Java API User Manual

v4.4
1 Introduction

LogFaces server API is created to let you, the programmer, use aggregated log data in your own system. If you are familiar with the way log4j works, look at logFaces API as a "super-receiver", which simultaneously gets data from many applications and hosts. You will be able to monitor errors coming from various ends in your system and use this information for your own needs. Many systems today have their own management or administration consoles, using logFaces API you will be able to embed log data into your own consoles, build customized log viewers or react to events in your system.

You can create your own Java applications which, by means of the API, can subscribe to an interesting logging events going through the logFaces server and get notified when those events really come across. Once received, the events are in your hands, you can do whatever processing you need with them. Your application could be, for example, an instrumentation utility, textual console, rich user interfaces window, or even a web page. It could be stateful or stateless, it's entirely up to you.

API will let you create and process complex rules for notifications and alerts. Imagine that you want to get paged when rate of exceptions from a data layer reaches 10 per hour, or there is a fatal error in some host. Using our API will allow you to blend this information nicely into your system.

Figure 1: Architecture
2 Structure and dependencies

API is distributed in logfaces-api.zip file which you should unzip somewhere on your local disk. Distribution contains /lib directory with jars which you should place in the classpath of your application, /doc directory with JavaDoc documentation and /examples containing Eclipse project which you can use directly to get started.

3 How to use the API

API serves two major purposes:

1. Allow real-time monitoring of logging data coming through logFaces server

2. Allow instant data queries

In order to use either, we first need to obtain a connection to the server. This is done by means of LogFacesAPI static factory. You have to call one of it's createConnection( ) methods at least once in your application in order to obtain LogFacesConnection instance – live connection to logFaces server.

LogFacesConnection has a built it monitor to keep connection with logFaces server. Whenever server goes down, the connection will detect this and notify its listeners. Same applies to the case when disconnected server comes live again. If you are interested in those events, you can supply LogFacesConnectionListener implementation when creating the connection and get notified instantly when this happens.

Once you've got LogFacesConnection object, you can use it either for creating views or queries. Note that there is no need to keep the connection object in the scope, it can always be obtained from LogFacesAPI.getConnection( ) call.

Make sure to do LogFacesAPI.closeConnection( ) when you're done with you application, otherwise the connection will be lingering on server side for a long time and may result in resource leaks.
4 Real-time views

In order to obtain a real-time stream of logging events from logFaces server, we need to open `LogFacesConnection`, create `LogFacesView` and implement `LogFacesViewListener` for processing the events which will be coming through the view when it’s started. Consider this code snippet:

```java
public class ProblemsView implements LogFacesViewListener, LogFacesConnectionListener {
    public static void main(String[] args) throws Exception {
        new ProblemsView().start(args[0], Integer.parseInt(args[1]));
    }

    private void start(String host, int port) throws Exception {
        LogFacesConnection connection = LogFacesAPI.openConnection(host, port, false, this);
        LogFacesView view = connection.createView("problemsView", this);

        CriteriaFilter criteria = LogFacesAPI.makeCriteria();
        criteria.addRule()
            .addCondition(EventAttribute.loggerLevel, Operation.emore, Level.WARN.toInt())
            .addCondition(EventAttribute.thrown, Operation.is, Boolean.TRUE);
        view.setCriteriaFilter(criteria);
        view.activate();
    }

    public void handleEvent(LoggingEvent event) {
        System.out.println(event.getRenderedMessage());
    }

    public void connected() {
        System.out.println("connected to logfaces server");
    }

    public void disconnected() {
        System.out.println("lost connection with logfaces server");
    }
}
```

<table>
<thead>
<tr>
<th>Line</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>38</td>
<td>To receive data from the server we implement <code>LogFacesViewListener</code>. When events will be pushed to the view, we will process them in <code>handleEvent</code>(...) method.</td>
</tr>
<tr>
<td>38</td>
<td><code>LogFacesConnectionListener</code> is an optional interface you can implement to get notified when connection with server is lost or gained. Note that your application can be working all the time with or without logFaces server available – the API hides all the communication aspects.</td>
</tr>
<tr>
<td>44</td>
<td>Creating connection with logFaces server. Use <code>LogFacesAPI</code> static factory for managing connections. Along with host name and port number, you can also pass <code>LogFacesViewListener</code>.</td>
</tr>
<tr>
<td>45</td>
<td>Create the real-time view from connection</td>
</tr>
<tr>
<td>47-51</td>
<td>Create criteria filter and attach it to view to focus on events which are above WARN level AND are thrown exceptions.</td>
</tr>
<tr>
<td>52</td>
<td>Activate the view. This will start internal listeners and send received events to <code>handleEvent(...)</code> method implementation.</td>
</tr>
<tr>
<td>55-57</td>
<td>Implementation of <code>LogFacesViewListener.handleEvent</code>. The API will call this method when there is an event received from the server. Events are pushed to the view asynchronously using a background thread, but it's a good practice not to block this call and let it go as soon as you done processing the event.</td>
</tr>
<tr>
<td>59-65</td>
<td>Implementation of optional <code>LogFacesConnectionListener</code> interface which can be used to track connection state between application and logFaces server.</td>
</tr>
</tbody>
</table>

It's a good idea to do `logFacesAPI.closeConnection()` when you're done with you application.
5 Using Criteria

Use CriteriaFilter interface to build complex Boolean expressions used in both real-time views and queries. Criteria is a set of rules where each rule is a set of conditions. By default the criteria is OR typed - it will qualify when ANY of its rules qualify while ALL condition in each rule are true. You can change the criteria type to AND, - such criteria will qualify when ALL its rules qualify while ANY condition in each rule is true. There is a simple interfaces to build fairly complex expressions written with a few lines of code.

Consider the example below, this criteria has two rules each having two conditions. It will qualify any log event which is a thrown exception and its level is higher or equal to WARN, OR event is of FATAL level with MDC sessionID containing word “session1”.

Using EventAttribute and Operation enumerations you can improvise with all sorts of expressions.

```java
protected void start(String host, int port) throws Exception{
    LogFacesConnection connection = LogFacesAPI.openConnection(host, port, false, this);
    LogFacesView view = connection.createView("problemsView", this);
    CriteriaFilter criteria = LogFacesAPI.makeCriteria();
    criteria.addRule()
        .addCondition(EventAttribute.loggerLevel, Operation.omore, Level.WARN.toInt())
        .addCondition(EventAttributethrown, Operation.is, Boolean.TRUE);
    criteria.addRule()
        .addCondition(EventAttribute.loggerLevel, Operation.is, Level.FATAL.toInt())
        .addCondition("sessionID", Operation.contains, "session1");
    view.setCriteriaFilter(criteria);
    view.activate();
}
```

<table>
<thead>
<tr>
<th>Line</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>57-59</td>
<td>Create connection to the server and create logFaces view</td>
</tr>
<tr>
<td>61</td>
<td>Create empty criteria</td>
</tr>
<tr>
<td>62</td>
<td>Add first rule to criteria</td>
</tr>
<tr>
<td>63</td>
<td>Add condition to qualify level for WARN and upper levels</td>
</tr>
<tr>
<td>64</td>
<td>Add condition to qualify only thrown exception</td>
</tr>
<tr>
<td>65</td>
<td>Add second rule to criteria</td>
</tr>
<tr>
<td>66</td>
<td>Add condition to qualify level to be equal to FATAL</td>
</tr>
<tr>
<td>67</td>
<td>Add condition to qualify MDC property named “sessionID” to contain word “session1”</td>
</tr>
<tr>
<td>68</td>
<td>Let the view use this criteria, this can be done any time even after view is activated.</td>
</tr>
<tr>
<td>70</td>
<td>Activate the view and start receiving call backs from the server</td>
</tr>
</tbody>
</table>
6 Queries

It is possible to obtain historical data from logFaces database. Use LogFacesQuery instance for doing this. Consider the following code snippet:

```java
public class ProblemsQuery{
    public static void main(String[] args) throws Exception{
        new ProblemsQuery().go(args[0], Integer.parseInt(args[1]));
    }

    public void go(String host, int port) throws Exception{
        LogFacesConnection connection = LogFacesAPI.openConnection(host, port, false);
        LogFacesQuery query = connection.createQuery("myquery");
        query.setTimeRange(TimeRange.HOUR_1);
        query.setMaxSize(100);
        query.setSizePage(1);
        CriteriaFilter criteria = LogFacesAPI.makeCriteria();
        criteria.addRule()
            .addCondition(EventAttribute.logerLevel, Operation.emore, Level.ERROR.toInt())
            .addCondition(EventAttribute.thrown, Operation.is, Boolean.TRUE);
        query.setCriteria(criteria);
        query.getResultsAsync(new MyQueryListener());
    }

    class MyQueryListener implements LogFacesQueryListener{
        public void handleQueryResults(List<LoggingEvent> events) {
            for(LoggingEvent event : events){
                System.out.println(String.format("[%s] %s",
                    event.getProperty("application"),
                    event.getRenderedMessage()));
            }
        }
    }
```

<table>
<thead>
<tr>
<th>Line</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>38</td>
<td>Creating connection with logFaces server. Use LogFacesAPI static factory for managing connections.</td>
</tr>
<tr>
<td>39</td>
<td>Once opened, the connection can be used for creating LogFacesQuery objects. Note that each query is named and can be retrieved from connection any time.</td>
</tr>
<tr>
<td>40</td>
<td>Specify the time range to cover by the query</td>
</tr>
<tr>
<td>41</td>
<td>Limit the amount of results which will be returned, use -1 for unlimited</td>
</tr>
<tr>
<td>42</td>
<td>Specify the page size if using asynchronous query. Results will be delivered in batches of specified size.</td>
</tr>
<tr>
<td>44-48</td>
<td>Create criteria to match the log events. In this example we fetch thrown exceptions with ERROR+ seventies.</td>
</tr>
<tr>
<td>50</td>
<td>Launch the query using a listener with asynchronous processing</td>
</tr>
<tr>
<td>54-63</td>
<td>Query listener implementation, this is where we process the results returned by query. Note that results will arrive in the batches of the page size specified in the query definition.</td>
</tr>
</tbody>
</table>
7 Running examples in Eclipse

API distribution comes with example Eclipse project which you might want to import into your workspace and try it out. There are two classes in the project – ProblemsView and ProblemsQuery, exactly the ones described in previous sections.

In order to import the project into your workspace, click on File / Import menu and select “Existing Projects into Workspace”. Then navigate to the installation directory and select examples folder:

![Import Projects]

To run the examples, make sure to specify two arguments in launch configuration, one for the server host and another for the port number. Before running the examples, make sure that logFaces server is running.