



DATADOG

# Production profiling with JDK Flight Recorder & JDK Mission Control

Jean-Philippe Bempel, Datadog  
@jpbempel



# Production Time Profiling and Diagnostics

“The big challenge is no longer really performance. The big challenge is profiling, and especially profiling in production.”

- Tony Printezis, JVM engineer, Twitter  
(Devoxx 2015, “Life of a Twitter JVM Engineer”, 49:49)



## Production Profiling and Diagnostics?

- Minimal Observer Effect
  - Low overhead
  - Not affect the behaviour of the application
  - Not undo optimizations
- Safe to use in production
  - Well tested
  - Widely used





## Data Flight Recorder for the JVM

- Built into the JVM
- Records information about the JVM and the application
- Low overhead / High performance
- APIs available to record custom information
- Can be used to solve a range of different problems
- Open sourced by Oracle in OpenJDK 11
- Backported to OpenJDK 8 since 8u262/8u272!





## Flight Recorder Helps You...

### Resolve problems faster

- Data can always be recorded - no need for a reproducer
- Recordings can be captured and shared
- Find real bottlenecks in your applications

### Designed for production systems - low observer effect

- Profiling in JFR will not undo optimizations like scalarization
- Profiling in JFR will not be skewed by safe points (like Async Profiler)



# Comprehensive Tool Chain

- Control the Flight Recorder
  - Command line parameters
  - POJO API
  - JMX API
  - `jcmd` (tool in the JDK)
  - JDK Mission Control



# Comprehensive Tool Chain

- Add custom data
  - Java API (in the JDK)
  - JDK Mission Control (agent)
  - Third party integrations (Open Tracing, Brave etc)
  
- Use the data
  - JDK Mission Control (application)
  - JDK Mission Control (core libraries)
  - `jfr` (tool in the JDK)



## Flight Recorder Data

- Data recorded as events
- Events are data points in time
  - Event types have fields (a.k.a. attributes)
  - Fields are self describing (typed / content types e.g. long / epoch ms)



## Different Kinds of Data

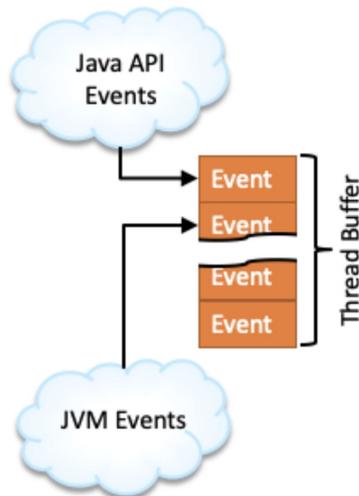
- CPU Profiling
- Allocation Profiling
- Thread Latency Profiling
- GC
- Compiler
- Memory Leak Profiler
- File & Socket IOs
- ...and much more

Size is typically about 2 megs per minute (app. 100k events)



## Flight Recorder Innards

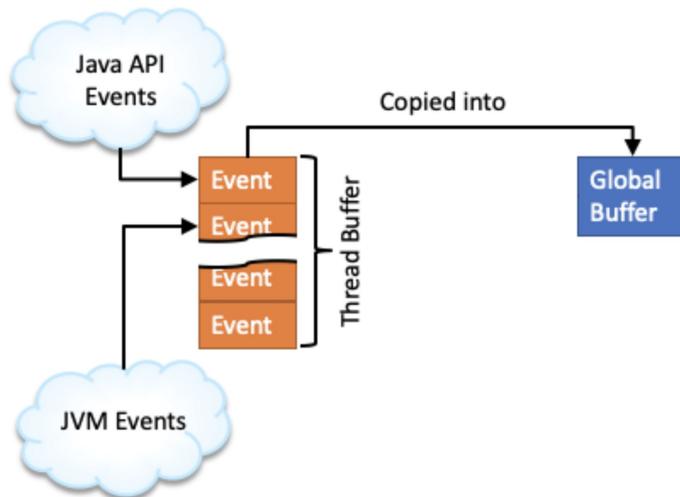
- High performance recording engine
- Events recorded into thread buffers





## Flight Recorder Innards

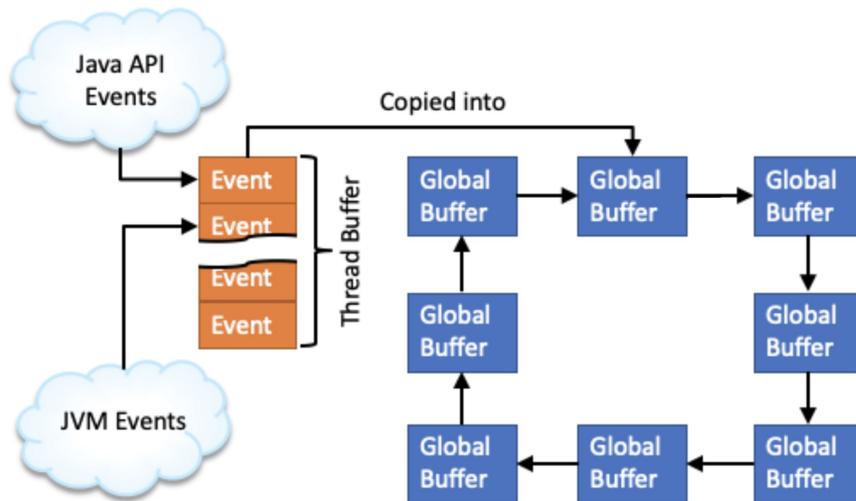
- High performance recording engine
- Events recorded into thread buffers
- When full, copied into global buffer





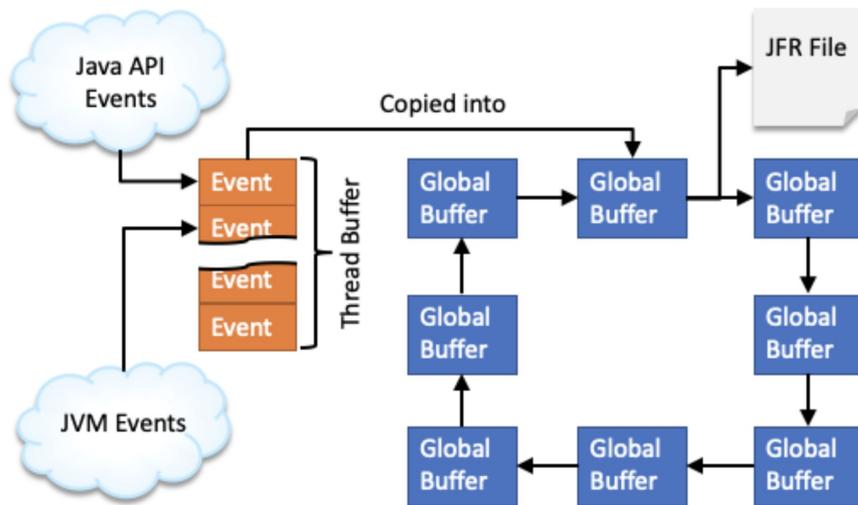
# Flight Recorder Innards

- High performance recording engine
- Events recorded into thread buffers
- When full, copied into global buffer
- Can be configured to keep on overwriting/reusing the buffer



## Flight Recorder Innards

- High performance recording engine
- Events recorded into thread buffers
- When full, copied into global buffer
- Can be configured to keep on overwriting/reusing the buffer
- ...or emit to disk





## Adding Custom Events

- Easy to correlate with events from the runtime
  - E.g. record a distributed tracing span, with trace id, span id and parent span id, then see what happened during the processing of that particular span
- Piggy back on the whole JFR infrastructure (jcmd, jmc, command line flags)
- High performance
  - High precision, cheap timestamping
  - Cheap stack traces
  - Binary, compact data
- Self describing, easy to consume

Examples: [github.com/thegreystone/java-svc/tree/master/jfr](https://github.com/thegreystone/java-svc/tree/master/jfr)



## Simple JFR Event Generation Example

```
public class HelloJfr {  
    @Label("Hello World")  
    static class HelloWorldEvent extends Event {  
        @Label("Message")  
        String message;  
    }  
    public static void main(String [] args) {  
        HelloWorldEvent event = new HelloWorldEvent();  
        event.message = "Hello World!";  
        event.commit();  
    }  
}
```



## Metadata Example

```
@Label("Native Library Load")
@Name("org.example.process.NativeLibraryLoad")
@Description("Emitted on the loading of a native library")
@Category("Process")
public final class LibraryLoadEvent extends Event {
    @Label("File Name")
    String fileName;

    @Label("Start Address")
    @MemoryAddress
    long startAddress;

    @Label("Bytes Loaded")
    @DataAmount(DataAmount.BYTES)
    long bytesLoaded;

    @Label("Library file creation time")
    @Timestamp(Timestamp.MILLISECONDS_SINCE_EPOCH)
    long creationTime;
}
```



# Controlling the Flight Recorder

- JDK Mission Control
  - Via JMX
- JCMD
  - Command line tool to talk to running JVMs
- Command line flags
  - `-XX:StartFlightRecording=delay=20s,duration=60s,name=MyRecording,filename=/tmp/myrecording.jfr,settings=profile`
- Programmatically
  - JMX API
  - Pojo API



## Flight Recorder Templates (.jfc files)

Contains information about what and how to record, e.g.:

- What event types to enable
- What thresholds to use for events with durations
- What periodicity to sample requestable events

There are two templates by default:

- default.jfc – less than 1% overhead
- profiling.jfc – less than 2% overhead

Templates are located in the lib/jfr folder of the JDK.

Templates can be edited and exported from JMC.



## Looking at Flight Recordings

- **JDK `jfr` tool**
  - Simple command-line tool to look at recordings
- **JDK parser**
  - Supports recordings with the JDK version the recording was created
  - Included in the JDK
  - External iteration



## Looking at Flight Recordings

- JMC core parser
  - Supports recordings from all versions of JFR
  - Compiles and runs on JDK 8+
  - Analysis Rules
  - Declarative / Internal iteration
  - Available as Maven artifacts



# Looking at Flight Recordings

## ● JMC

The screenshot displays the Java Mission Control (JMC) interface. The left sidebar shows the navigation tree with 'Method Profiling' selected. The main window is titled 'Method Profiling' and shows a table of top methods. Below this, there are sections for 'Predecessors', 'Successors', and 'Stack Trace'.

Top Method	Count	Percentage
Object org.openjdk.jmc.common.collection.FastAccessNumberMap.get(long)	1,066	10.2 %
Object it.unimi.dsi.fastutil.longs.Long2ObjectOpenHashMap.get(long)	410	3.94 %
voidjdk.internal.reflect.UnsafeFieldAccessorImpl.ensureObj(Object)	358	3.44 %
Boolean com.datadog.profiling.jfr.DDFrame\$DDMethod.isHidden()	291	2.8 %
ProfileProto\$Location com.datadog.profiling.pprof.MappedProfile.getLocation(long)	202	1.94 %
HashMap\$Node java.util.HashMap.getNode(Object)	200	1.92 %

Stack Trace	Count	Percentage
Object org.openjdk.jmc.common.collection.FastAccessNumberMap.get(long)	1066	100 %
Object org.openjdk.jmc.flightrecorder.internal.parser.v1.ValueReaders\$PoolReader.read(IDataInput, boolean)	474	44.5 %
Object org.openjdk.jmc.flightrecorder.internal.parser.v1.ValueReaders\$ReflectiveReader.read(IDataInput, boolean)	239	22.4 %
Object org.openjdk.jmc.flightrecorder.internal.parser.v1.ValueReaders\$ArrayReader.read(IDataInput, boolean)	223	20.9 %
Object org.openjdk.jmc.flightrecorder.internal.parser.v1.ValueReaders\$ReflectiveReader.read(IDataInput, boolean)	223	20.9 %
void org.openjdk.jmc.flightrecorder.internal.parser.v1.TypeManager\$TypeEntry.readConstant(IDataInput, boolean)	223	20.9 %
void org.openjdk.jmc.flightrecorder.internal.parser.v1.TypeManager.readConstants(long, IDataInput, boolean)	223	20.9 %
long org.openjdk.jmc.flightrecorder.internal.parser.v1.ChunkLoaderV1.readConstantPoolEvent(IDataInput, boolean)	223	20.9 %

Stack Trace	Count	Percentage
Object org.openjdk.jmc.common.collection.FastAccessNumberMap.get(long)	1066	100 %
Object org.openjdk.jmc.flightrecorder.internal.parser.v1.ValueReaders\$PoolReader.read(IDataInput, boolean)	474	44.5 %
void org.openjdk.jmc.flightrecorder.internal.parser.v1.TypeManager\$TypeEntry.readConstant(IDataInput, boolean)	279	26.2 %
void org.openjdk.jmc.flightrecorder.internal.parser.v1.TypeManager.readEvent(long, IDataInput, boolean)	201	18.9 %
Object org.openjdk.jmc.flightrecorder.internal.parser.v1.ValueReaders\$PoolReader.resolveObject(IDataInput, boolean)	103	9.66 %
Object org.openjdk.jmc.flightrecorder.internal.parser.v1.ValueReaders\$StringReader.read(IDataInput, boolean)	6	0.563 %
TypeManager\$TypeEntry org.openjdk.jmc.flightrecorder.internal.parser.v1.TypeManager.get...	3	0.281 %



# Looking at Flight Recordings

- Datadog Profiling!

The screenshot displays the Datadog Profiling interface for a Java application. At the top, it shows the application name 'prof-analyzer', environment 'env: prod', version 'v3943875-b15eb158', and a timestamp 'on Feb 24 09:28:52 (4d 17h uptime)'. A 'Download Profile Data' button is visible in the top right.

The main navigation bar includes 'Performance', 'Analysis (2)', 'Metrics', and 'Runtime Info'. A warning message states: 'Analysis: Detected (Fake CVE) QueuedThreadPool\$Runner CVE vulnerability. This affects security.' A 'View Full Analysis' button is located to the right of this message.

The central part of the interface features a flame graph. The top of the graph indicates 'CPU Time: 4m 12s over 60s, across all threads'. Below this, a table lists the methods and their corresponding CPU time. The methods are color-coded by category: JVM (pink), Garbage Collection (orange), ParallelGCFailedAllocation (green), and others. The graph shows a hierarchy of calls, with 'Thread.run()' at the top, followed by 'ThreadPoolExecutor\$Worker.run()', and then various worker methods like 'FutureTask.run()' and 'ChunkLoaderV1.call()'. The bottom of the graph shows a detailed view of the 'JfrParser.parseJfr(L...' method, which is highlighted in red.

On the right side, there is a list of methods with their CPU time and a progress bar. The methods are sorted by CPU time, with 'ParallelGCFailedAllocation' at the top (25.96s) and 'ArrayList.elementData(int)' at the bottom (2.83s). The list includes methods like 'FastAccessNumberMap...', 'Long2ObjectOpenHashM...', 'DDFrame\$DDMethod.isH...', 'ValueReaders\$Reflective...', 'HashMap.getNode(Object)', 'ImpreciseScaleFactor.tar...', 'UnsafeFieldAccessorImpl...', 'ParallelGCSystemGC', 'HashMap.resize()', 'UnsafeObjectFieldAccess...', 'Field.checkAccess(Class, O...', 'UnsafeObjectFieldAccess...', and 'ArrayList.elementData(int)'. A 'Method' dropdown menu is visible at the top of this list.



## The profiler will always lie to you...

- Every profiler in existence will lie to you
  - Change runtime behaviour (e.g. undo optimizations)
  - Biases (safe point bias)
- Talk: [Profilers Are Lying Hobbits \(and we hate them!\)](#) from Nitsan Wakart



## The profiler will always lie to you...

- Know your profiler and it's chosen trade-offs
  - JFR Execution Samples (CPU profiling) chooses low runtime impact and constant overhead. Will not include native samples.
  - JFR allocation profiling is sampling for nursery allocations - not exact counts but a good estimation, especially over time.
  - JFR provides contextual information, e.g. for locks (monitor enter):
    - Thread monitor class
    - Which thread was holding the monitor
    - Monitor address
    - ...

There is always a price, in this case thresholds...



DATADOG

# JFR Demos



DATADOG

# JDK Mission Control

# JDK Mission Control - JMC

## Tools suite

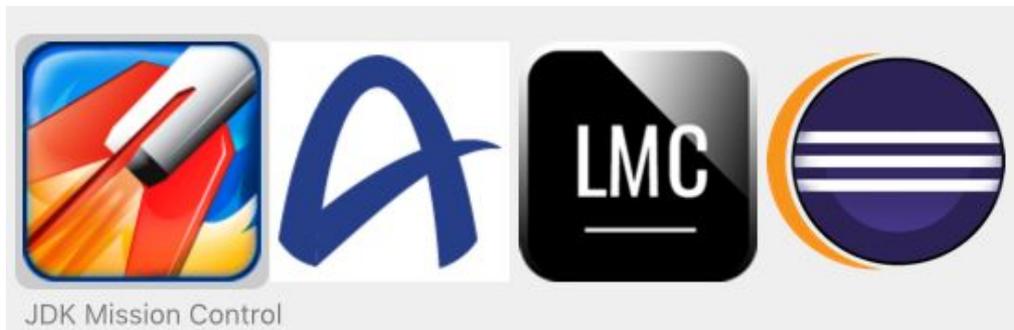
- JFR
  - Create
  - Analyze
- JMX Console
  - Real time monitoring
- Additional plug-ins
  - JOverflow
  - Moar JFR
  - Plug-in plug-in



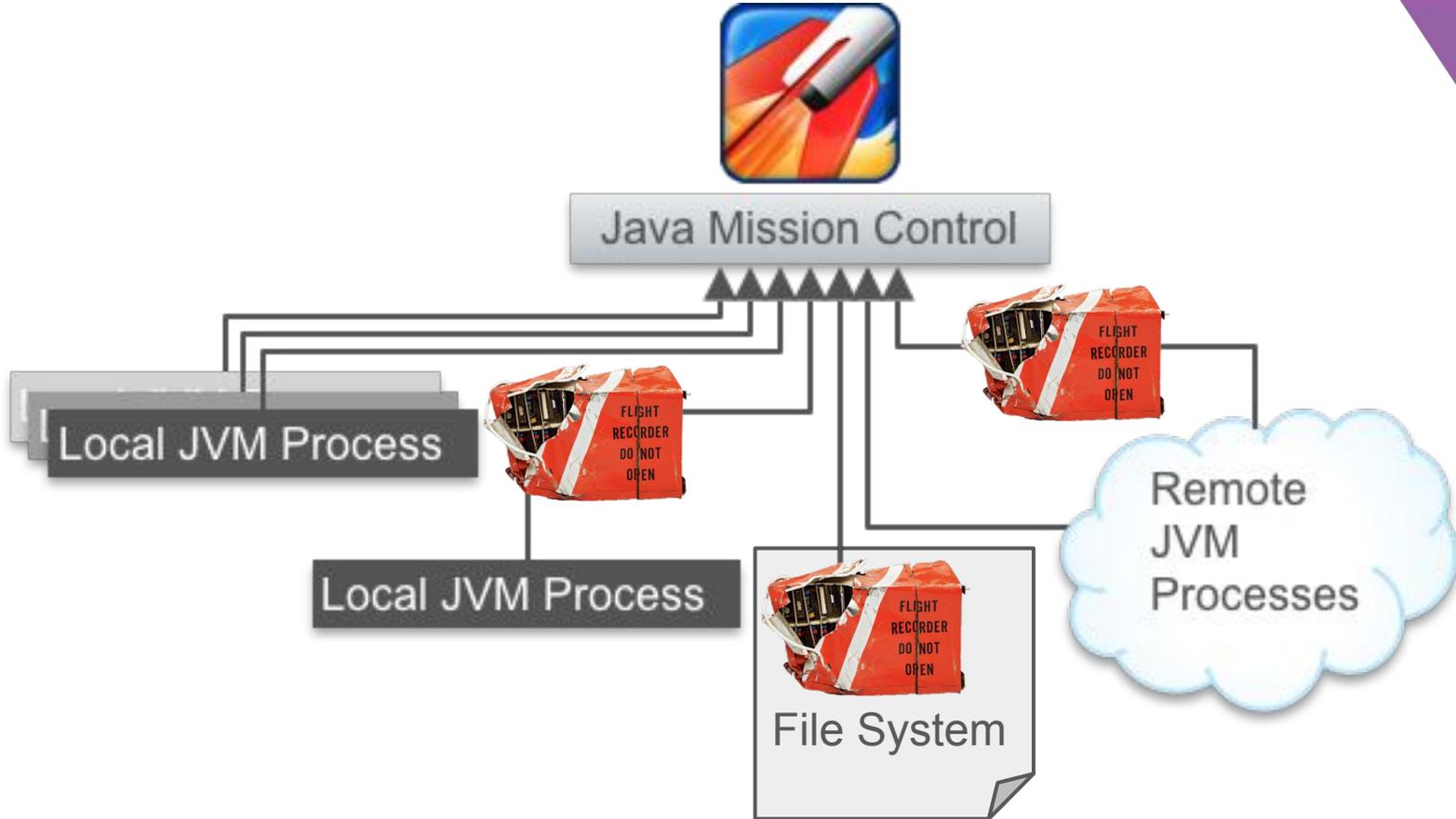


## Open Source

- OpenJDK project
- Actively worked on by Oracle, RedHat, DataDog, individual contributors...
  
- Several distributions of the application... and Eclipse plug-ins



# Where to get Flight Recordings





# Mission Control + Flight Recorder UI

- JFR Wizard for creating recordings
- Rules Overview
- Prebuilt pages
  - Method Profiling
  - Locks
  - Memory
  - ...
- Event Browser
  - All the events
- Custom pages
  - For JDK events or your custom events





DATADOG

# JMC Demos



## Extensive API

```
public static void main(String[] args) throws Exception {  
    var file = new File(args[0]);  
  
    ItemCollection events = JfrLoaderToolkit.loadEvents(file);  
    ItemCollection monitorEnterEvents = events.apply(JdkFilters.MONITOR_ENTER);  
  
    IQuantity eventCount = monitorEnterEvents.getAggregate(Aggregators.count());  
    IQuantity avg = monitorEnterEvents.getAggregate(Aggregators.avg((JfrAttributes.DURATION));  
    IQuantity stddev = monitorEnterEvents.getAggregate(Aggregators.stddev(JfrAttributes.DURATION));  
  
    System.out.println(String.format("# of events: %d, avg: %s, stddev: %s\n",  
        eventCount.longValue(),  
        avg.displayUsing(IDisplayable.AUTO),  
        stddev.displayUsing(IDisplayable.AUTO)));  
}
```





# HTML Report Example

```
public static void main(String[] args) throws Exception {
```

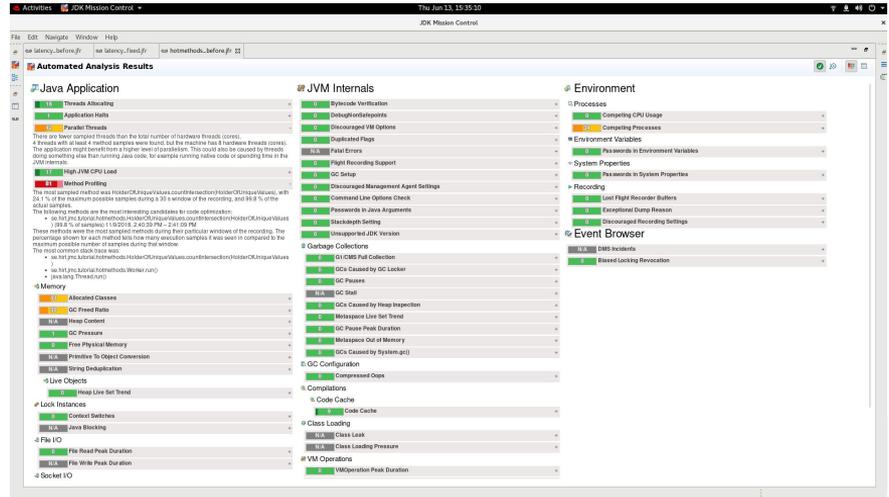
```
    var file = new File(args[0]);
```

```
    var recording = JfrLoaderToolkit.loadEvents(file);
```

```
    var report = JfrHtmlRulesReport.createReport(recording);
```

```
    System.out.println(report);
```

```
}
```





DATADOG

# JMC core Demos



# JMC Agent

- Incubation project in JMC
  - Not published (yet)
  - Build from source
- Declaratively insert JFR events anywhere

```
<event id="demo.jfr.MyEvent" >
  <name>My Awesome Event</ name>
  <description>This is the best event ever.</ description>
  <path>demo/jfr</ path>
  <stacktrace>>true</ stacktrace>
  <class>org.openjdk.jmc.agent.test.InstrumentMe</ class>
  <method>
    <name>myInstrumentedMethod</ name>
    <descriptor>(Lorg/openjdk/jmc/bciagent/test/Gurka;)V</ descriptor>
    <parameter index="0">
      <name>Gurka Attribute</ name>
      <description>The one and only Gurk-parameter</ description>
      <contenttype>None</ contenttype>
    </parameter>
  </method>
</event>
```





DATADOG

# Summary



## Resources

Join the project mailing list and start coding right now!

Repo: <https://github.com/openjdk/jmc>

Mailing list: <http://mail.openjdk.java.net/mailman/listinfo/jmc-dev>

Slack: <https://jdkmissioncontrol.slack.com>

Download JMC: Your distribution of choice (e.g. AdoptOpenJDK)

JMC Tutorial: <https://github.com/thegreystone/jmc-tutorial>

# Q & A

@jpbempel