Branding

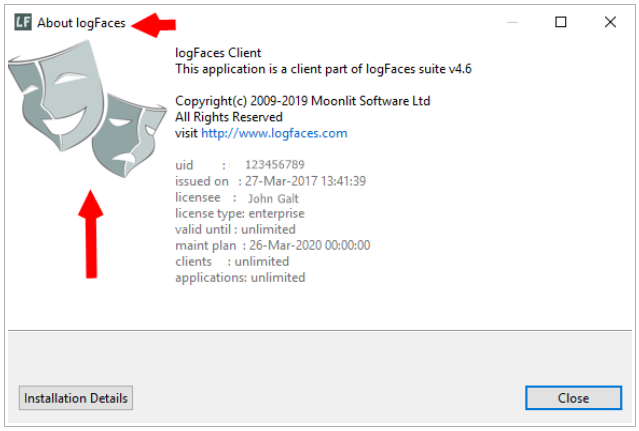
It is possible to customize the splash screen so that it shows your own background image. Make the following lines to be the first the logfaces.ini file:

```
-showsplash
c:/full-path-to-image/splash.bmp
```

Make sure that image file is of **BMP** format and is exactly 455 x 295 pixels. The login screen controls will overlay the background image, so you want to design your background image to coexist with the dialog controls placed on top of it. Here is the current built-in splash screen for the reference:

It is also possible to customize title and image in about dialog. Append the following properties to the logfaces.ini file:

```
-Dcom.moonlit.logfaces.title = "My custom title"
-Dcom.moonlit.logfaces.image = /full-path-to-image/my-image.png
```



3.17.3 Launching workspace on start-up

It is possible to launch the client so that it automatically loads the workspace by name or by the file name. To do this, use **-workspace** argument which is followed by workspace name or the file name. If name has spaces, make sure to decorate it with quotes. For example:

```
logfaces.exe -workspace "My workspace"
```

```
logfaces.exe -workspace "c:\logfaces\My workspace.lfs"
```

3.17.4 Launching queries on start-up

It is possible to launch the client so that it automatically runs a query before showing up. To do this, use **-query** argument which is followed by **semi-column** separated query parameters. None of the parameters is mandatory (see defaults below). For example, the following command line will display first 100 exceptions with ERROR and above severity level :

```
logfaces.exe -query name=Problems;thrown=true;loggerLevel=error
```

Here is the full list of parameters which can be used:

Parameter	Description	Default
name	Query name, will be displayed on the editor tab	My query
fromTime	Include data from this time (UTC numeric long in msec)	0
toTime	Include data until this time (UTC numeric long in msec)	MAX_LONG
domains	Comma separated list of application names to match	-
hosts	Comma separated list of host names to match	-
loggers	Comma separated list of logger names to match	-
exception	Text to match in stack traces	-
matchMessage	Text to match in event message	-
onlyThrown	True for including only thrown exceptions	false
level	Log4j level, numeric or strings (INFO, ERROR, etc)	TRACE
limit	Maximum numbers of events to fetch	100
timeZone	Which time zone to use for display	JVM time zone

Table 3.1: Command line parameters

3.18 Preferences

Global application preferences are accessible through the File menu and allow the following settings. In **Appearance** section you can customize fonts and colors of main application views and editors.

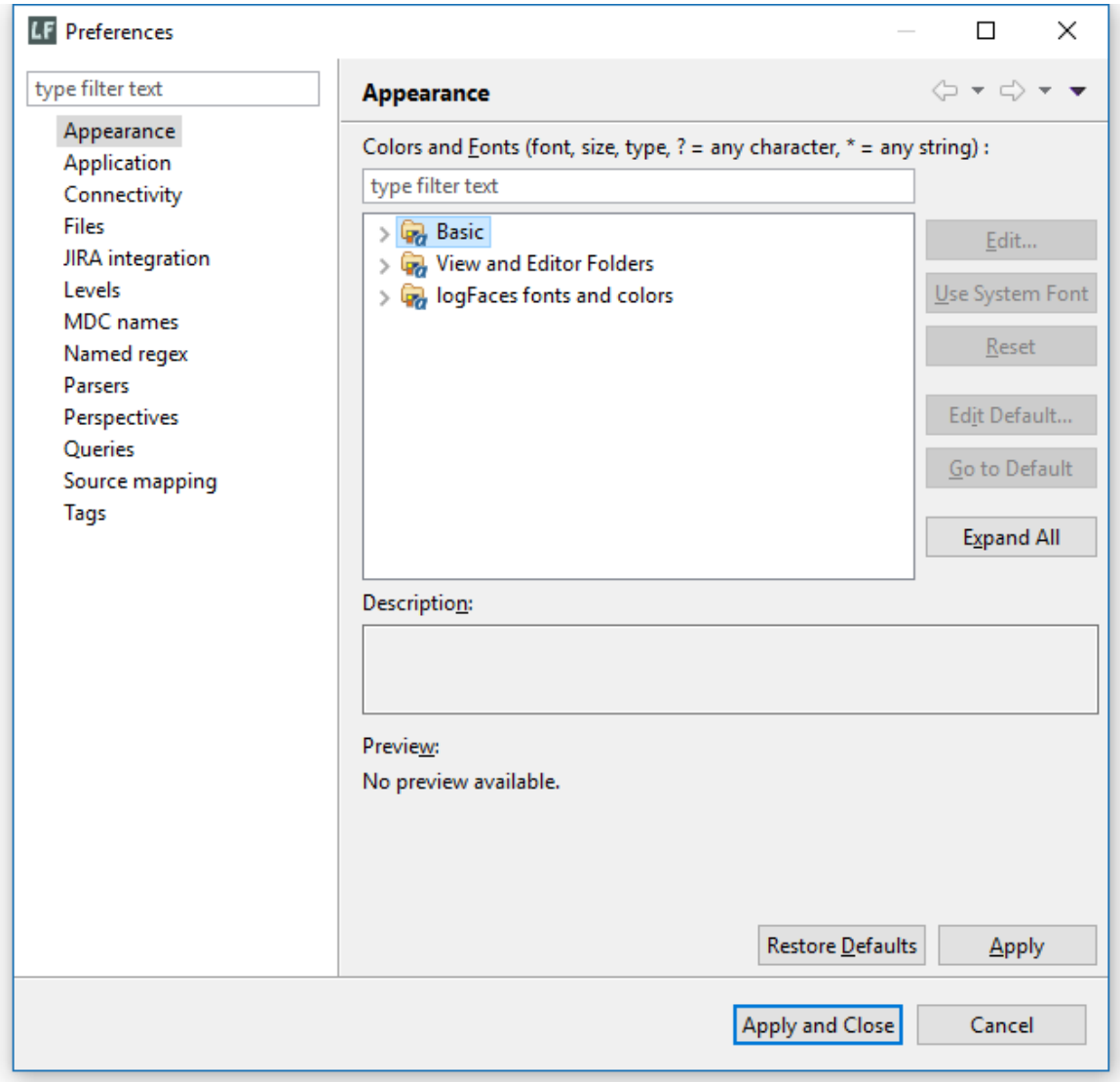


Figure 3.18.1: Appearance

In **Application** section you can specify whether the client main window should be minimized to tray or stay minimized in the task bar so that you see the title. When application is minimized to tray, double clicking on its icon will restore its main window.

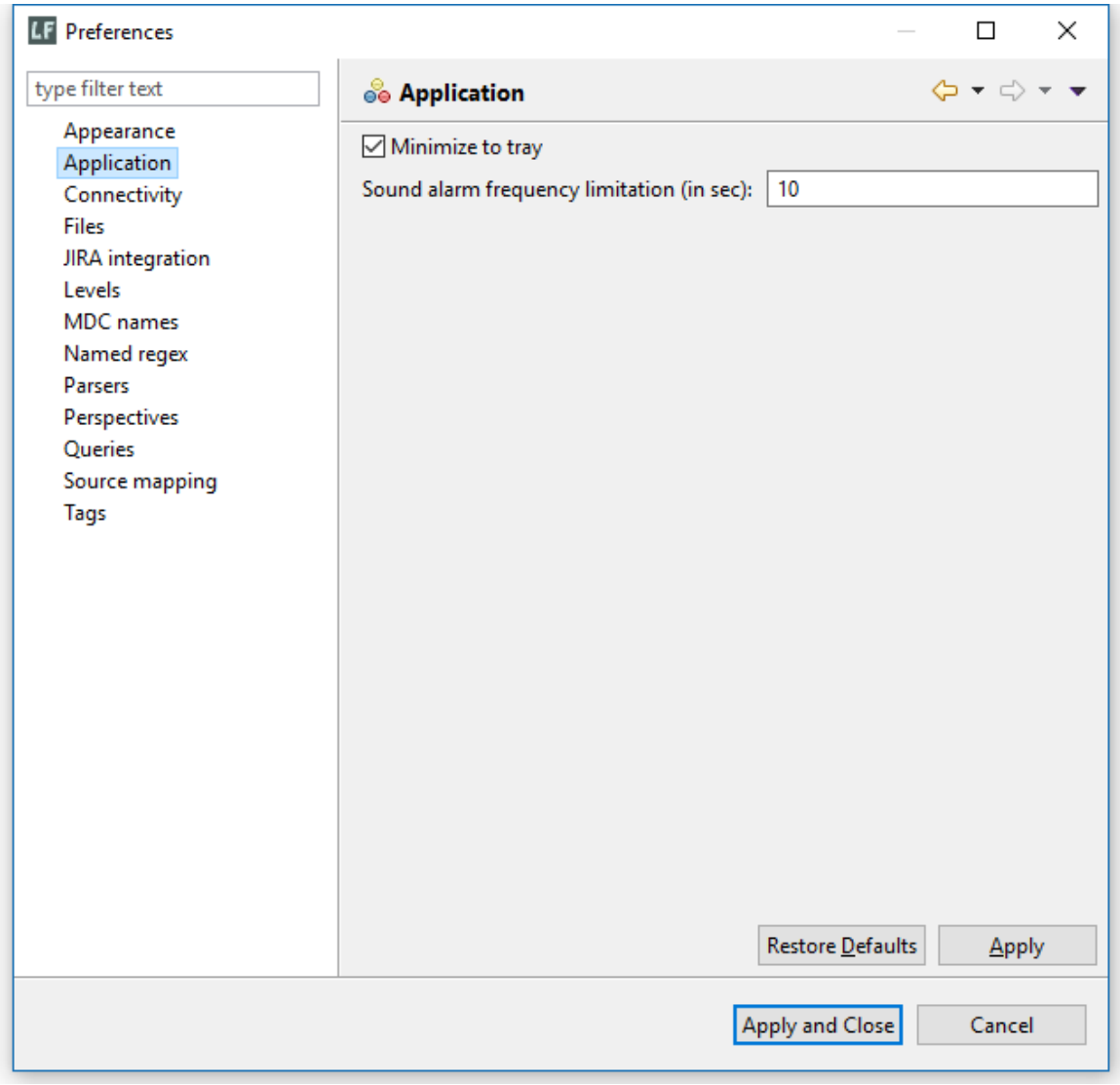


Figure 3.18.2: Application preferences

In **Connectivity** section you can specify the server connection endpoints which apply to a currently used workspace.

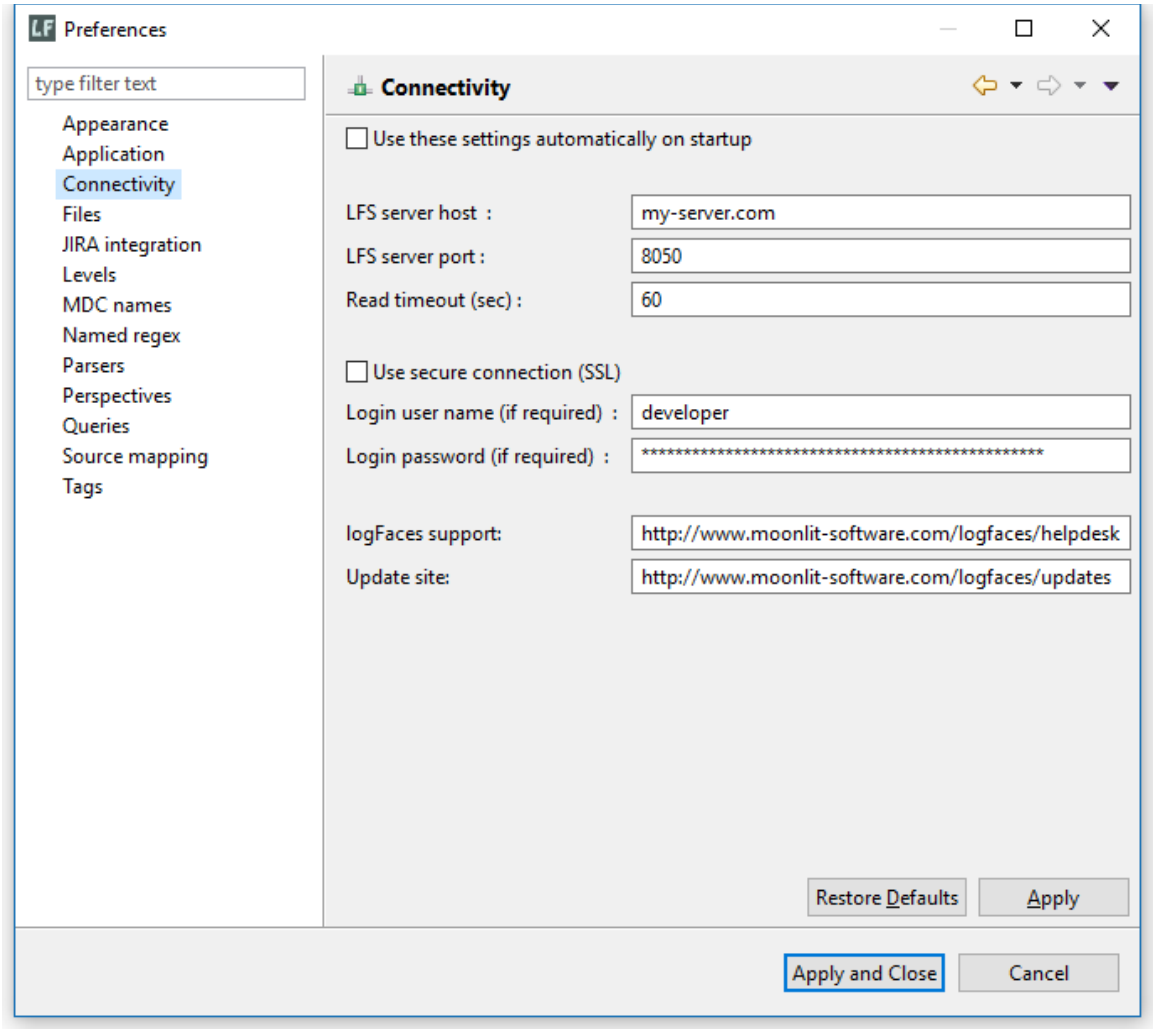


Figure 3.18.3: Connectivity preferences

Support site URL is used for submitting bugs, feature requests or questions – you will be taken to this URL automatically when selecting **Help/Report bug or request feature** menu.

In **Files** section you specify the layout format of log files and external editor to be used throughout the application. The layout format will be applied to all file related operations when saving data to a file or copying logs into clipboard. The format is specified in Apache Log4j documentation, you might want to read about it here.

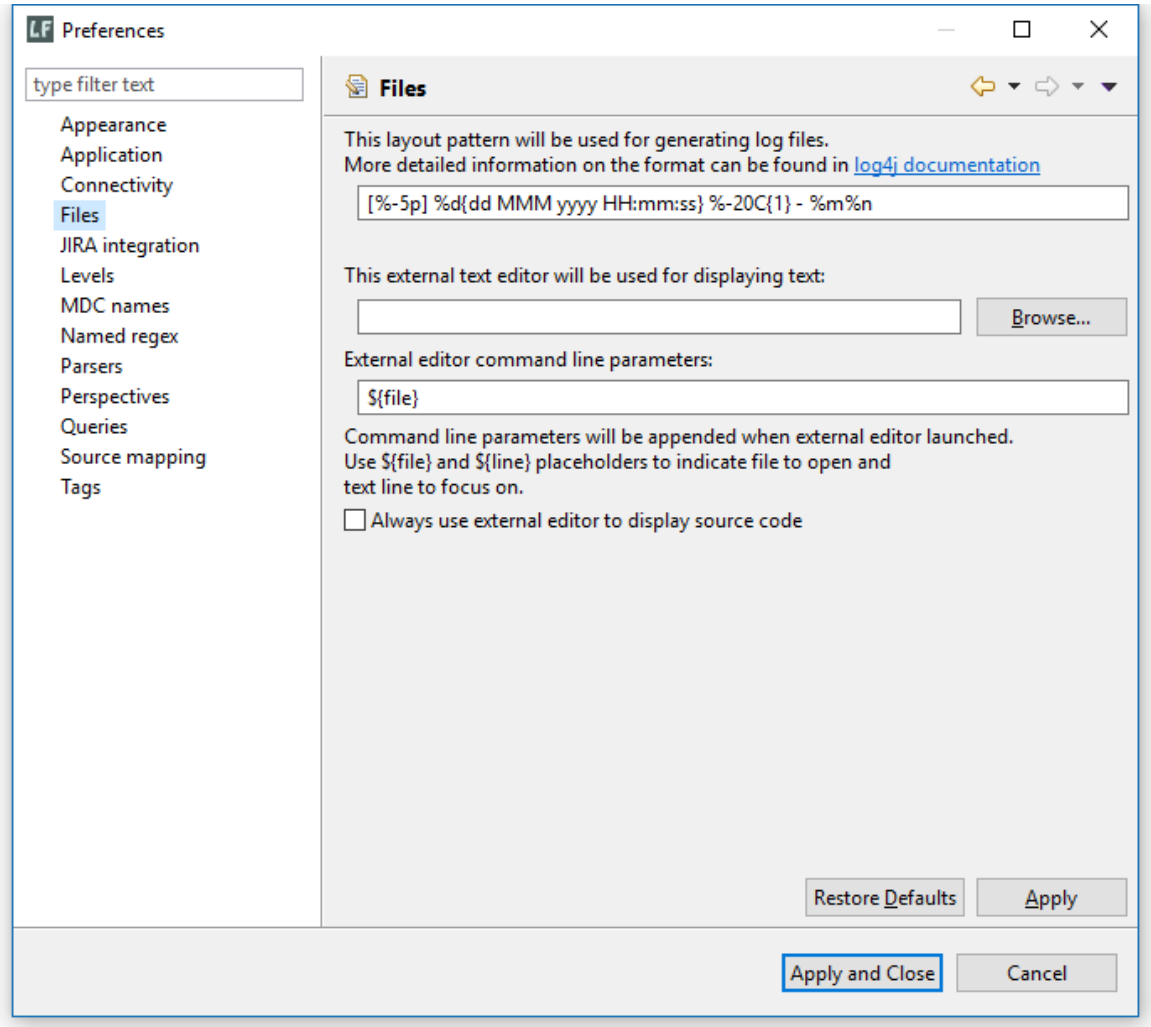


Figure 3.18.4: File preferences

Note that external text editor can also be used for displaying the source files. By default sources are displayed in internal source viewer, but you can change this and have your own editor opened whenever sources are displayed. You can also use ${file} and ${line} variables to construct a command line for your editor.

JIRA Integration allows creation of issues in JIRA projects directly from logFaces context with actual log statements attached. Integration is based on JIRA REST API which will be called on your behalf as developer. For this to work we need the URL for creating issues, the user name (usually an email) and the API token which obtained from your JIRA administration site. Once enabled, you should be able to report issues directly from real time perspectives and queries with a simple right click of a mouse.

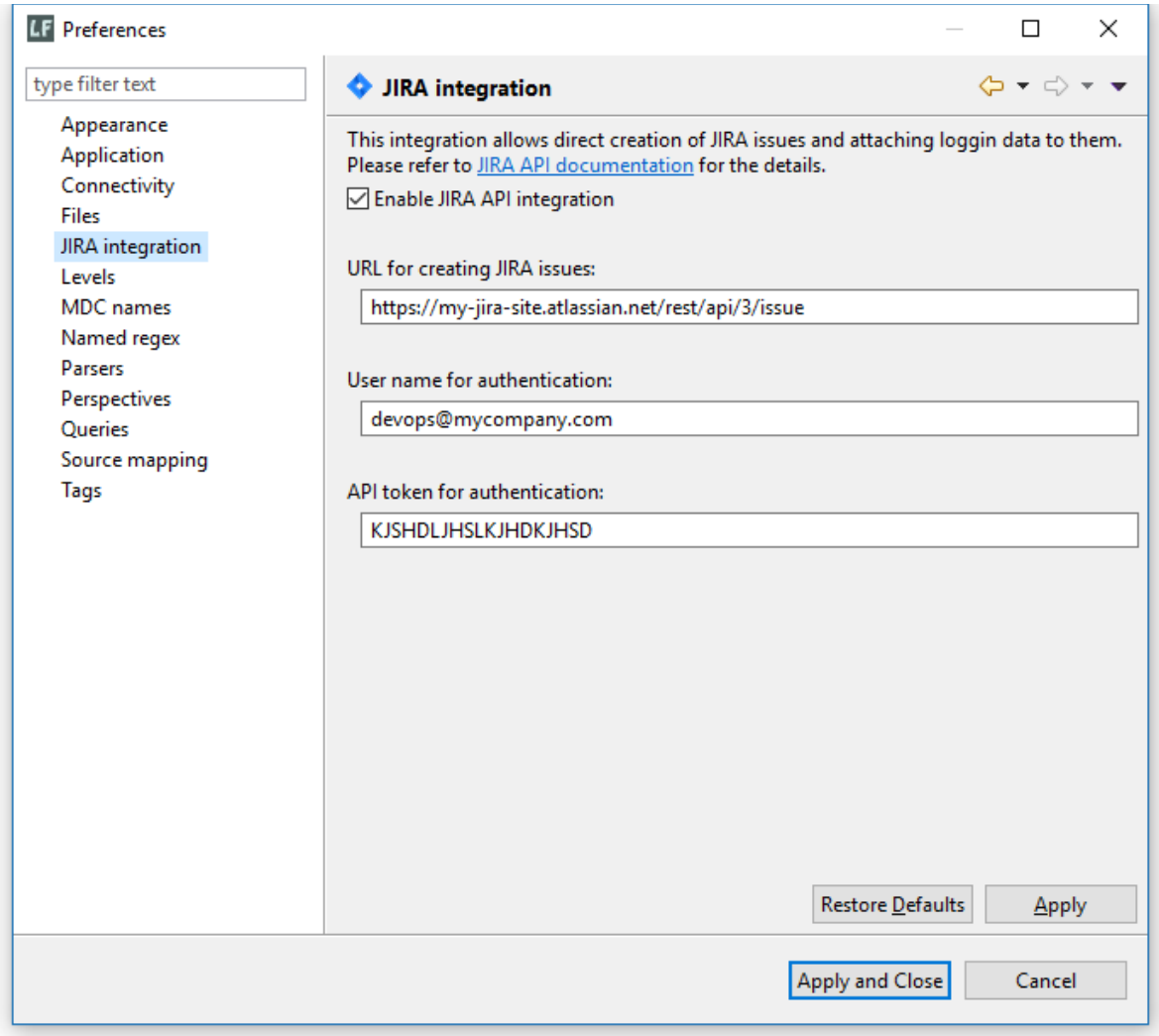


Figure 3.18.5: JIRA integration

In **Levels** section you can customize the way log statements look – background or foreground colors and image icons. Note that some axillary views are using the colors you define here.

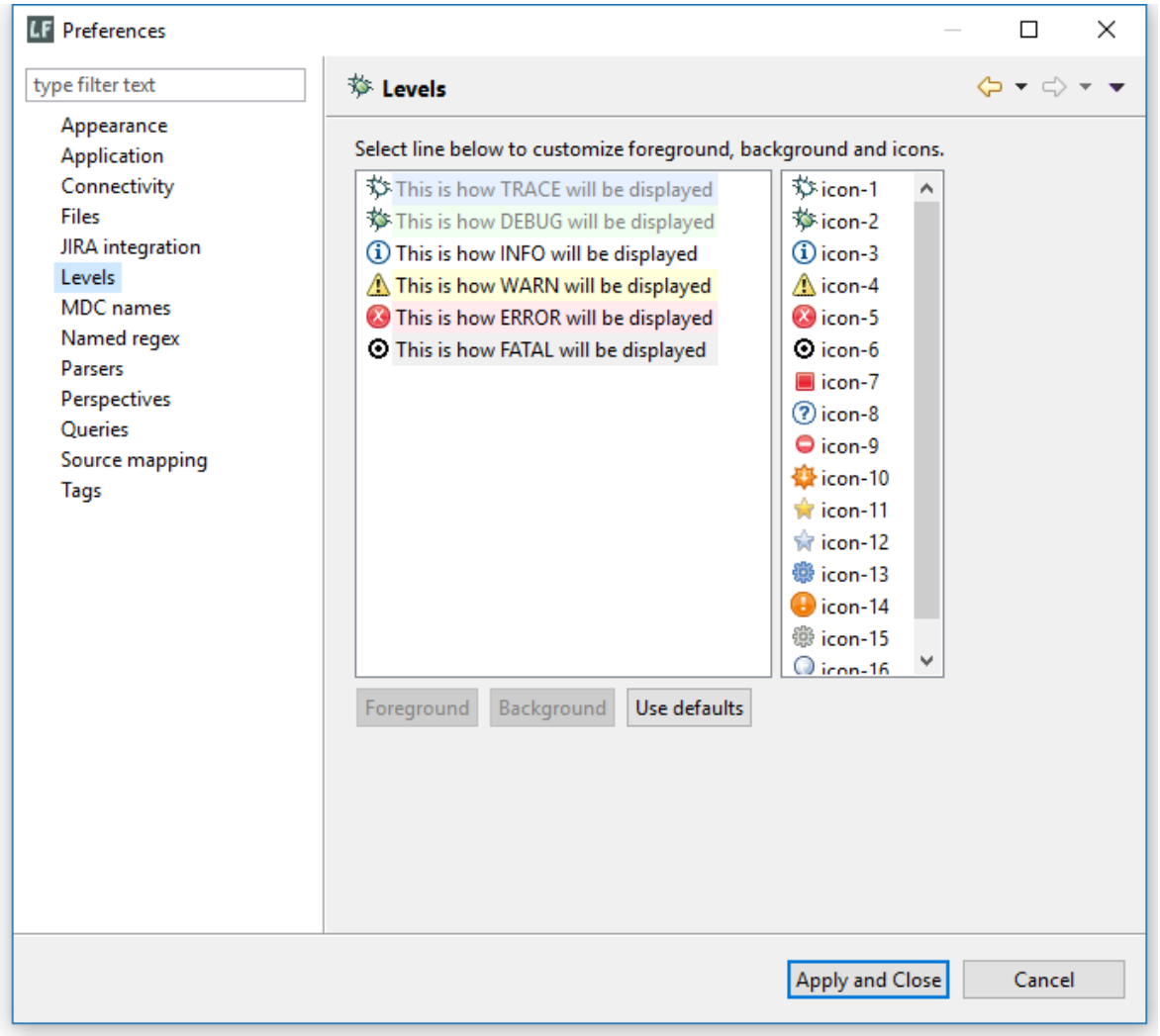


Figure 3.18.6: Severity level styles

In **MDC names** section you will find current MDC names mapped from applications to the logFaces server. MDC mapping is modified on server side, so in case you need to change the mapping, you will have to use server administration.

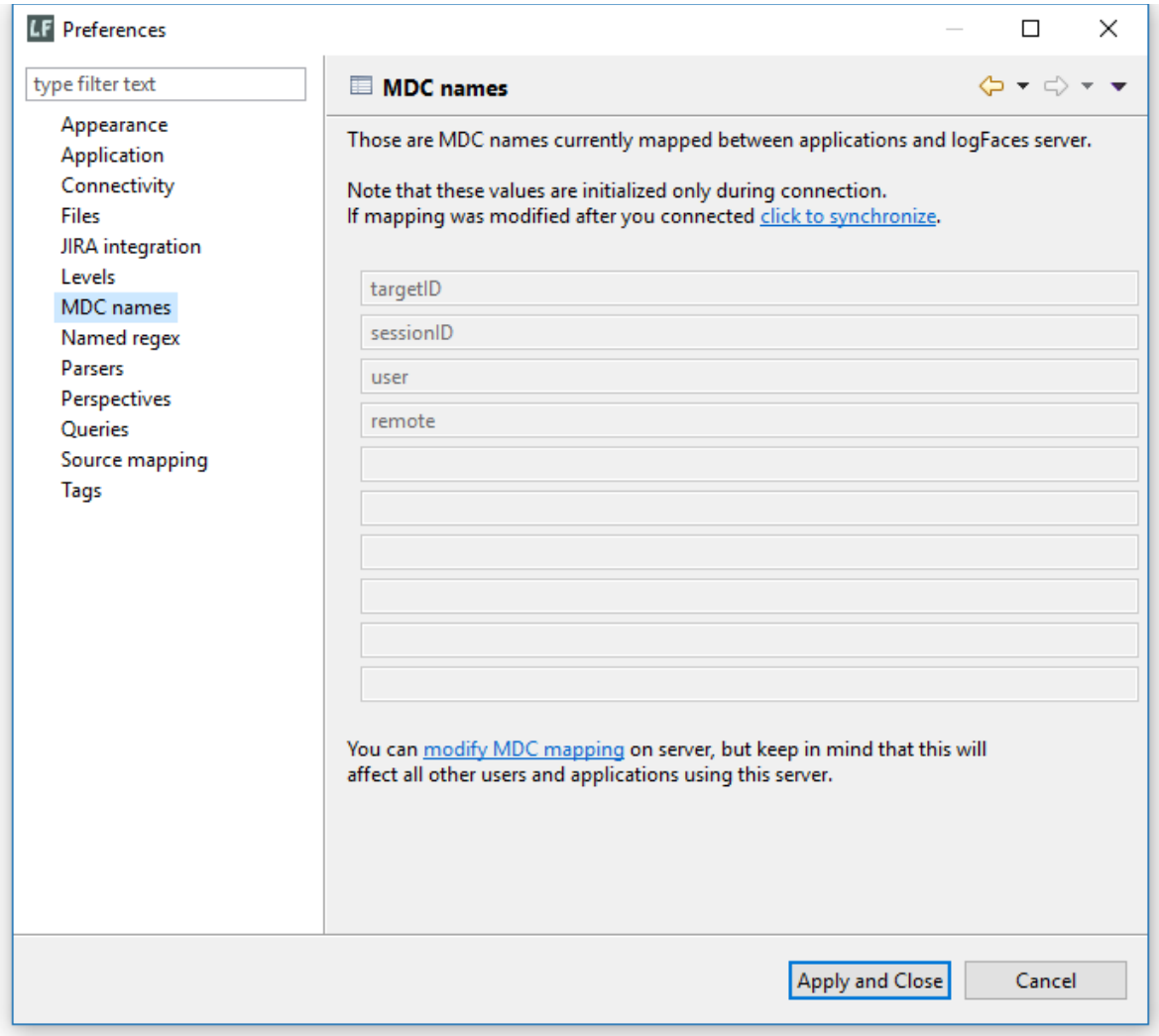


Figure 3.18.7: MDC mapping

When MDC mapping has some names, you will be able to use those names in queries and filters. For example, when SESSION_ID name is defined, it will appear in data tables column and you will be able to search events based on this name as well as trace real time data and filter data tables containing particular value of this attribute.

Named regular expressions are used with log analysis, they define custom categories which then used to slice and spread the log data. For more details see “Named regular expressions” section.

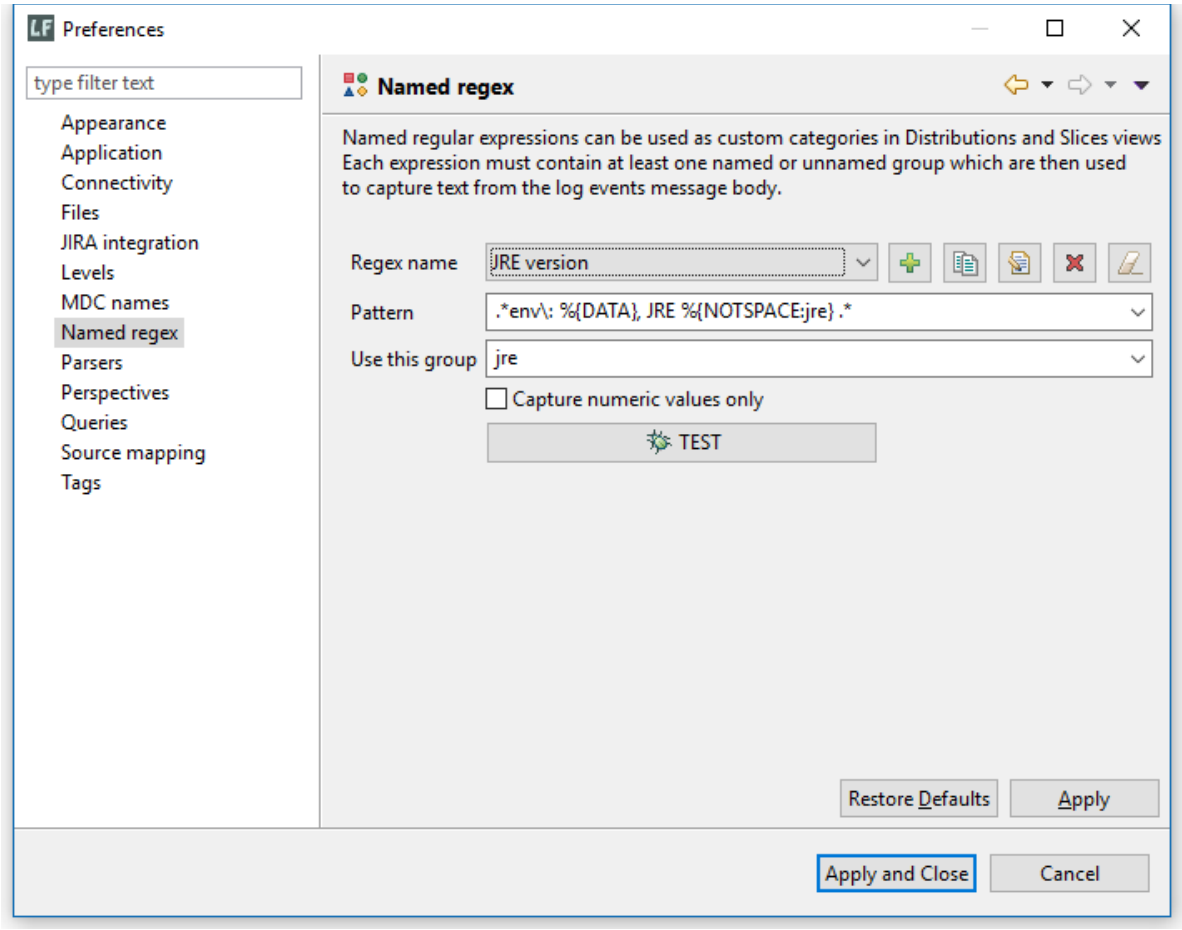


Figure 3.18.8

In **Parsers** section you will be able to define parsers for processing log files offline (without server). This feature is designed to exploit client analytical capabilities to display and search directly by opening any text file provided that its format can be parsed with regular expressions. More details can be found in “Viewing raw log files” section .

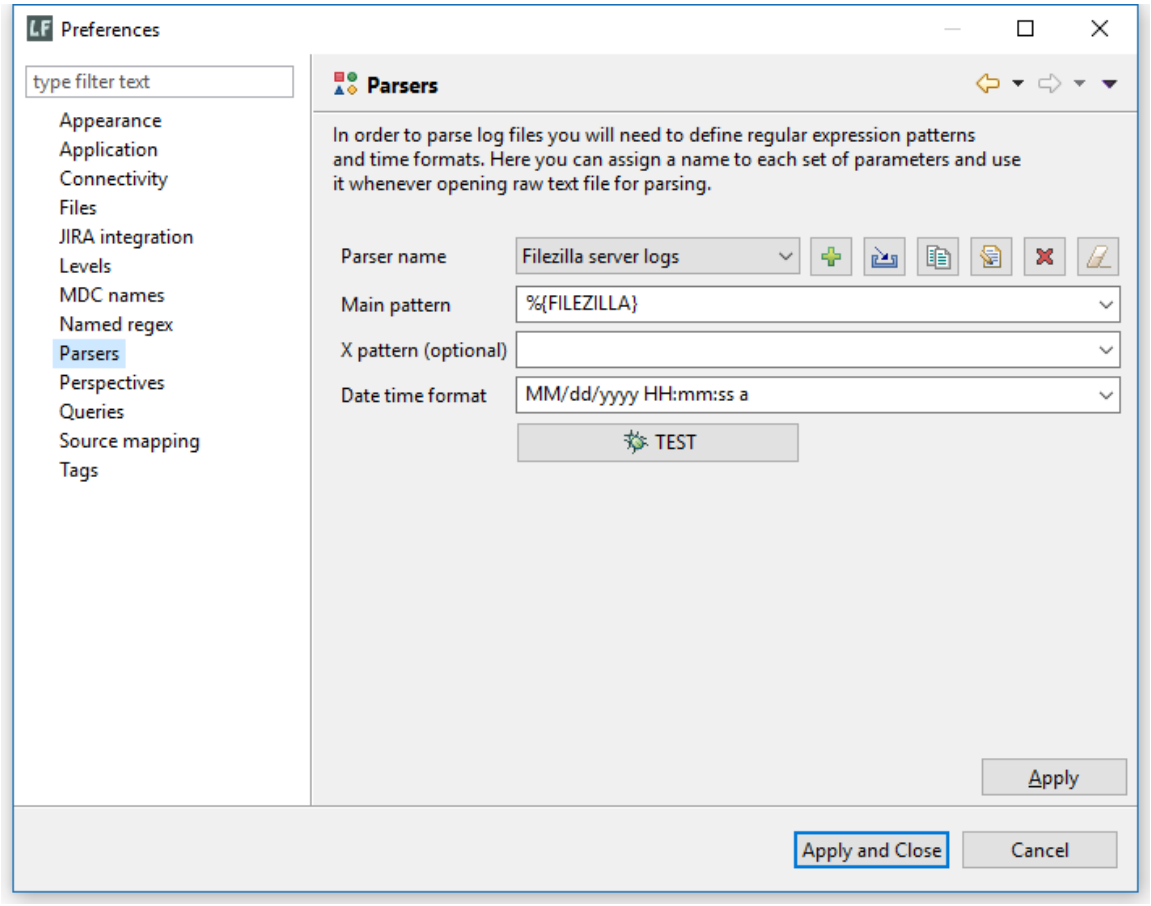


Figure 3.18.9: Parsers

In **Perspectives** section there are ways to customize real-time perspective views. When real-time view is opened, there is an option to grab historical data from database before the view gets live – this will give you the context for current log events coming into the view.

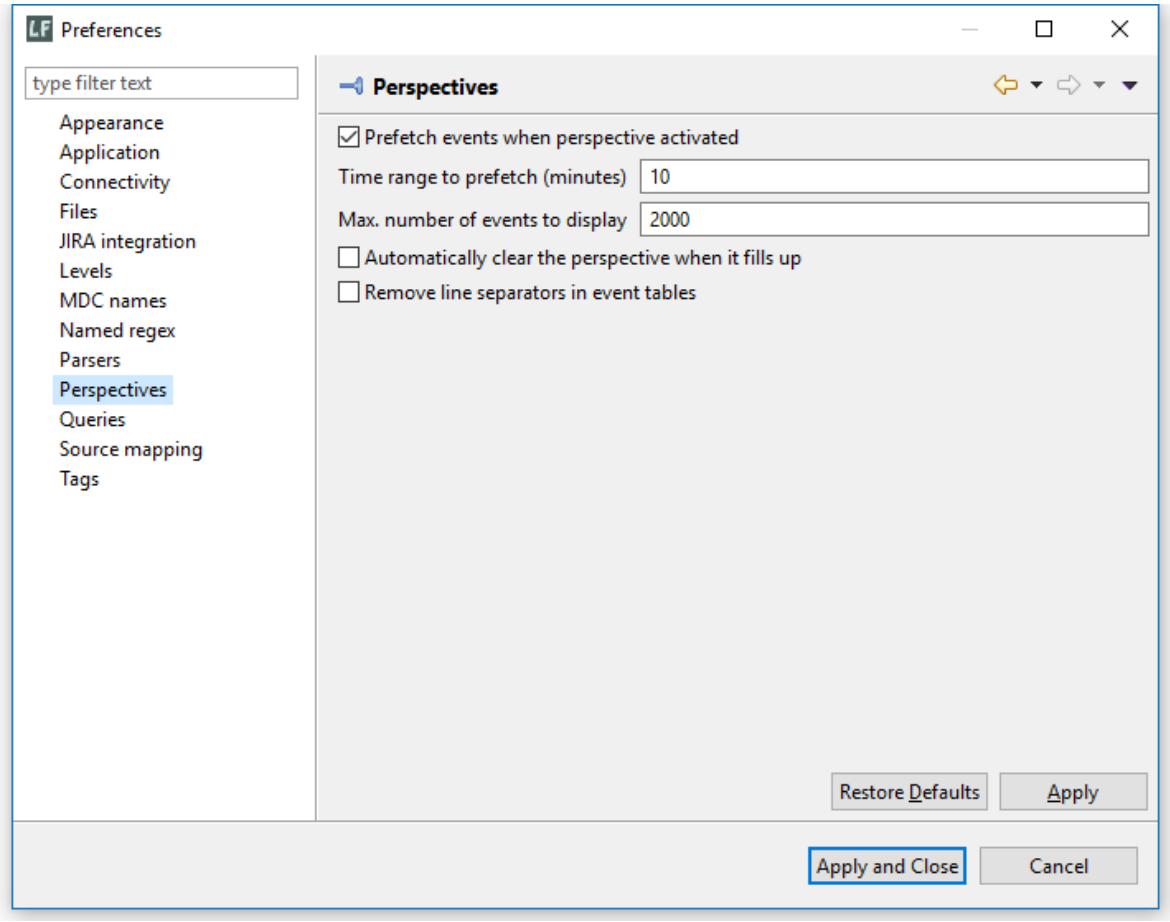


Figure 3.18.10: Perspective preferences

You can limit the size of the view to a certain amount of events, when view gets full it starts rotating by removing older events. You may want to clear up the view entirely when it fills up. Some messages may have End Of Line separators – this can be optionally removed to fit the message nicely into the table.

In **Plugins** section you can manage the plugins - adding, importing, removing and synchronizing with server. Plugins listed here will be available in the context menus of data views.

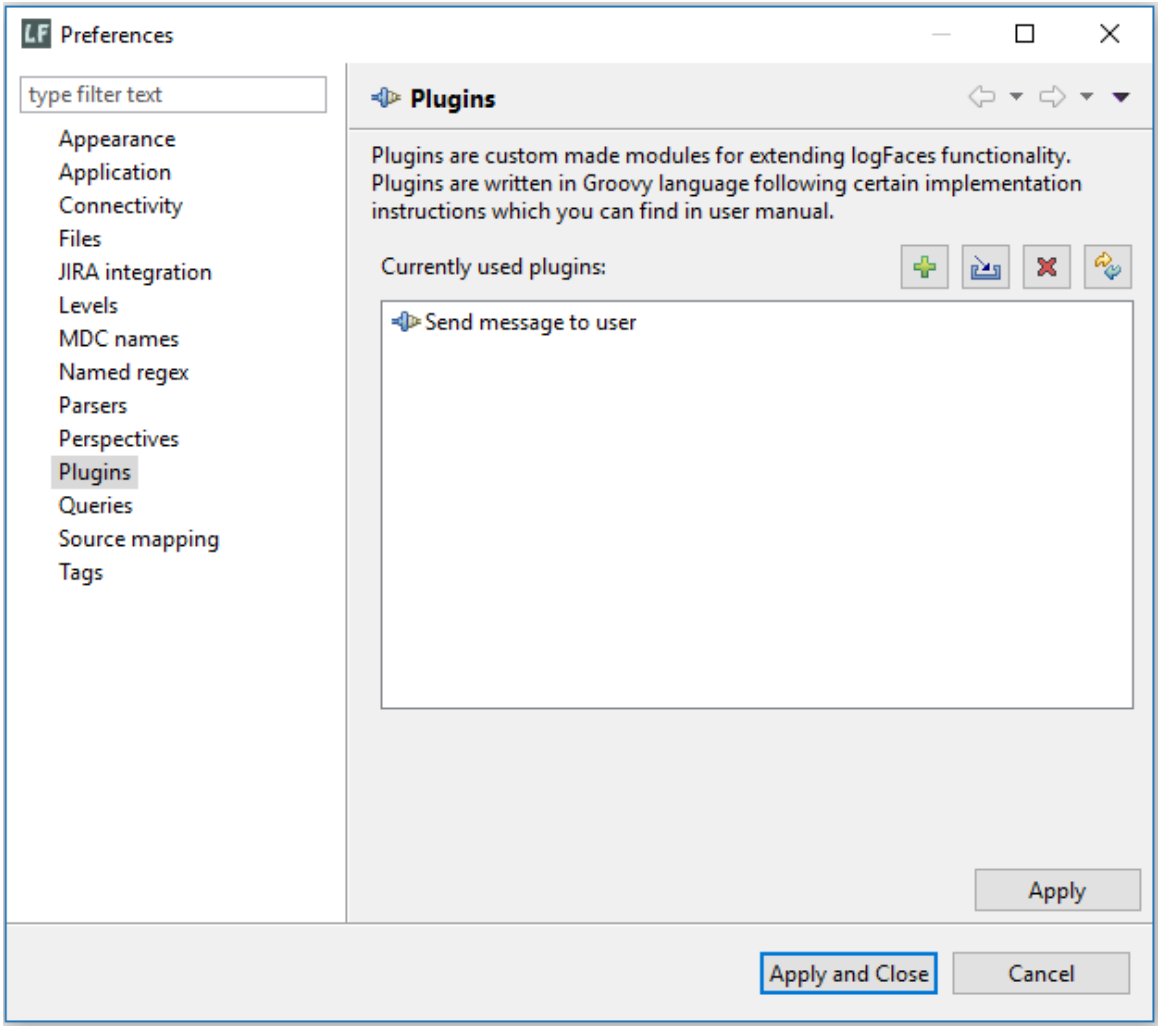


Figure 3.18.11: Plugins management

In **Queries** section you can specify the way queries are executed against the server. These are global settings and applied to all queries throughout the client.

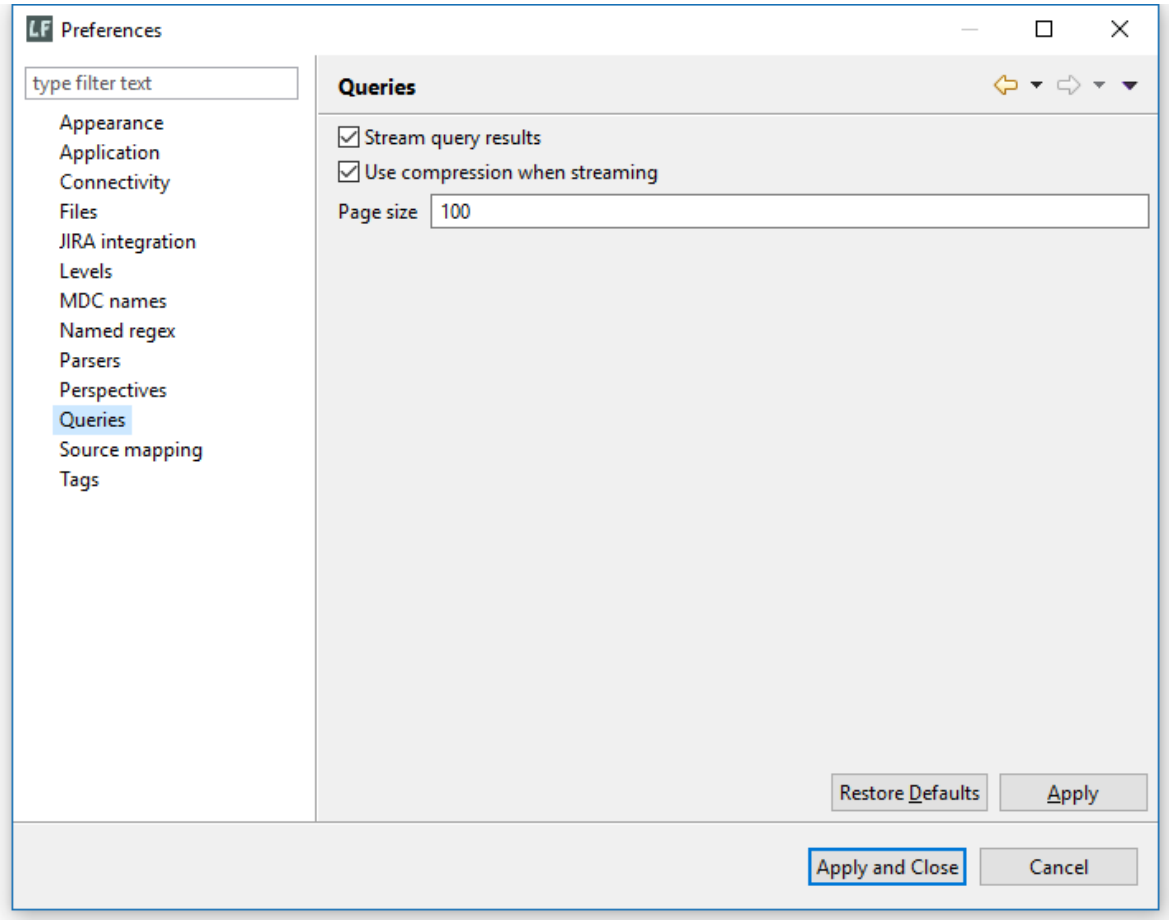


Figure 3.18.12: Query preferences

If you ever used logFaces over the public internet or VPN, you might have noticed that large result sets are slow to get to the client even with a well tuned database. It may get particularly painful with huge queries containing exception dumps, which is often what people do. This issue is addressed by two options you can try here - “Stream Query Results” and “Use Compression”.

The slower the client-server connection, the more vivid will be this improvement. Now, instead of paging the result sets and hitting the server on every page, we do streaming which takes a single server hit and delivers everything to the client in one shot. The stream may get compressed. Plus it will bypass many conversions while streaming simple JSON text.

In **Source Mapping** section you can specify where to look for actual source files. References to source files (if enabled in your appenders) will be available for every log event as well as exception stack traces. So, in order to jump into a source code directly from logFaces client, this mapping is essential.

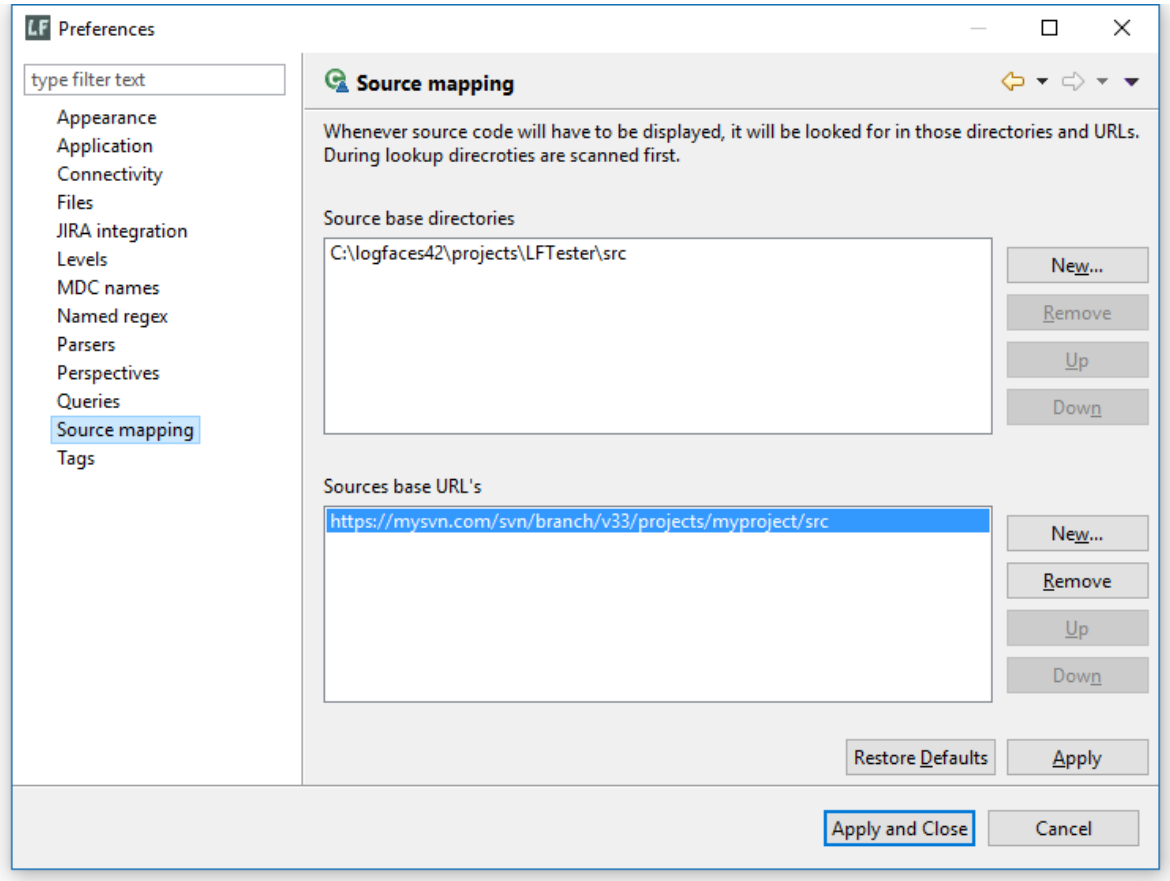


Figure 3.18.13: Sources mapping

When source file is being resolved, client first tries to locate the file in directories in the order you specify. If not found, the client will try to locate the file in one of the URL's. The URL must be HTTP based and contain the base for file locations, client will append relative file path to the URL during look up.

Tags add domain specific indications to a dull technical log stream. You create a tag by giving it a short friendly name, color and matching criteria. Logs matching the tag criteria get tagged before they get displayed. The criteria is the same bunch of rules we use in queries and filters, nothing new here. Tags are a very powerful tool - they participate in view filters, queries and analytical charts.

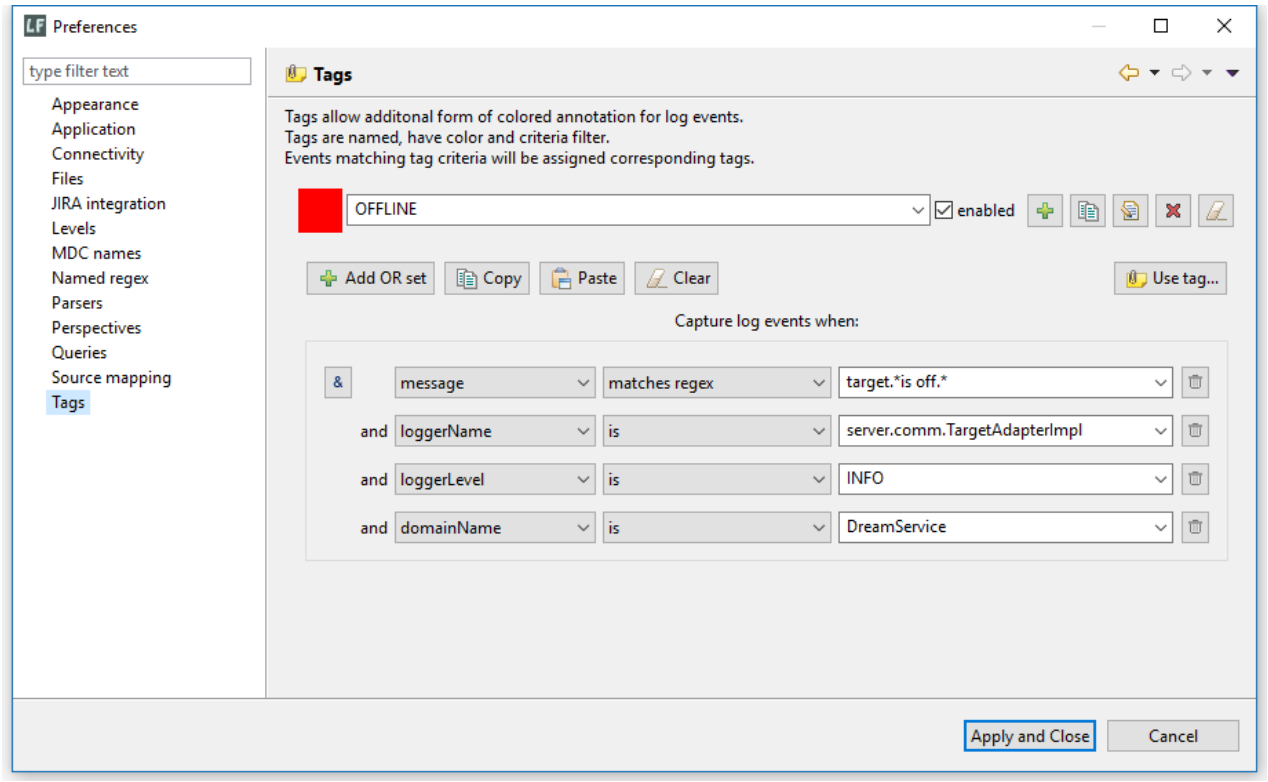


Figure 3.18.14: Tags preferences

3.19 Status bar

Status bar displays the following information:

- current connection state
- current connection end point
- current version of logFaces server
- current license
- number of database records (click on the icon to refresh the counter)
- current RAM used on client versus maximum RAM allocated plus manual garbage collector.

Figure 3-3.19.1 Status Bar

4 REST API

logFaces server comes with an industry standard REST API to allow direct access to the most common tasks such as queries, real-time monitoring, access to repository data, etc. The API consists of several HTTP calls with JSON payloads and parameters.

4.1 Access control

Access control is enforced when authentication is enabled in server administration. Otherwise, anyone can do server calls without restrictions.

When authentication is enabled, the caller is expected to login prior doing any other server calls. REST API access control is based on access tokens which are issued on each `/login` call and then expected to be returned to server with subsequent calls in Authorization header.

Along with authentication, tokens may carry authorization information. This gets into effect when logFaces server runs with authorization enabled. If authorization is not enabled, then every authenticated user is treated equally as 'anonymous' role bearer.

Access tokens are durable for 24 hours and should be re-acquired when expired. It is not required to perform `/logout` all the time, but if done so - the token is revoked instantly and can't be used any longer. Server will respond with HTTP code 403 if it fails to validate the token. Here is the format of the calls:

Request	`POST /api/login?user=name&password=pass`
Response headers	`Content-Type : application/json`
Response code	200 - logged in ok 403 - access is denied 500 - server error

Payload	{
	token : 'askdfhlkajhsdlkfjhalskdfj.....'
	}

Request	`POST /api/logout`
Request header	`Authorization : 'askdfhlkajhsdlkfjhalskdfj.....'`
Response code	200 - logged in ok 403 - access is denied 500 - server error

4.2 Server info

Do this call to obtain general information about the server such as server version, OS type, license, database type, driver, etc:

Request	GET /api/server/about
Request headers	Authorization : 'askdfhlkajhsdlkfjhalskdfj.....'
Response code	200 - logged in ok 403 - access is denied 500 - server error
Response headers	Content-Type : application/json
Response body	{ "licenseType": "site", "licenseHolder": "CN=xxx,O=yyy", "licenseID": "9827347238", "licenseIssueDate": "26-Feb-2014", "licenseMaintPlanCoverage": "26-Feb-2015", "serverVersion": "4.3.3.234", "osVersion": "Windows 8.1 (6.3), amd64", "javaVersion": "1.8.0_25", "dbProduct": "Apache Derby", "dbDriverVersion": "10.7.1.1 - (1040133)", "dbDriverName": "Apache Derby Embedded JDBC Driver", "dbVersion": "10.7.1.1 - (1040133)" }

4.3 Server health

Use this call for obtaining **authentication free** health state of the server instance. Can be easily integrated with external third party health monitors.

Request	GET /api/server/health
Response code	200 - instance is healthy and operational 503 - instance is up but not operational *** - instance health is not known
Response body	If instance is reachable the body will contain the current engine state: **idle**, **starting**, **running**, **stopping**. The instance is considered healthy and operational only when in '**running**' state

4.4 Server state

This call is provided for server state monitoring. The information provided by this call is identical to the one described in the admin status page:

Request	`GET /api/server/state`
Request header	`Authorization : 'askdfhlkajhsdlkfjhalskdfj.....'`

Response	`200 - logged in ok`
	`403 - access is denied`
	`500 - server error`

Payload	{
	"uptime": 1220164,
	"status": "running",
	"nofLastErrors": 0,
	"maxMemory": 1908932608,
	"totalMemory": 319291392,
	"freeMemory": 227224672,
	"nofThreads": "27",
	"nofClients": 0,
	"nofAppenders": 0,
	"verbose": false,
	"totalRx": "0",
	"loadAct": "0",
	"loadAvg": "0",
	"loadMax": "0",
	"dbTotalCommit": "0",
	"dbThroughputAvg": "0",
	"dbThroughputAct": "0",
	"overload": "0.00",
	"overloadAlarm": false,
	"overflowHit": 0,
	"overflowCache": 0,
	"overflowDisk": 24,
	"dbSize": "5.9 MB",
	"dbCount": 1097,
	"dbProduct": "Apache Derby",
	"dbDriverVersion": "10.7.1.1 - (1040133)",
	"dbDriverName": "Apache Derby Embedded JDBC Driver",
	"dbVersion": "10.7.1.1 - (1040133)",
	"dbMaintRelevant": true,
	"dbMaint": "not scheduled",
	"databaseOK": true
	}

4.5 Queries

To fetch historical data from logFaces database the caller initiates **new query** and then makes subsequent calls to draw the results page by page. Size of the page is specified as one of the query parameters submitted with this POST request:

Request	`POST /api/query`
Request header	`Authorization : 'askdfhlkajhsdlkfjhalskdfj.....'` `Content-Type : application/json`
Request body	see query format

Response	`200 - logged in ok`
	`403 - access is denied`
	`500 - server error`

Response body	see results format

To page through the results of started query, a series of paging request should follow until no more results indicated. Note that `${qid}` parameter must be taken from the previous call to identify the query being paged through.

Request	`GET /api/query/${qid}`
Request header	`Authorization : 'askdfhlkajhsdlkfjhalskdfj.....'`
Request body	see query format

Response	`200 - logged in ok`
	`403 - access is denied`
	`500 - server error`

Response body	see results format

4.5.1 Query format

The following parameter can be submitted with query request, it must be valid JSON structure as shown in the example below:

Name	Description	Default
limit	maximum number of events to fetch, -1 for no limitation	-1
pageSize	size of the page for drawing the result set	100
fromTime	cover events from this time (UTC epoch time in ms)	current server time - 1 hour
toTime	cover events until this time (UTC epoch time in ms)	current server time
order	sorting of results: ascending,descending,natural	ascending
criteria	see criteria format for more details	

```
{
    "limit": -1,
    "pageSize": 10,
    "criteria":[
        [
            {
                "attr": "thrown",
                "oper": "is",
                "expr": true
            }
        ],
        [
            {
                "attr": "domainName",
                "oper": "is",
                "expr": "app1"
            },
            {
                "attr": "hostName",
                "oper": "is",
                "expr": "hos1"
            }
        ]
    ]
}
```

4.5.2 Query results

Query results come in JSON format containing **query id**, indication that there are **more** results available to draw, and the array of results in the format described in data model.

The **qid** should be used to draw the next page of results if needed. When there are no more results available, the query can be considered as complete, calling with the same **qid** will never produce any more results.

Below is a query response example:


```
{
    "qid": "2-1501149036451",
    "more": true,
    "events": [
        {
            "t": 1501145460099,
            "q": 123,
            "p": 5000,
            "r": "DefaultQuartzScheduler_Worker-4",
            "h": "dima8",
            "a": "lfs",
            "m": "inflow rate avg 0 (evt/sec), max 2 (evt/sec), total rxed: 117",
            "w": false,
            "g": "com.moonlit.logfaces.server.receivers.Router",
            "f": "Router.java",
            "e": "updateBenchmark",
            "l": "242",
            "c": "com.moonlit.logfaces.server.receivers.Router"
        }
    ]
}
```

4.5.3 Criteria format

Criteria describes how to match the log data with a set of rules and conditions making complex boolean statement either true or false. Criteria expressions are used throughout logFaces for queries, real time monitoring, triggers, reports, and other tools. This is how it works:

Each Criteria is a collection (array) of Rules. Each Rule is a collection (array) of Conditions. Each Condition is a single boolean expression. When ALL Conditions in Rule are true, the Rule is true. When ANY of the rules is true, the Criteria is true. Conditions can match any of the attributes described in data model using any of the matcher operations listed below:

Operation	Description
is	equals, good for any type of attribute
isnot	not equals, good for any type of attribute
contains	for matching sub-strings in string attributes
notcontains	negative matching of sub-strings in string attributes
regex	matching attributes by regular expression
noregex	negative matching attributes by regular expression
more	higher than, used for numeric attributes
less	lower than, used for numeric attributes
emore	higher or equals than, used for numeric attributes
eless	lower or equals than, used for numeric attributes

This example illustrates Criteria with two rules, it will match **either** thrown events **OR** anything from application named 'app1' **or** host named 'host1'.

```
{
    "limit": -1,
    "pageSize": 10,
    "criteria":[
        [
            {
                "attr": "thrown",
                "oper": "is",
                "expr": true
            }
        ],
        [
            {
                "attr": "domainName",
                "oper": "is",
                "expr": "app1"
            },
            {
                "attr": "hostName",
                "oper": "is",
                "expr": "hos1"
            }
        ]
    ]
}
```

4.6 Monitoring

It is also possible to instantiate real-time perspectives on server and continuously draw the data matched by the their criteria. Similar to how data queries implemented, the monitoring calls come in two parts. First we want to create new perspective with parameters, and then we want to call for more data.

4.6.1 Start perspective

Request	`POST /api/perspectives`
Request headers	`Authorization : 'askdfhlkajhsdlkfjhalskdfj.....'` `Content-Type : application/json`

Request body	`{`
	`criteria : [...]`
	`}`
	
	format of criteria is identical to queries

Response	`200 - logged in ok`
	`403 - access is denied`
	`500 - server error`

Response body	`{`
	`"pid": "…."`
	`}`
	
	this is a perspective id for the next calls

4.6.2 Pull perspective data

Once perspective is started, the caller can draw the data by periodically doing the second call below. Note that `${pid}` parameter must be taken from the previous call to identify the query being paged through. The `${timeout}` parameter (in seconds) makes this call blocked by server when no data is available. When new data arrives, the call will return immediately with all the data. However, when no data is available, the serve will hold the call for at least number of seconds specified. This is done to reduce the unnecessary trips to and from the server by the caller.

Request	`GET /api/perspectives/${pid}/listen?timeout=${timeout}`
Request headers	`Authorization : 'askdfhlkajhsdlkfjhalskdfj.....'`
Response	200 - logged in ok 403 - access is denied 500 - server error
Response body	[{ a: 'application-name', h: 'host-name', g: 'logger name or exception class', …. },...] see data model for mode details

4.6.3 Remove perspectives

Real time perspectives are server resources and should be disposed of not used any longer. Do the following call to remove active perspective.

Request	`POST /api/perspectives/${pid}/remove`
Request headers	`Authorization : 'askdfhlkajhsdlkfjhalskdfj.....'`
Response	200 - logged in ok 403 - access is denied 500 - server error

4.7 Repository

Repository in logFaces is a meta-data describing entire collection of logs held in database. From repository, the caller can get a names of hosts, applications, loggers and exceptions known to the system.

Request	`GET /api/repo`
Request headers	`Authorization : 'askdfhlkajhsdlkfjhalskdfj.....'`

Response	`200 - logged in ok`
	`403 - access is denied`
	`500 - server error`

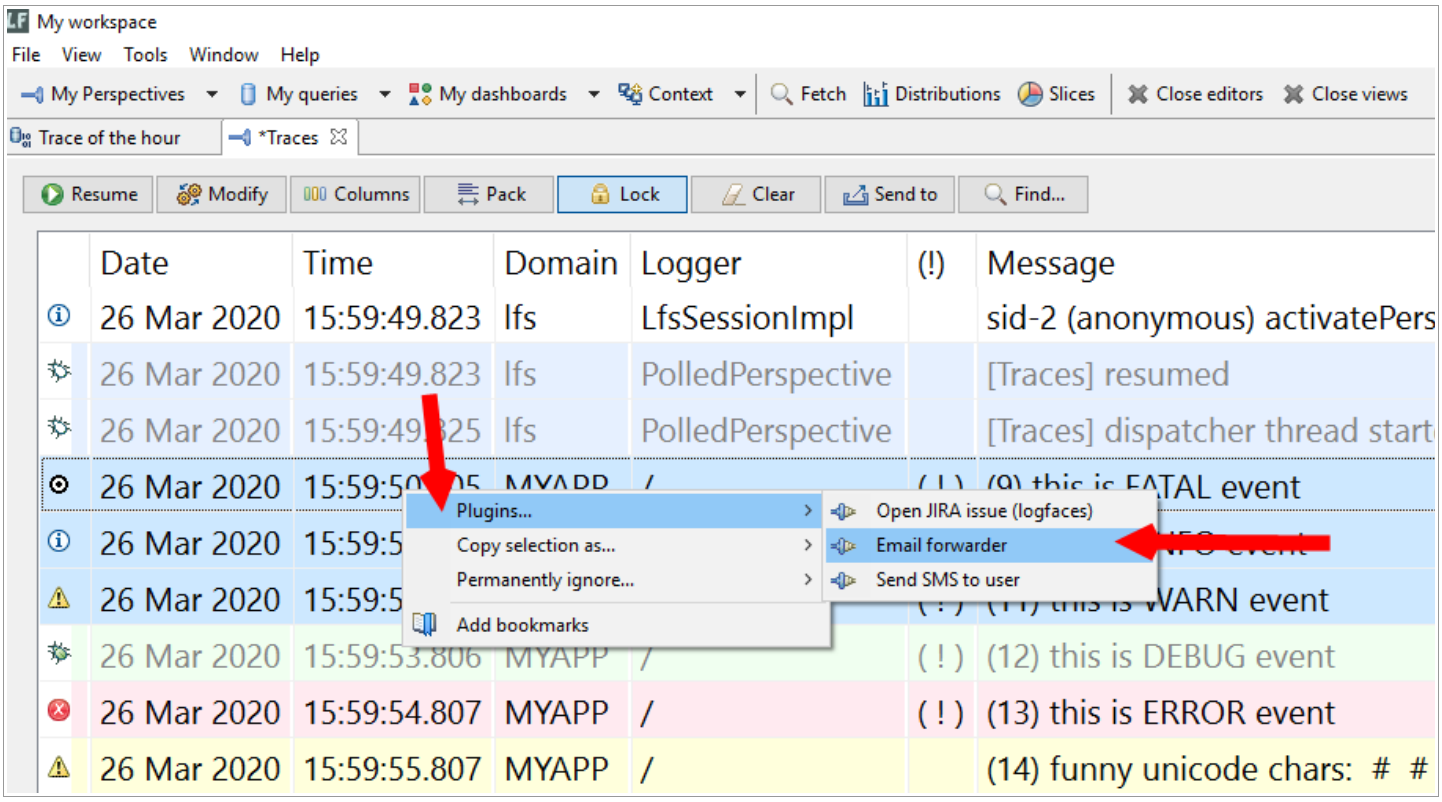
Response body	`[`
	`{`
	`a: 'application-name',`
	`h: 'host-name',`
	`g: 'logger name or exception class'`
	`},...`
	
	`]`

5 Plugins

logFaces has many features, however there is no rule that fits all, we all have different goals. Plugins extend existing functionality. This is done by running software written by you, using your own domain knowledge and trying to achieve your very own specific goals. This can be done directly from the logFaces user interface when you see the interesting log data.

Plugins are modules written in Groovy language following the guidelines outlined in this section. Once deployed on your server, plugins show up in your clients and can be used by all team members. For example, if you want to open a ticket in your customer support system when you see certain logs on your screen, you would call the function implemented in a particular plugin. The plugin may prompt user for arguments (if required) and then execute its code.

When client connects to a logFaces server it downloads all the plugins available in that server instance and makes them available for usage. The right-click context menus in all data views have a **Plugins** sub-menu with the list of all available plugins currently available. For example:



The execution of plugin function is performed inside client JVM using an isolated class loader with all the dependencies you supply with that specific plugin. Dependencies may include any jars or resources your plugin needs to serve the user call.

So, what is the work-flow for extending logFaces functionality with plugins?

1. **Create** a class implementing the `LogFacesPlugin` interface in your own IDE using whatever tools you normally use. We have created a starter project in GitHub in order to demonstrate how plugins usually look. Feel free to re-use and extend.

2. **Test** the plugin before deploying it. This is also up to you how testing is done. Just make sure that it does what you expect it to do and doesn't kill the hosting JVM (which are your clients).

3. **Deploy** to you server.

5.1 Implementation

Creation of logFaces plugins is based on the implementation of a single interface called `LogFacesPlugin`. The plugin development requires **lfs-core.jar** in your class path. It can be found in popular Maven repositories using the following coordinates:

```
<dependency>
  <groupId>com.logfaces</groupId>
  <artifactId>lfs-core</artifactId>
  <version>5.2.4</version>
</dependency>
```

Here is the `LogFacesPlugin` interface definition:

LogFacesPlugin.java

```
1  package com.moonlit.logfaces.server.core;
2
3  import java.util.List;
5  import java.util.Map;
6
7  public interface LogFacesPlugin {
8      public String getName();
9      public List<String> getArgs();
10     public Object handleEvents(List<LogEvent> events, Map<String,String> args);
11     public Object validate();
12 }
```

Implementation of this interface must be in Groovy language. Lets have a closer look and see how it should be implemented:

Plugins are named and their names show up in server administration page as well as inside client menus. The `getName()` method serves this purpose.

Sometimes plugins need to be invoked with arguments. If this is the case, the `getArgs()` method is used to prompt the user for the input. What expected from this call is a list of argument names, which is later used for calling the `handleEvents(..)` method which is the core of your plugin implementation. The user will not be prompted if no arguments are required.

The `List<LogEvent>` comes from the context where the user actually invokes the plugin and the arguments are the parameters the user wants the call to be executed with. Note that the `LogEvent` interface describes the actual log events from the logFaces data model and your plugin has a chance

to peek inside them (if needed) and do whatever is required with this information.

For example, if you want to send an email with selected log statements, your plugin may want to format the log lines and attach them to an email payload. The details of the `LogEvent` interface can be found in the JavaDoc, which can also be found in central Maven repository.

In some cases, it will make sense to validate the plugin before it gets deployed into foreign JVM. Use the `validate()` call for doing anything which will be required to verify the plugin well being. Plugins which throw any kind of exception are rejected during the deployment process. Users receive a text message with whatever this call returns.

In its minimal form a plugin can be comprised of a single Groovy class implementing the `LogFacesPlugin` but this is oversimplification. Any non-trivial coding requires some dependencies on other libraries and resources. There are no restriction on what or how your plugin uses as long as it compiles into a valid Java byte code. Note that plugins are loaded by the host JVM using a separate class loader. So, if you are referencing third-party classes (or resources) make sure that all these dependencies are available at the run time.

Some plugins may require a life cycle management for graceful initialization or termination invoked explicitly when instantiated or removed by the logFaces server. Such plugins should implement an interface named `Initializable` from the dependency jar. The server will then attempt to call `initialize()` and `terminate()` methods on such plugins whenever they get started or stopped.

5.2 Testing recommendations

Testing before deploying your plugin is essential. Basically, you want to test how the `handleEvents(..)` method behaves and whether it does what you expect it to do. One of the ways to test is to create JUnit test case and challenge your plugin from different directions. Note that you will have to offer some mockup implementation of the `LogEvent` interface which represents the actual log event from our data model.

5.3 Deployment and administration

Once your plugin is ready and tested, the final step is to share it with your users. Plugins are managed in logFaces server and delivered to all clients when they connect to this server. Go to `admin/context/plugins` page and upload your plugin files. Make sure you include all the dependencies your plugin requires. The server will attempt to load and validate the plugin and if

everything is OK, it will be listed and become ready for consumption. Plugins are stored on the server local directory named `/plugins` .