Production Debugging

Jean-Philippe Bempel
@jpbempel
@jpbempel.bsky.social

Agenda

- Why debugging in production?
- JVM instrumentation
- Building a production debugger

Why debugging in production?

DATADOG

Observability

- Adding logs
- Adding spans
- Adding new attributes/tags for spans
- Adding metrics

CI/CD

- Restarting a service is effortless

- Introducing a new version is not effortless:
  - pushing change
  - build + test time (minutes to hours)
  - time to reach prod (staging/trains) (minutes to hours)
  - Think about reverting this or make it permanent

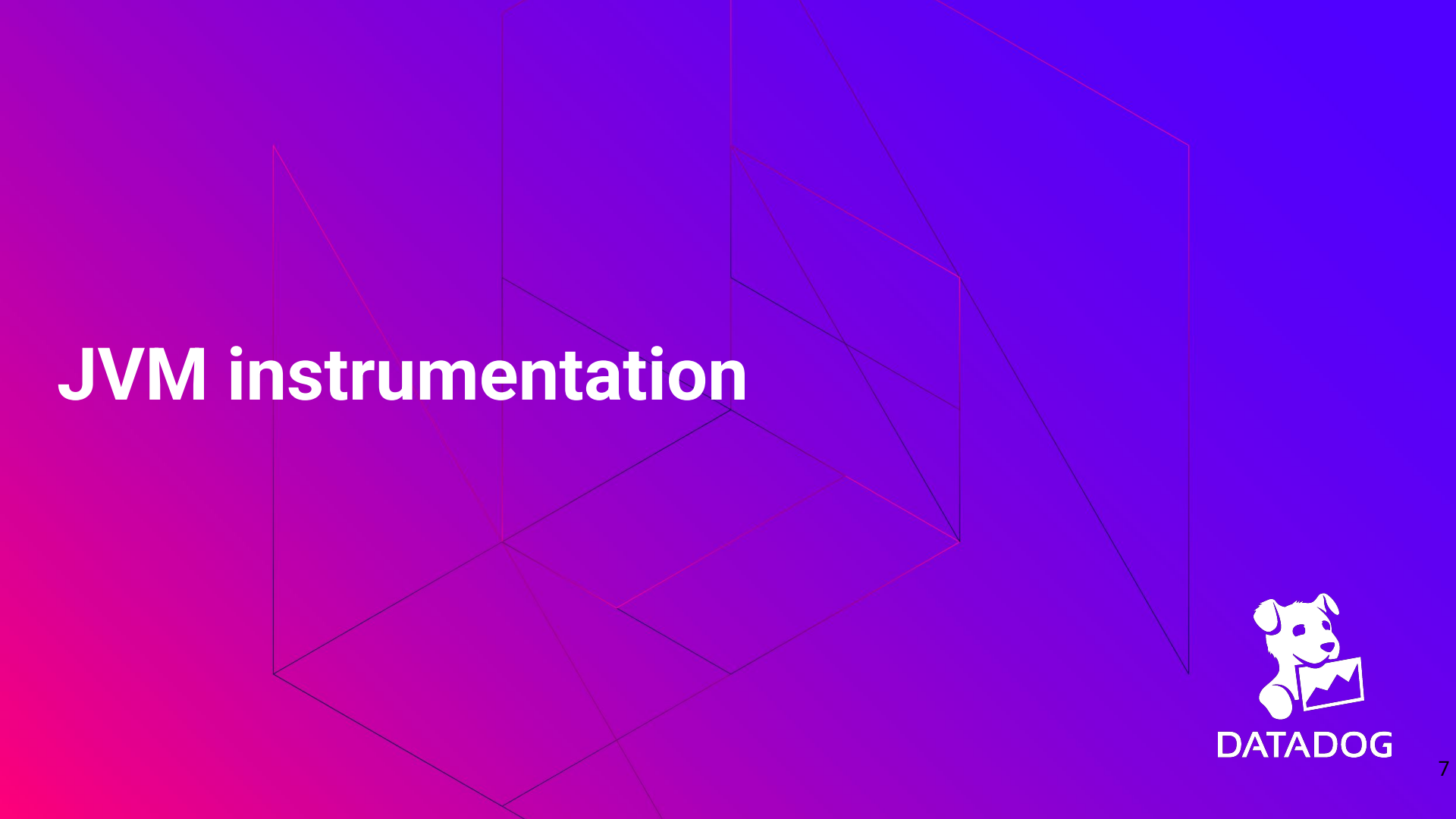
- Adding manually a new line of code could be painful

Production data

- Issue only happen in production
  - Higher volume/load, real data
  - Race conditions
  - State, caches

- Transient or temporary observability due to overhead

JVM instrumentation



DATADOG

Instrumentation API

- JVM feature since JDK 1.5
  - JVMTI
  - available on all JVM implementing the spec

- need to use java agent

```
java -javaagent:agent.jar
```

- java agent called before main method
  provide instance of Instrumentation

```
public static void premain(String args, Instrumentation inst) {
}
```

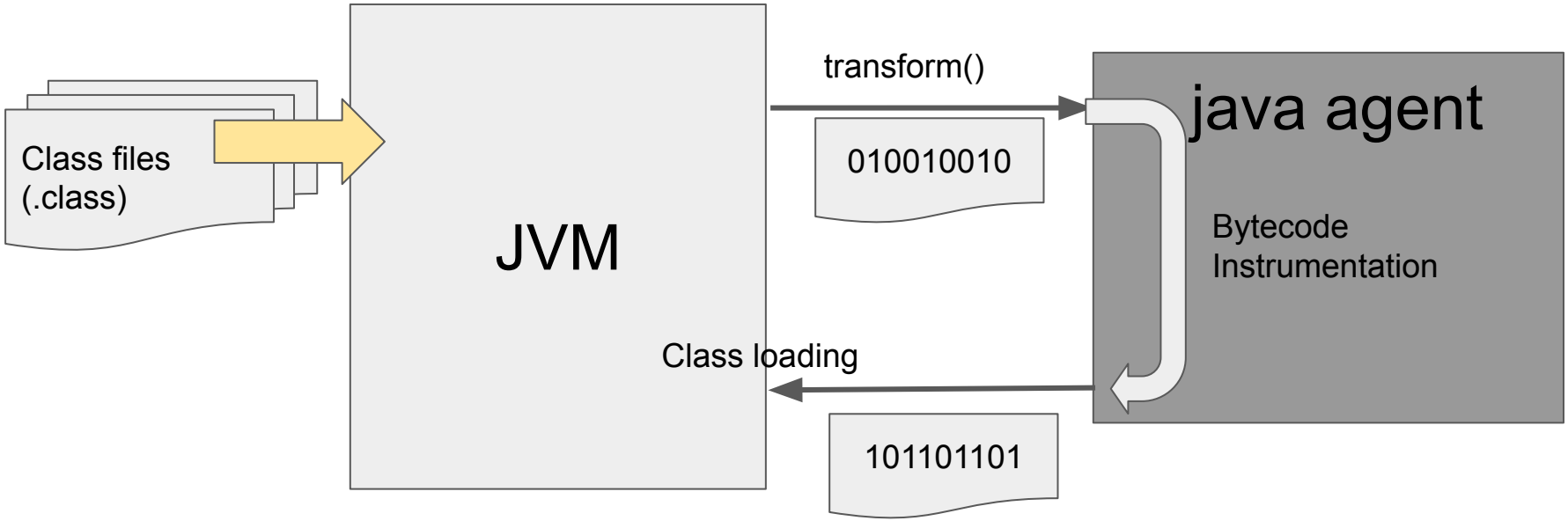
Instrumentation API (simplified)

```
public interface Instrumentation {
  void addTransformer(ClassFileTransformer transformer);
  boolean removeTransformer(ClassFileTransformer transformer);
  void retransformClasses(Class<?>... classes) throws UnmodifiableClassException;
  Class[] getAllLoadedClasses();
}
```

```
public interface ClassFileTransformer {
  byte[] transform(ClassLoader loader,
                   String className,
                   Class<?> classBeingRedefined,
                   ProtectionDomain protectionDomain,
                  byte[] classfileBuffer)
                   throws IllegalClassFormatException;
}
```

DATADOG

Java agent interaction



Retransformation

- Can instrument a class at any time
  - Just call `inst.retransformClasses(MyClass.class)`
  - Reloads the class from original classfile & calls all the `ClassFileTransformer::transform()`

- Some restrictions:
  - Can’t add/rename/remove fields
  - Can’t add/rename/remove/change sig methods

- Can add methods if done at class load time

Bytecode instrumentation

- ASM library
  - low level, Core & Tree API

- ByteBuddy
  - can build a full java agent for you
  - built on top of ASM
  - higher level API for bytecode manipulation

Building a production debugger

Adding a span

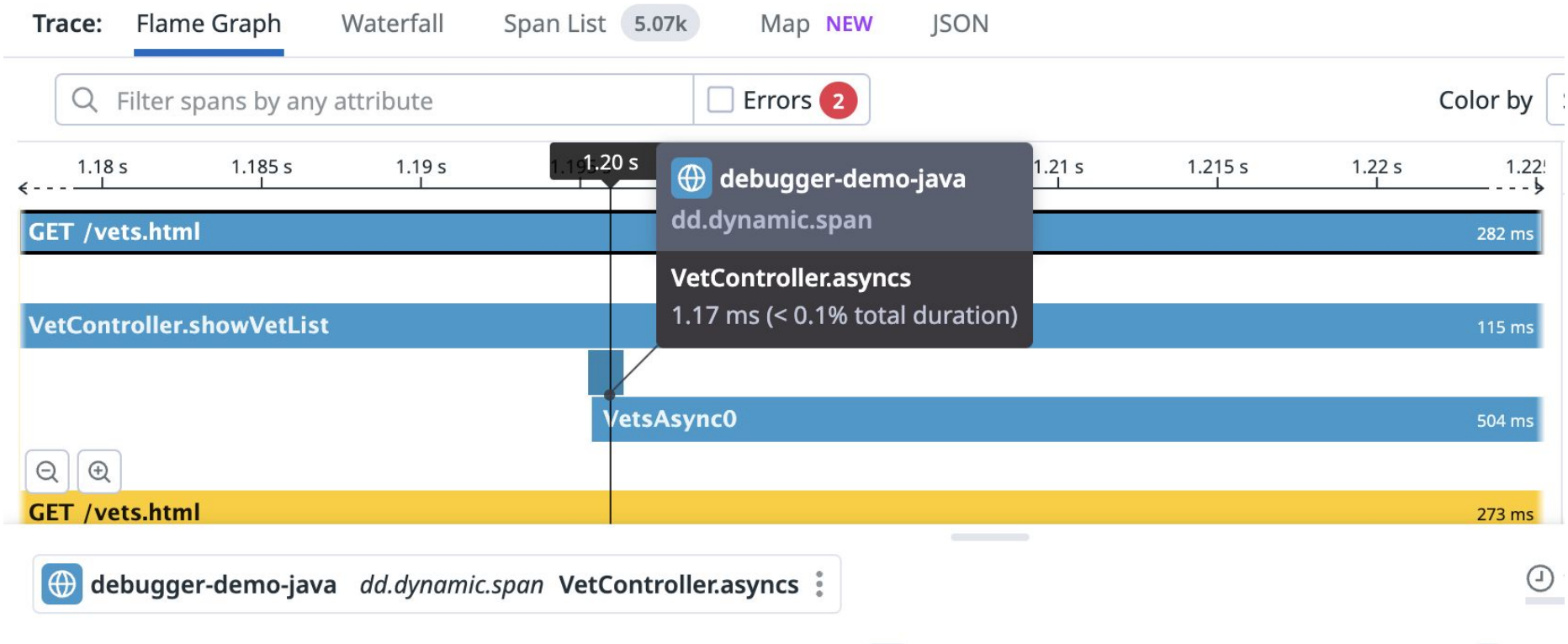
Adding manually an Otel span:

```
Span span = tracer
    .spanBuilder("span name")
    .startSpan();

try {
  // …
} finally {

  span.end();

}
```

Adding a span

1. Provide code location (Class + Method)
2. Load class or retransform
3. Matching in `ClassFileTransformer::transform`
4. Insert method calls into classfile buffer
5. execute the code

Dynamic span



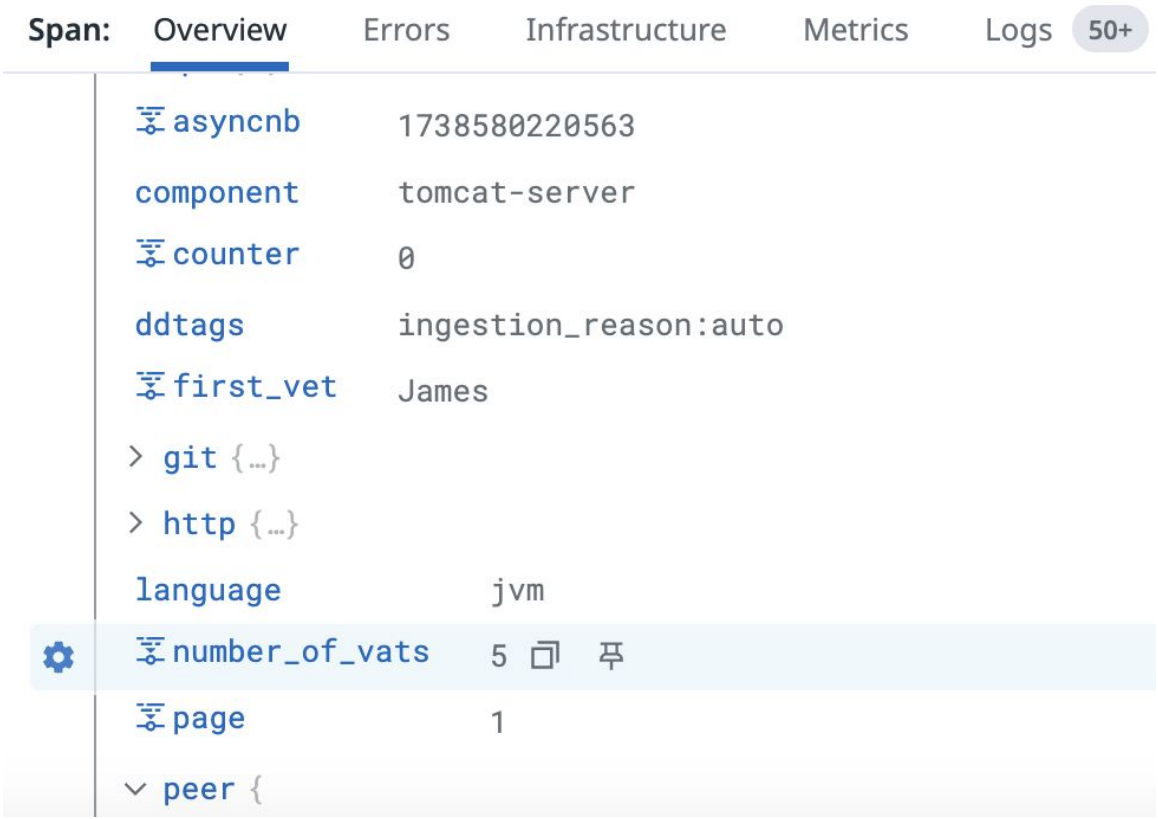
Adding a span attribute

Adding manually an Otel attribute for a the current span

```
Span.current().setAttribute("key", "value")

Span.current().setAttribute("name", this.name)

Span.current().setAttribute("arg", myArg)
```

Good way to pass specific simple values to a backend

Dynamic span attribute



Adding a log

Adding manually a log (Slf4J)

```
Logger LOGGER = LogFactory.getLogger("myLogger");

LOGGER.info("starting...");

LOGGER.info("Name={}", this.name);

LOGGER.info("myArg={}", myArg)
```

Dynamic log

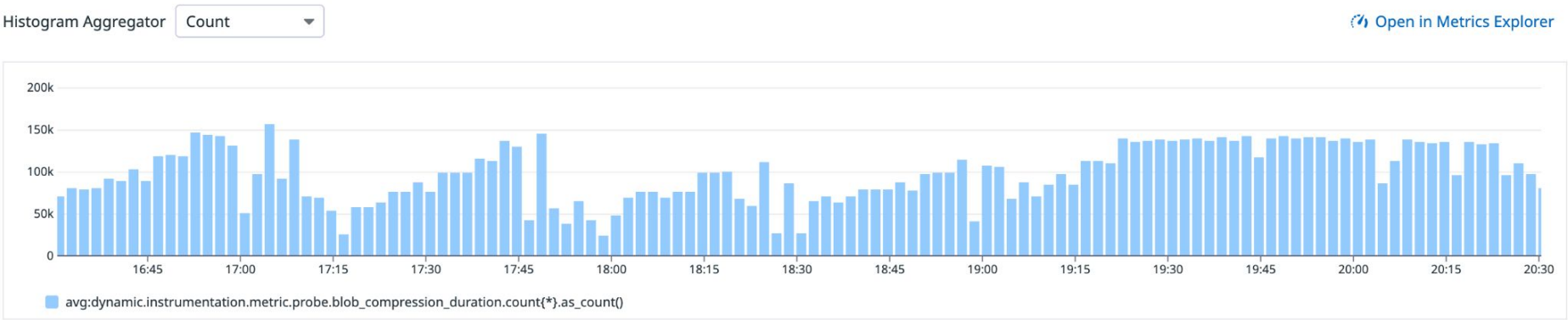
```
Feb 03 16:34:02.343  Hello new monitoring API
Feb 03 16:34:02.343  Hello new monitoring API
Feb 03 16:34:02.343  Executed showVetList, with page=1 and uuid=6eb50bce-0722-4d48-b560-f01e99bc85aa
Feb 03 16:34:02.336  In VetController.java, line 144
Feb 03 16:34:02.336  In VetController.java, line 144
```

Adding a metric

Adding manually a metric

```
Meter meter = MeterProvider.get("com.datadog")
LongCounter counter = meter
                        .counterBuilder("callCount")
                        .build();
counter.add(1L);

counter.add(list.size());
```

Dynamic metric



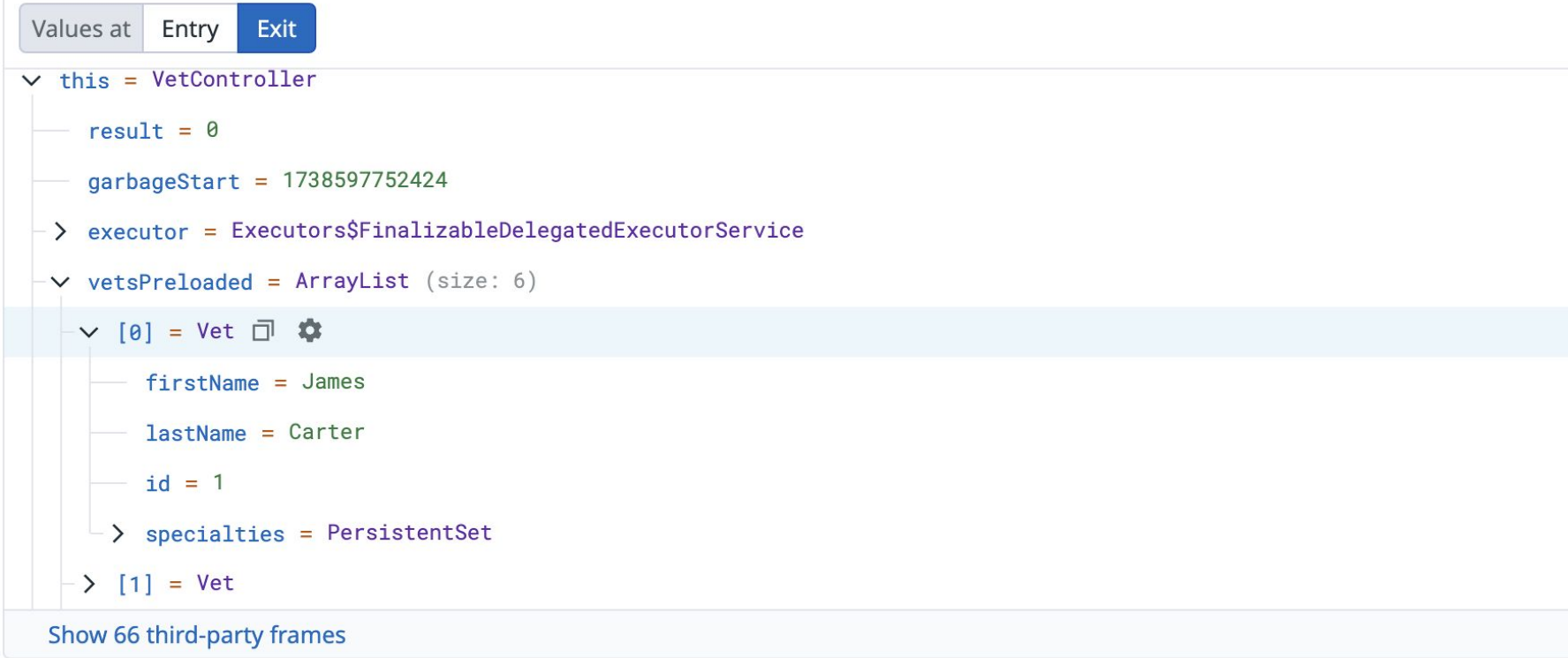
DATADOG

Capturing execution context

- At Code Location capture all:
  - arguments
  - fields
  - local var
  - return value
  - unhandled exception

- For each reference, capture sub-fields

Snapshot


Values at
Entry
Exit
this = VetController
result = 0
garbageStart = 1738597752424
executor = Executors$FinalizableDelegatedExecutorService
vetsPreloaded = ArrayList (size: 6)
[0] = Vet
firstName = James
lastName = Carter
id = 1
specialties = PersistentSet
[1] = Vet
Show 66 third-party frames

Capturing execution context

- Need to send data through a dedicated channel
- Great for exploration, state of execution
- Quickly need to interpret some known classes: Collections, Optional, …
- beware of the overhead to capture + serialize
- Security, leaking sensitive information

Sampling

- Depending on the code location or type of capture
- Sampling is good enough and required
- if executed once per request => no sampling
- if executed into tight loop => sampling like 1/s

Conditions

- Capture only if condition is true

- Can filter out heavily and bias the capture to erroneous cases ex:
  - arg < 0 (should never happen)
  - user == “john” (specific data, a priori knowledge)
  - method duration > 500ms (timeout)

Capture limits

- Depth (# ref follows in object graph)
- String length
- Collection count (size of lists, maps, …)
- Number of captured items (args, fields, local vars)

Exceptions

- Exceptions in Java have stacktraces, very useful!
- But how we end up in this exceptional situation? what was the value for `arg`?

```
void process(String arg) {
  if (arg.matches("^[a-z0-9]+")) {
    throw new InvalidArgumentException();
  }
}
```

Exceptions

- When exception happens:
  - Analyze the stacktrace
  - instrument n top frames

- Wait for next exception to happen

- When exception re-occurs:
  - Capture execution context
  - Correlate frames and captured data

References

- java.lang.instrument API
- Opentelemetry Java API
- ByteBuddy
- ASM
- The definitive guide to Java agents by Rafael Winterhalter
- Source code of Datadog debugger

Q&A

@jpbempel
@jpbempel.bsky.social