JDK Flight Recorder
Events and API features

Liberica JDK
Revision 1.0
October 17, 2023

bellsoft

BellSoft software contains open source software. Additional information about third party code is available at https://bell-sw.com/third_party_licenses. You can also get more information on how to get a copy of source code by contacting info@bell-sw.com.

The use of any software product referenced in this document is governed by the applicable license agreement, which is not modified in any way by the terms of this notice.

Contents

1. Introduction

JDK Flight Recorder (JFR) is a diagnostic tool built into OpenJDK. Besides out of box integration with JDK tools like Mission Control and jcmd, Flight Recorder has an API.

JDK Flight Recorder is a part of OpenJDK, so is its API. If your JDK distribution is JFR enabled, you can access it via API too; no extra dependency is required.

There are three prominent use cases for using the Flight Recorder API.

- **Control Flight Recorder.** There are multiple ways to control JFR sessions: jcmd, JMX, JVM arguments. Handling sessions with API is an extra option offering you even more flexibility.
- **Read events from JFR binary files.** This can be handy if you want to build automated reporting or integrate JFR with your monitoring stack. Custom processing of recordings is also essential for the next use case, custom events.
- **Create application-specific Flight Recorder events.** You can leverage JFR infrastructure to produce events specific to your applications. Features such as periodic events and thresholds are also available for custom events.

2. Introducing a simple event

JDK Flight Recorder is designed around events. A JFR event is a simple data record, and its type defines fields available in the data record. Fields can be of simple types such as string, numbers, or boolean. There is also special handling for classes and threads, but other complex objects are not supported. Finally, a stack trace could be associated with each event.

JVM has more than a hundred built-in event types. Besides that, your application can define new custom event types and start producing related events, which will later be stored in JFR recording files and available through JFR capable tools such as Mission Control.

Let's start instrumenting a simple Spring Web application. First, we will generate an event for each HTTP call.

You will require Liberica JDK with JDK Flight Recorder support. Use Liberica JDK 11 or greater; in case you prefer JDK 8, pick the latest version. You may also want to have Mission Control ready. BellSoft provides a Mission Control executable as well.

To declare a new JFR event, we should first declare a subclass of the jdk.jfr.Event class. Let's consider a very simple event with a few custom fields.

```
package com.example.restservice;
import jdk.jfr.Event;
public class RestCallEvent extends Event {
  public String path;
  public String key;
  public long dataSize;
}
```

Now, we need application code to produce events.

```
package com.example.restservice;
import java.util.Map;
import java.util.concurrent.ConcurrentHashMap;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.RestController;
@RestController
public class DemoController {
  private final Map<String, String> store = new ConcurrentHashMap<>();
  @GetMapping("/get")
  public String get(
       @RequestParam(value = "key") String key) {
```

```
    RestCallEvent event = new RestCallEvent();
    event.begin();
    event.key = key;
    try {
      String value = store.getOrDefault(key, "Not Found");
      event.dataSize = value.length();
    return value;
    } finally {
      event.end();
      event.commit();
    }
  }
  @GetMapping("/set")
  public String set(
       @RequestParam(value = "key") String key,
       @RequestParam(value = "val", defaultValue="") String val)
    RestCallEvent event = new RestCallEvent();
    event.begin();
    event.key = key;
    try {
      store.put(key, val);
      event.dataSize = val.length();
      return val;
    } finally {
      event.end();
      event.commit();
    }
  }
}
```

The `jdk.jfr.Event` base class automatically adds timestamp and event duration fields, though we need to demarcate the event's duration using `being()` and `end()` methods. The commit() method tells JFR that event data is complete and can be put into the event buffer.

In our example, each event has a new instance of an event object created for it, which is not necessary. We can reuse event instances for multiple events. It makes a garbage-free JFR event producer totally possible, although you would need to implement an instance per thread or some other thread-safe object reuse pattern.

It's time to test our new event.

While the application is up and running, you can start the Flight Recorder session with jcmd using the following command (5016 is a JVM process PID in our case):

```
jcmd 5016 JFR.start
```

```
5016:
Started recording 1. No limit specified, using maxsize=250MB as default.
Use jcmd 5016 JFR.dump name=1 filename=FILEPATH to copy recording data to file.
```

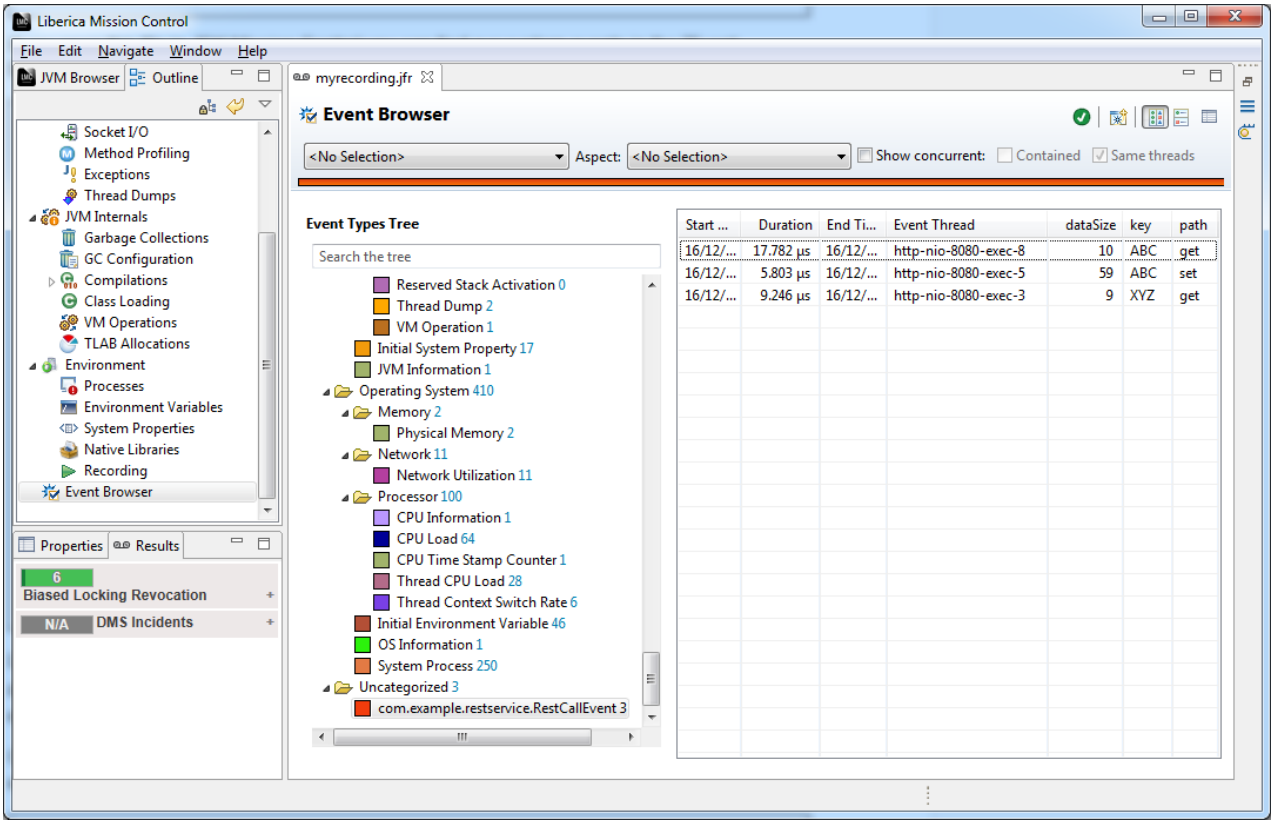
Notice the command hint provided by jcmd to dump out the flight recording later.

When recording is active, hit your HTTP endpoint a few times.

Now we can dump the recorded JFR events and stop the recording session.

```
jcmd 5016 JFR.dump name=1 filename=myrecording.jfr
5016:
Dumped recording "1", 726.3 kB written to:
/home/aragozin/myrecording.jfr
jcmd 5016 JFR.stop name=1
```

Open this file in JDK Mission Control: you'll find our custom events in the **Event Browser** report.



Let's try another tool from JDK's arsenal. jfr is a little command-line tool that can parse JFR files and extract event data.

The command below will print our events filtered by event type (a fully qualified name of an event class).

```
jfr.exe print --events "com.example.restservice.RestCallEvent" myrecording.jfr
com.example.restservice.RestCallEvent {
 startTime = 05:38:28.704
 duration = 17.781 us
 path = "get"
 key = "ABC"
 dataSize = 10
 eventThread = "http-nio-8080-exec-8" (javaThreadId = 24)
 stackTrace = [
  com.example.restservice.DemoController.get(String) line: 29
  sun.reflect.NativeMethodAccessorImpl.invoke0(Method, Object, Object[])
  sun.reflect.NativeMethodAccessorImpl.invoke(Object, Object[]) line: 62
  sun.reflect.DelegatingMethodAccessorImpl.invoke(Object, Object[]) line: 43
  java.lang.reflect.Method.invoke(Object, Object[]) line: 498
  ...
 ]
}

com.example.restservice.RestCallEvent {
 startTime = 05:38:45.819
 duration = 5.803 us
 path = "set"
 key = "ABC"
 dataSize = 59
 eventThread = "http-nio-8080-exec-5" (javaThreadId = 21)
 stackTrace = [
  com.example.restservice.DemoController.set(String, String) line: 47
  sun.reflect.NativeMethodAccessorImpl.invoke0(Method, Object, Object[])
  sun.reflect.NativeMethodAccessorImpl.invoke(Object, Object[]) line: 62
  sun.reflect.DelegatingMethodAccessorImpl.invoke(Object, Object[]) line: 43
  java.lang.reflect.Method.invoke(Object, Object[]) line: 498
  ...
 ]
}

com.example.restservice.RestCallEvent {
 startTime = 05:39:00.382
 duration = 9.246 us
 path = "get"
 key = "XYZ"
 dataSize = 9
 eventThread = "http-nio-8080-exec-3" (javaThreadId = 19)
 stackTrace = [
  com.example.restservice.DemoController.get(String) line: 29
```

```
  sun.reflect.NativeMethodAccessorImpl.invoke0(Method, Object, Object[])
  sun.reflect.NativeMethodAccessorImpl.invoke(Object, Object[]) line: 62
  sun.reflect.DelegatingMethodAccessorImpl.invoke(Object, Object[]) line: 43
  java.lang.reflect.Method.invoke(Object, Object[]) line: 498
  ...
 ]
}
```

Here, you can see the details of the events we've just produced.

Fields defined in event class have become event attributes. There are also several attributes added to every custom event type.

- startTime — timestamp of the event.
- duration — time between `start()` and `stop()` method calls.
- eventThread — reference to a Java thread that has produced an event.
- stackTrace — stack trace at `event.commit()` call.

3. JFR events versus logging

We can produce JFR events. But how is it different from plain logging?

Custom JFR events are not a logging replacement, as you can probably benefit from both in the same application.

Below are the most significant features of JFR events.

Low overhead. JFR events are optimized to reduce the impact on application code performance. Logging in a heavy, loaded application can easily become a performance bottleneck and contention point. JFR infrastructure is well optimized for concurrent and latency-sensitive applications.

Thresholds. You can set a duration threshold and keep only ``slow'' events.

Periodic events. After scheduling events, JFR will take care of calling your event producer.

Concurrent session management. The same JVM may have multiple Flight Recorder sessions running in parallel (e.g., one that continues on start-up and one initiated via jcmd). In this case, JVM will route your events to both recordings according to individual settings for each.

Compact stack trace storage. JFR files store stack traces in a compacted binary form, which is much more efficient than a plain text.

On the other hand, logs are flexible, and you are free to put everything into your log files. With JFR, you should keep your event structured and compact to get the most value out of JFR infrastructure. You can store arbitrary strings in JFR events, but large events are hard to manage. Both your runtime overhead and resulting recording file sizes will grow proportionally to the amount of events' data.

4. More JFR events

Let's improve our current event and add some more.

First, add some metadata to our event and attributes.

- The `@Label` annotation allows putting human-readable names on events and attributes. They will be visible in Mission Control.
- With `@Description`, you can add more details.
- `@Category` is essential for filtering and navigating in Mission Control's **Event Browser** report.
- `@DataAmmout` and a few other annotations mark measurement units for numeric attributes.

```
package com.example.restservice;
import jdk.jfr.Category;
import jdk.jfr.DataAmount;
import jdk.jfr.Description;
import jdk.jfr.Event;
import jdk.jfr.Label;
@Category("MyEvents")
@Label("REST Event")
@Description("REST Request Processing Event")
public class RestCallEvent extends Event {
  @Label("Request Path")
  public String path;
  @Label("Request Key")
  public String key;
  @Label("Result Size")
  @DataAmount(DataAmount.BYTES)
  public long dataSize;
}
```

Now, we'll introduce a periodic event. It will be a cache statistics event.

```
package com.example.restservice;
import java.lang.ref.WeakReference;
import java.util.Map;
import jdk.jfr.Category;
import jdk.jfr.Description;
import jdk.jfr.Event;
import jdk.jfr.FlightRecorder;
```

```
import jdk.jfr.Label;
import jdk.jfr.Period;
import jdk.jfr.StackTrace;
@Category("MyEvents")
@Label("Cache Stats")
@Description("Simple cache statistics")
@Period("10s")
@StackTrace(false)
public class CacheStatsEvent extends Event {
  @Label("Cache Name")
  public String cacheName;
  @Label("Cache Size")
  public int cacheSize;
public static void enableStatsRecording(String cacheName, Map<?, ?> cache) {
  final WeakReference<Map<?, ?>> ref = new WeakReference<Map<?,?>>(cache);
  final CacheStatsEvent event = new CacheStatsEvent();
  event.cacheName = cacheName;
  FlightRecorder.addPeriodicEvent(CacheStatsEvent.class, new Runnable() {
    @Override
    public void run() {
      Map<?, ?> cache = ref.get();
      if (cache == null) {
        FlightRecorder.removePeriodicEvent(this);
      } else {
        event.begin();
        event.cacheSize = cache.size();
        event.commit();
      }
    }
  });
}
}
```

Periodic events require explicit registration of callbacks with JDK Flight Recorder. While it's allowed to have multiple callbacks for the same event type, you also have to manage their deregistration.

In the example above, we are using a weak reference to avoid accidental memory leaks and automatically unregister callback once the GC clears the cache object.

There is also a particular type of period called `everyChunk` that will instruct JFR to trigger event callback at least once per recording file. It's useful for dumping immutable configuration data, so there is little point in writing the same data to the same JFR file more than once. `everyChunk` is the default value for the `@Period` annotation.

Let's introduce a cache configuration event, too.

```
package com.example.restservice;
import java.lang.ref.WeakReference;
import jdk.jfr.Category;
import jdk.jfr.Description;
import jdk.jfr.Event;
import jdk.jfr.FlightRecorder;
import jdk.jfr.Label;
import jdk.jfr.Period;
import jdk.jfr.StackTrace;
@Category("MyEvents")
@Label("Demo Config")
@Description("Demo Controller Configuration")
@Period()
@StackTrace(false)
public class DemoConfigEvent extends Event {
  @Label("Set is enabled")
  public boolean isSetEnabled;
  @Label("Get is enabled")
  public boolean isGetEnabled;
  public static void enableConfigRecording(DemoController controller) {
    final WeakReference<DemoController> ref =
       new WeakReference<DemoController>(controller);
    final DemoConfigEvent event = new DemoConfigEvent();
    FlightRecorder.addPeriodicEvent(DemoConfigEvent.class, new Runnable() {
      @Override
      public void run() {
        DemoController controller = ref.get();
        if (controller == null) {
          FlightRecorder.removePeriodicEvent(this);
        } else {
          event.begin();
          event.isGetEnabled = controller.isGetEnabled();
          event.isSetEnabled = controller.isSetEnabled();
          event.commit();
        }
      }
    });
  }
}
```

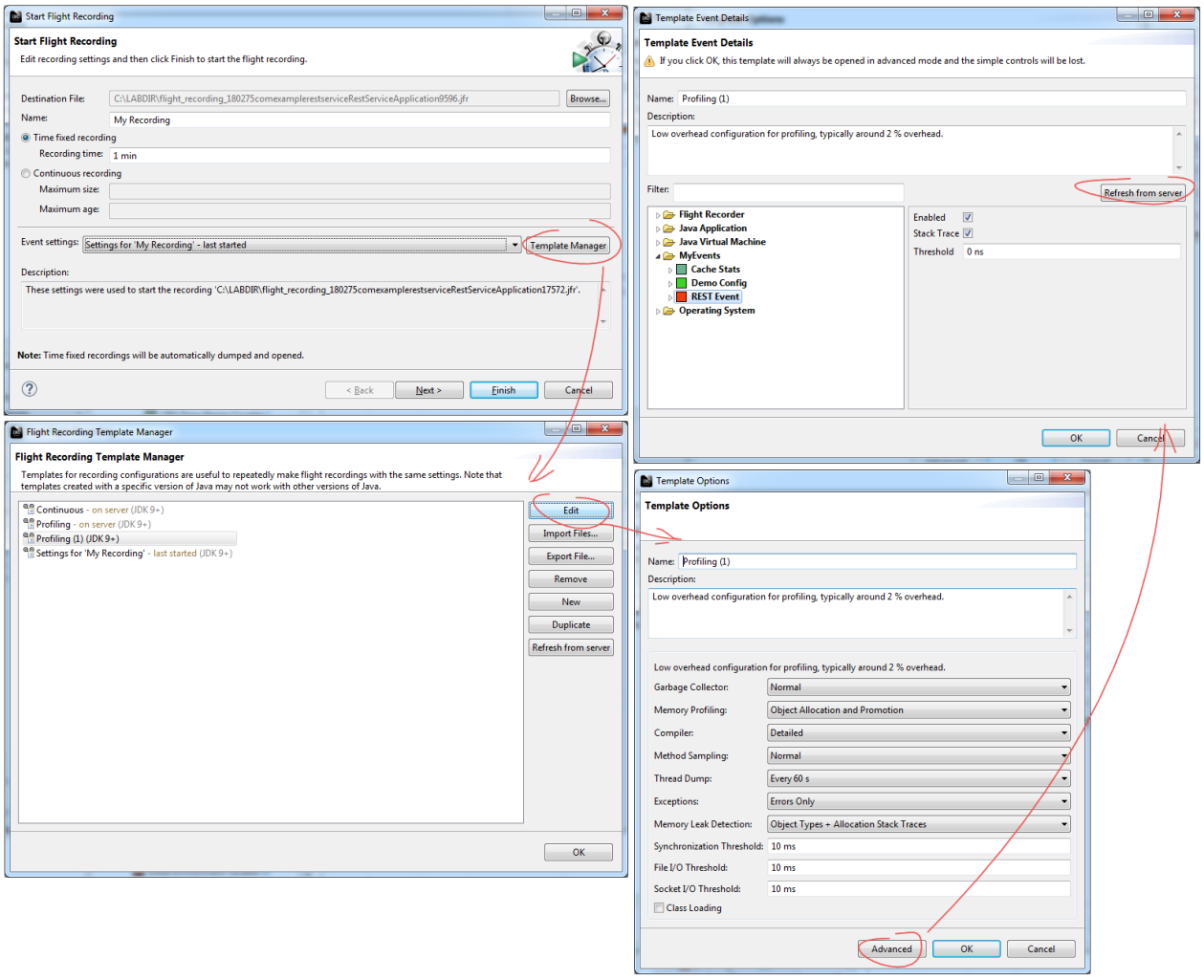
The `@StackTrace` annotation was used to disable stack traces for our periodic events.

5. Configuring custom events

Now, we have several custom events. Default recording settings for each custom event are defined with annotations on event class declaration. But soon you'd possibly want to fine tune these settings when starting a JDK Flight Recorder session.

The JFR session is customized via an XML configuration file. The recommended way to produce this configuration is through JDK Mission Control.

Provided our application is running, you can use JDK Mission Control to connect to it and open the **Start Flight Recording** window. Once the window is open, press **Template manager** and you will get a list of configurations. Copy some existing configuration and open the **Edit** dialog box on that copy. Now switch to advanced mode by clicking **Advanced** and then click **Refresh from server**.



The advanced mode in the template editor allows modifying settings for every event known to JDK Flight Recorder. A list of events and their metadata are available at running. **Refresh from server** pulls these data from the connected JVM, and you can see our custom events in the tree. Now, you have a choice: modify this configuration and use it with JDK Mission Control, or export it to a file to control JDK Flight Recorder from a console with the jcmd tool.

6. Controlling Flight Recorder via API

There are multiple ways to start flight recording, and one of them is an API.

You can programmatically prepare a configuration of the Flight Recorder session and start it or schedule the start later. There's a possibility to dump recorded events at any time and/or stop your recording session.

Why would you want to do it via API if jcmd and Mission Control can do it?

For multiple reasons. One is to remove extra options from the JVM start command and manage an automatic Flight Recording session start with application configuration. Another could be to control JDK Flight Recorder via HTTP instead of JMX.

You can also perform the automatic crash dumps. The idea of an automatic crash dump is to produce exhaustive diagnostic information, but only if the application hits a critical fault condition.

An automatic heap dump on particular exceptions illustrates this pattern nicely.

How does JDK Flight Recorder fit the pattern?

We can configure a background Flight Recorder session with verbose event level, but not persisted on disk (to avoid IO overhead). Events will be buffered in a fixed size memory buffer by Flight Recorder, then evicted to make room for new events. If a critical condition was encountered, we will dump recently collected events to the crash dump file.

Below is a simple example of a helper class for this pattern.

```
package com.example.restservice;
import java.io.IOException;
import java.nio.file.Path;
import java.text.ParseException;
import jdk.jfr.Configuration;
import jdk.jfr.EventType;
import jdk.jfr.FlightRecorder;
import jdk.jfr.Recording;
public class CrashDumpHelper {
  private static Recording rec;
  public static synchronized void activate()
          throws IOException, ParseException {
```

```
    if (rec != null) {
      Configuration conf = Configuration.getConfiguration("default");
      rec = new Recording(conf);
      configureEvents(rec);
      // disable disk writes
      rec.setToDisk(false);
      rec.start();
    }
  }

  private static void configureEvents(Recording rec) {
    for (EventType et: FlightRecorder.getFlightRecorder().getEventTypes()) {
      if (isEnabledForCrachDump(et)) {
        rec.enable(et.getName());
      }
    }
  }

  private static boolean isEnabledForCrachDump(EventType et) {
    // enabled application specific custom events
    ...
  }

  public void dump(Path filename) throws IOException {
    rec.dump(filename);
  }
}
```

A caveat here is that such a Flight Recorder dump is not very useful without intensive instrumentation of your code with custom events. You need application-specific instrumentation to be able to reconstruct a sequence of events leading to failure. In such a case, the low overhead of JFR events is a great advantage because your intensive instrumentation needs to have a low impact on the application's runtime performance.

7. Reading JFR events from file

The jfr tool from the OpenJDK toolset has the option to convert a JFR file to JSON. Combined with JSON processing tools such as jq, this could be enough to build sophisticated automated reporting out of JFR files.

However, if Java itself is your preferred tool for data processing, reading data out of JFR files directly is even easier.

The JFR file parser is a part of JVM runtime alongside other parts of JDK Flight Recorder. You can use the `jdk.jfr.consumer.RecordingFile` class to open a JFR file from disk. `RecordingFile` allows iterating through individual events in the file.

Below is a code example parsing the JFR dump file and printing out one of our custom events introduced earlier.

```
package com.example.restservice;
import java.io.IOException;
import java.nio.file.Path;
import java.nio.file.Paths;
import java.time.Instant;
import java.time.temporal.ChronoUnit;
import java.time.temporal.TemporalAmount;
import java.time.temporal.TemporalUnit;
import jdk.jfr.consumer.RecordedEvent;
import jdk.jfr.consumer.RecordingFile;
public class EventPrinter {
  private static final String REST_CALL_EVENT = RestCallEvent.class.getName();
  public static void main(String[] args) throws IOException {
    String recFile = args[0];
    RecordingFile rec = new RecordingFile(Paths.get(recFile));
    while (rec.hasMoreEvents()) {
      RecordedEvent event = rec.readEvent();
      if (REST_CALL_EVENT.equals(event.getEventType().getName())) {
        Instant startTime = event.getStartTime();
        String path = event.getString("path");
        String key = event.getString("key");
        long size = event.getLong("dataSize");
        long duration = event.getDuration().get(ChronoUnit.NANOS);
        System.out.println(String.format(
          "%s start=%s path=%s key=%s dataSize=%d duration=%dns",
          REST_CALL_EVENT, startTime, path, key, size, duration));
```

```
      }
    }
    rec.close();
  }
}
```

8. JFR event streaming in OpenJDK 14 and later

Starting with JDK 14 it is also possible to access events produced by JDK Flight Recorder in real-time. New API allows users to react to events immediately as they are produced, which opens new ways for integration with monitoring infrastructure. The same API also allows "tailing" on disk JFR files and getting your code to consume events as they are being written to disk, which is a good option to decouple your monitoring agent from the application JVM.

JDK Flight Recorder

Events and API features

bell//soft