honeycomb.io

# Debug Production with OpenTelemetry

## A Primer for the Full-Stack Java/Spring Engineer

February 14 2025

# Ken Rimple

Senior Developer Relations Advocate

Honeycomb

TODO HANDLES HERE

# About me

- Full stack engineer
- Java and Spring background
- Angular, React, Next.js and other UI frameworks
- Argue with the cloud (AWS) and now AI
- Joined Honeycomb in 2024

# How does telemetry become observability?

# What is Observability?

The ability to understand the state of a system by observing its outputs

# Observability Signals

**Traces**

*“What happened to the code, in a directed, acyclic graph of events”*

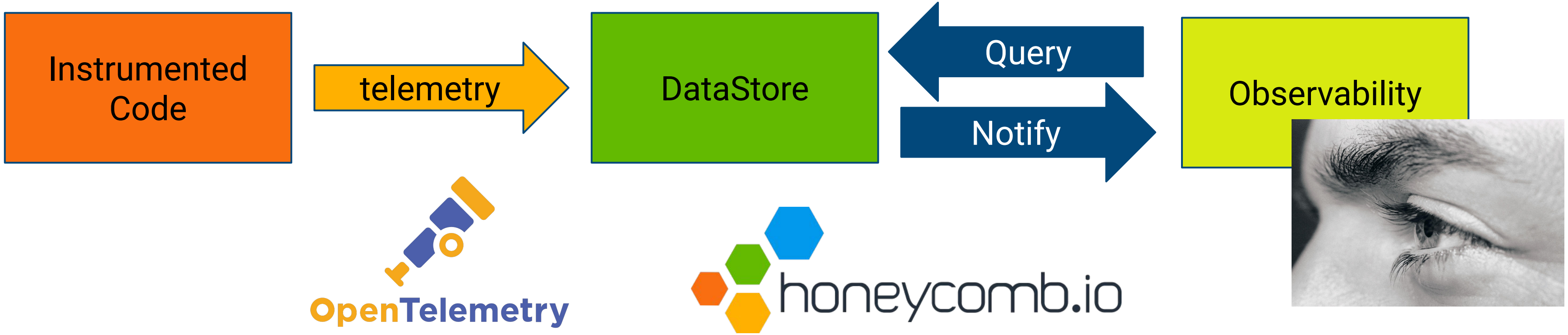
**Logs**

*“Messages sent from the code or framework, usually by logging APIs”*

**Metrics**

*“This many of those things happened, aggregated and reported on a schedule”*

# From Code to Observability in three steps



Instrumented Code

telemetry

DataStore

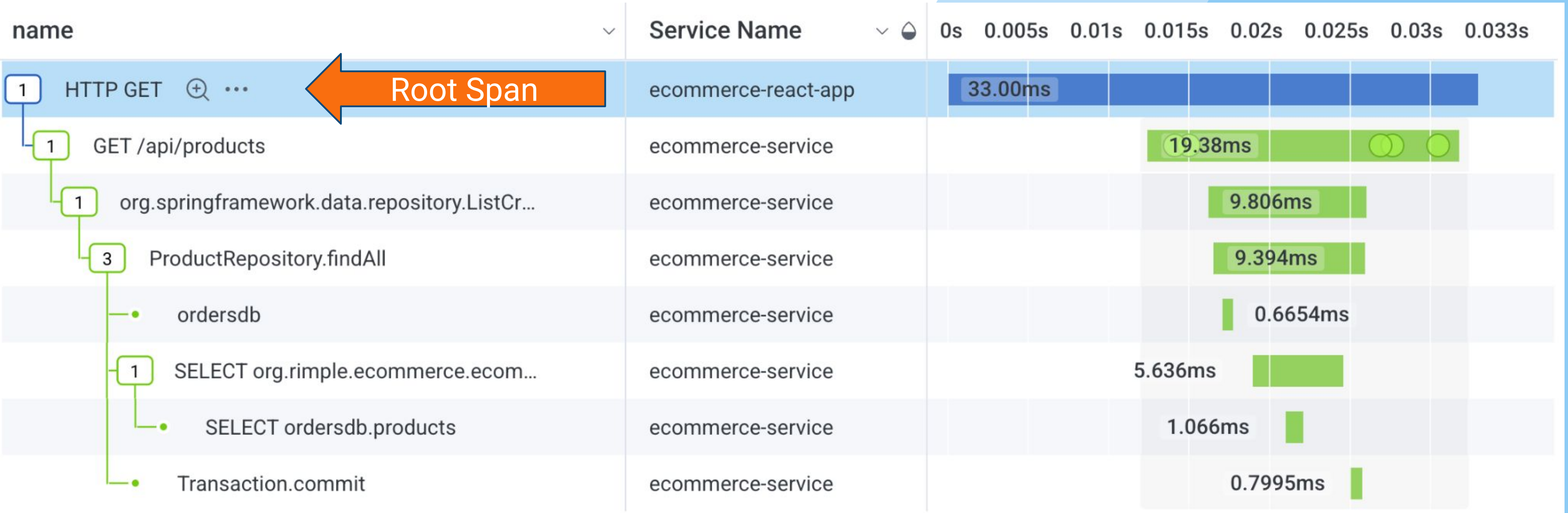
Query

Notify

Observability

# What is a trace?

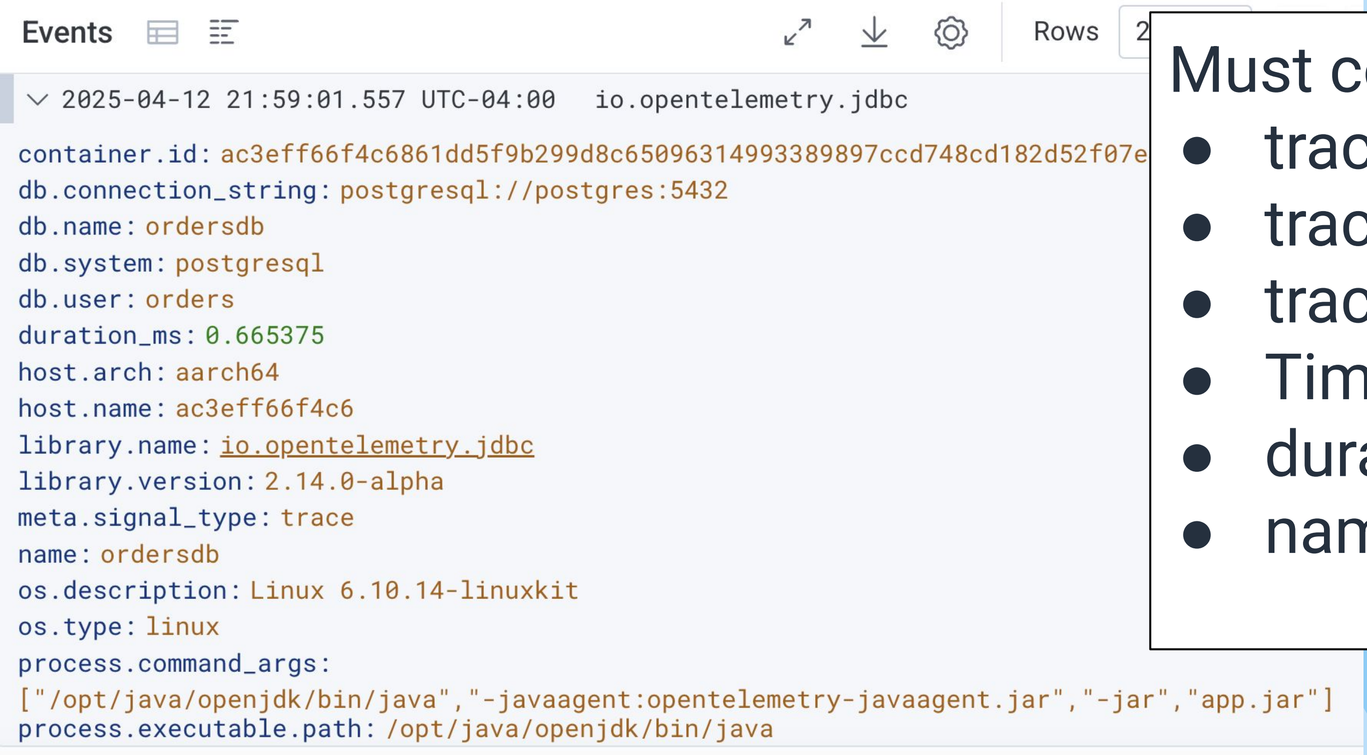
- A graph of spans, linked together by their span ids (trace.span_id = trace.parent_id)



Root Span

| name | Service Name | 0s 0.005s 0.01s 0.015s 0.02s 0.025s 0.03s 0.033s |
|---|---|---|
| 1 HTTP GET | ecommerce-react-app | 33.00ms |
| 1 GET /api/products | ecommerce-service | 19.38ms |
| 1 org.springframework.data.repository.ListCr... | ecommerce-service | 9.806ms |
| 3 ProductRepository.findAll | ecommerce-service | 9.394ms |
| ordersdb | ecommerce-service | 0.6654ms |
| 1 SELECT org.rimple.ecommerce.ecom... | ecommerce-service | 5.636ms |
| SELECT ordersdb.products | ecommerce-service | 1.066ms |
| Transaction.commit | ecommerce-service | 0.7995ms |

# Trace spans (meta.signal_type = trace)

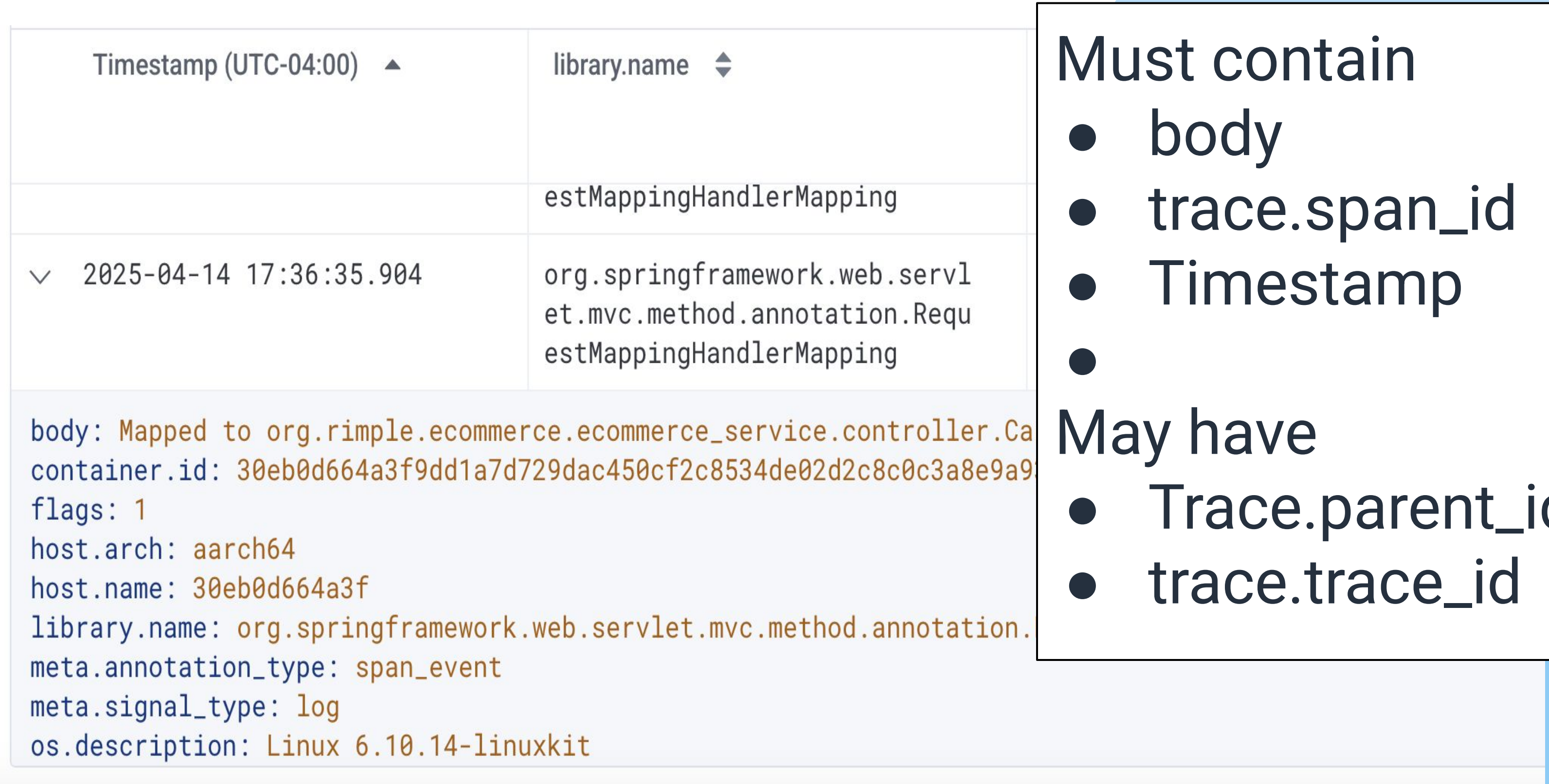


```
2025-04-12 21:59:01.557 UTC-04:00   io.opentelemetry.jdbc
container.id: ac3eff66f4c6861dd5f9b299d8c650963149933898 97ccd748cd182d52f07e
db.connection_string: postgresql://postgres:5432
db.name: ordersdb
db.system: postgresql
db.user: orders
duration_ms: 0.665375
host.arch: aarch64
host.name: ac3eff66f4c6
library.name: io.opentelemetry.jdbc
library.version: 2.14.0-alpha
meta.signal_type: trace
name: ordersdb
os.description: Linux 6.10.14-linuxkit
os.type: linux
process.command_args:
["/opt/java/openjdk/bin/java","-javaagent:opentelemetry-javaagent.jar","-jar","app.jar"]
process.executable.path: /opt/java/openjdk/bin/java
```

Must contain

- trace.trace_id
- trace.span_id
- trace.parent_id
- Timestamp
- duration_ms
- name

# Log spans (meta.signal_type = log)



| Timestamp (UTC-04:00) ▲ | library.name ⬍ |
|---|---|
| | estMappingHandlerMapping |
| 2025-04-14 17:36:35.904 | org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandlerMapping |

```
body: Mapped to org.rimple.ecommerce.ecommerce_service.controller.Ca
container.id: 30eb0d664a3f9dd1a7d729dac450cf2c8534de02d2c8c0c3a8e9a9
flags: 1
host.arch: aarch64
host.name: 30eb0d664a3f
library.name: org.springframework.web.servlet.mvc.method.annotation.
meta.annotation_type: span_event
meta.signal_type: log
os.description: Linux 6.10.14-linuxkit
```

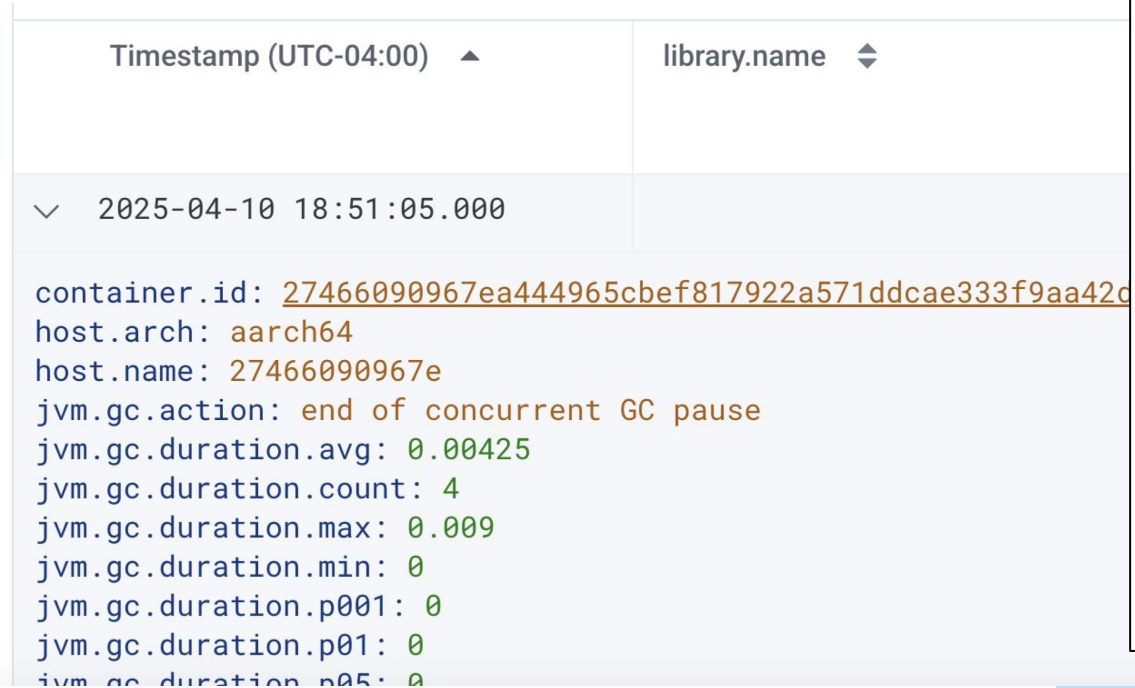
Must contain

- body
- trace.span_id
- Timestamp
-

May have

- Trace.parent_id
- trace.trace_id

# Metrics(meta.signal_type = metric)



Timestamp (UTC-04:00) library.name

2025-04-10 18:51:05.000

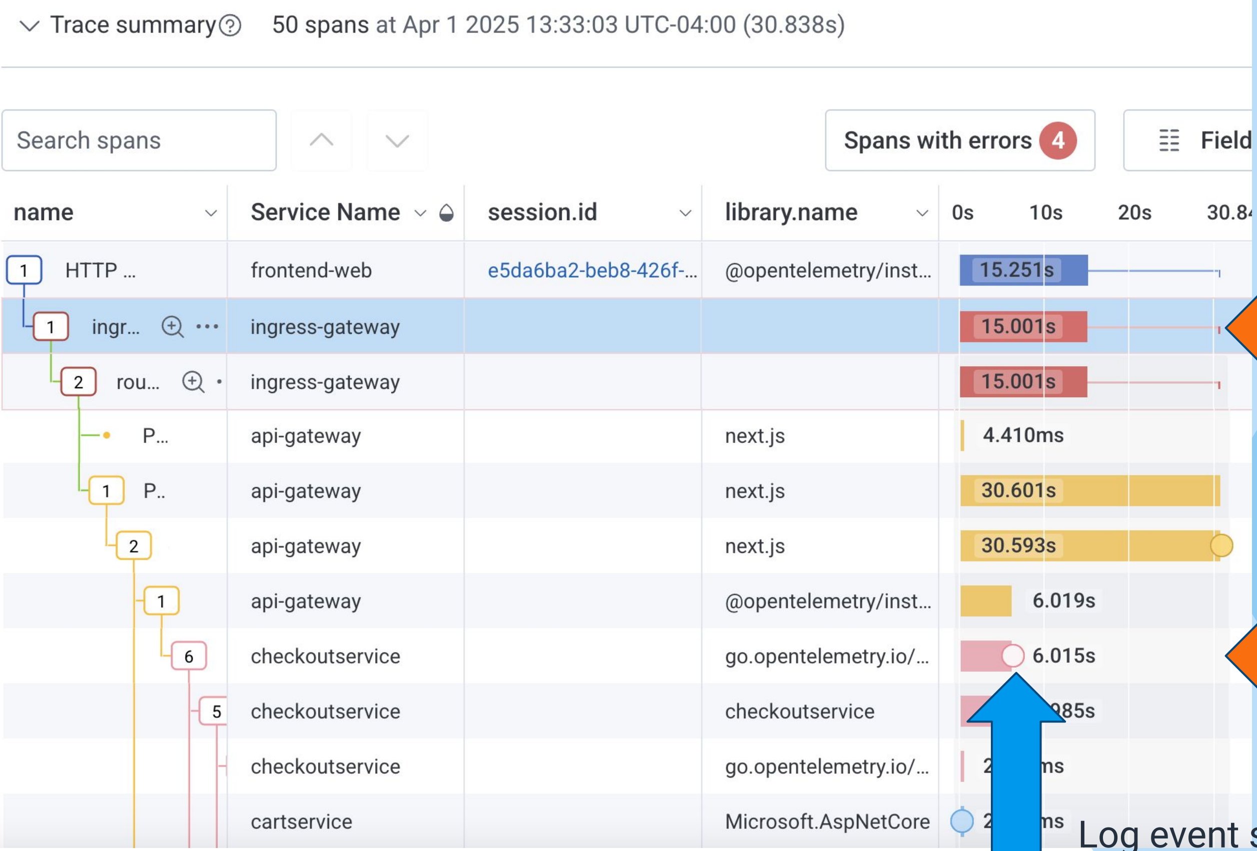
```
container.id: 27466090967ea444965cbef817922a571ddcae333f9aa42c
host.arch: aarch64
host.name: 27466090967e
jvm.gc.action: end of concurrent GC pause
jvm.gc.duration.avg: 0.00425
jvm.gc.duration.count: 4
jvm.gc.duration.max: 0.009
jvm.gc.duration.min: 0
jvm.gc.duration.p001: 0
jvm.gc.duration.p01: 0
jvm.gc.duration.p05: 0
```

Must contain

- Timestamp

Unlike trace, log spans these are generally pre-aggregated

# Example: trace with many microservices



Trace summary 50 spans at Apr 1 2025 13:33:03 UTC-04:00 (30.838s)

Search spans

Spans with errors 4

Field

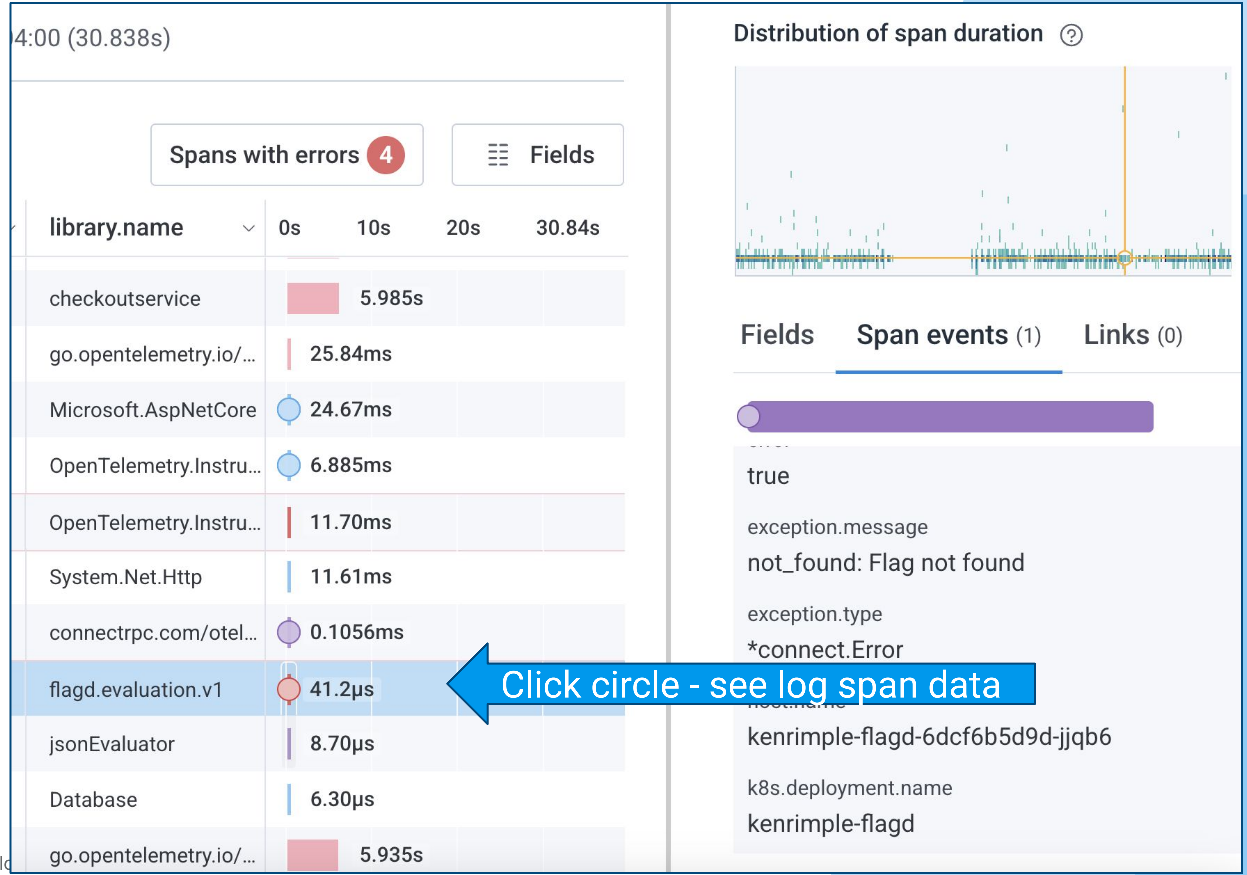
| name | Service Name | session.id | library.name | 0s 10s 20s 30.8 |
|---|---|---|---|---|
| 1 HTTP ... | frontend-web | e5da6ba2-beb8-426f-... | @opentelemetry/inst... | 15.251s |
| 1 ingr... | ingress-gateway | | | 15.001s |
| 2 rou... | ingress-gateway | | | 15.001s |
| P... | api-gateway | | next.js | 4.410ms |
| 1 P.. | api-gateway | | next.js | 30.601s |
| 2 | api-gateway | | next.js | 30.593s |
| 1 | api-gateway | | @opentelemetry/inst... | 6.019s |
| 6 | checkoutservice | | go.opentelemetry.io/... | 6.015s |
| 5 | checkoutservice | | checkoutservice | 085s |
| | checkoutservice | | go.opentelemetry.io/... | 2 ms |
| | cartservice | | Microsoft.AspNetCore | 2 ms |

Errors

Errors

Log event spans (click to view)

# Example log span event



| library.name | 0s 10s 20s 30.84s |
|---|---|
| checkoutservice | 5.985s |
| go.opentelemetry.io/... | 25.84ms |
| Microsoft.AspNetCore | 24.67ms |
| OpenTelemetry.Instru... | 6.885ms |
| OpenTelemetry.Instru... | 11.70ms |
| System.Net.Http | 11.61ms |
| connectrpc.com/otel... | 0.1056ms |
| flagd.evaluation.v1 | 41.2µs |
| jsonEvaluator | 8.70µs |
| Database | 6.30µs |
| go.opentelemetry.io/... | 5.935s |

true

exception.message
not_found: Flag not found

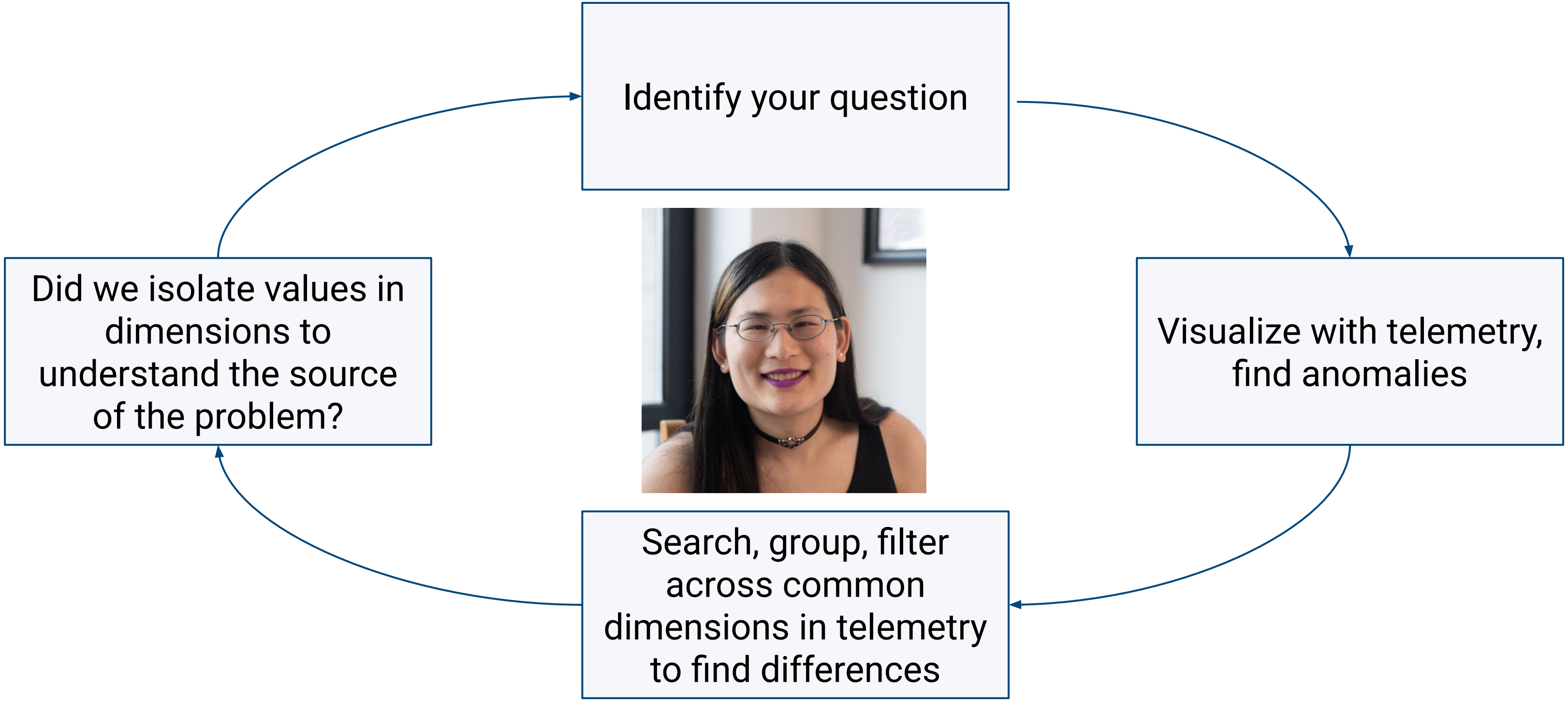
exception.type
*connect.Error

kenrimple-flagd-6dcf6b5d9d-jjqb6

k8s.deployment.name
kenrimple-flagd

Click circle - see log span data

# The Observability Core Analysis Loop



Identify your question

Visualize with telemetry, find anomalies

Search, group, filter across common dimensions in telemetry to find differences

Did we isolate values in dimensions to understand the source of the problem?

# OpenTelemetry SDK Configurations for Java

- OpenTelemetry Java Agent
- Spring Boot Starter

# Auto instrumentation with the Java  Agent

```
export OTEL_EXPORTER_OTLP_ENDPOINT=https://api.honeycomb.io:443
export OTEL_EXPORTER_OTLP_PROTOCOL=http/protobuf
export OTEL_EXPORTER_OTLP_HEADERS="x-honeycomb-team=${HONEYCOMB_API_KEY}"
export OTEL_SERVICE_NAME="ecommerce-service"

java -javaagent:opentelemetry-javaagent.jar  -jar app.jar
```

- Uses environment variables to configure the agent
- The agent automatically instruments based on a wide range of libraries
- This instrumentation includes traces, logs, and metrics by default
- The instrumentation can be configured on the Java agent with environment variables, flags, even on individual libraries

# Types of Instrumentation

- Automatic
  - Performed by instrumentation libraries
  - Based on configuration in OpenTelemetry SDKs
  - Varies based on language and framework
  - “Get me started quickly!”

- Manual
  - You add information you care about to your telemetry

# Why do you need manual instrumentation?

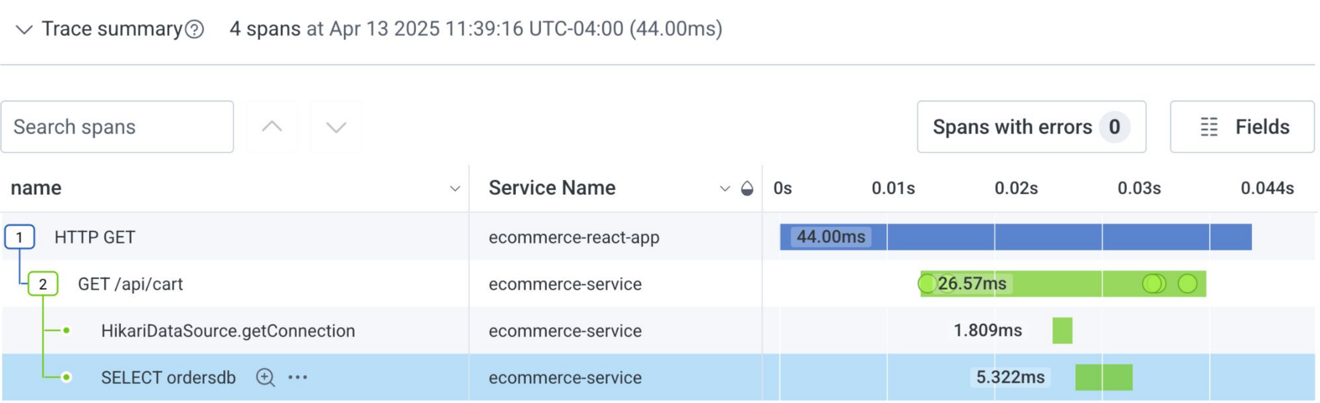
- To measure business objectives
- To capture complex processes
- To handle novel events
- To enrich spans with useful details

# Adding Spans with the OpenTelemetry API

```
var openTelemetry = GlobalOpenTelemetry.get();
var tracer = openTelemetry.getTracer(“chat-service”);
```

# Default tracing level for Spring Starter

- Configures less tracing out of the box than the standard Java Otel Agent approach
- Does not instrument all spring beans…
- Example below: captures the endpoint and then database API call



Trace summary 4 spans at Apr 13 2025 11:39:16 UTC-04:00 (44.00ms)

Search spans

Spans with errors 0

Fields

| name | Service Name | 0s 0.01s 0.02s 0.03s 0.044s |
|---|---|---|
| 1 HTTP GET | ecommerce-react-app | 44.00ms |
| 2 GET /api/cart | ecommerce-service | 26.57ms |
| HikariDataSource.getConnection | ecommerce-service | 1.809ms |
| SELECT ordersdb | ecommerce-service | 5.322ms |

# Adding Spans with brute force – Aspects

- **USE SPARINGLY!!!** Can create a lot of spans, spans are the unit of cost
- This example uses AOP Around Advice with a pointcut - too wide, and you get a TON of spans

```
@Component
@Aspect
public class MethodTracingAspect {
private final Tracer tracer;

  @Autowired
  public MethodTracingAspect(OpenTelemetry openTelemetry) {
    this.tracer = openTelemetry.getTracer("ecommerce-service");
  }

  @Around("execution(* org.rimple.ecommerce.ecommerce_service..*(..))")
  public Object traceMethod(ProceedingJoinPoint pjp) throws Throwable {
    // instrumentation here
  }
```

# Creating a Span in the Aspect traceMethod

```
Span span = tracer.spanBuilder(methodSig.getName())
    .setAttribute("method.name", methodName)
    .startSpan();

try (Scope scope = span.makeCurrent()) {
  span.setAttribute("method.args", Arrays.toString(pjp.getArgs()));
  Object result = pjp.proceed();
  span.setStatus(StatusCode.OK);
  return result;
} catch (Throwable t) {
  span.recordException(t);
  span.setStatus(StatusCode.ERROR, "Exception: " + t.getMessage());
  throw t;
} finally {
  span.end();
}
```

# Now, DON’T DO THAT

- Proliferates spans anywhere the pointcut matches
- You want to instrument the novel, not the expected

# Adding spans with @Span annotation

```
@WithSpan(value = "updateItemQuantity")
@PostMapping("/items/{productId}")
public Cart updateItemQuantity(
   @RequestHeader("X-User-ID") String userId,
   @PathVariable Long productId,
   @RequestBody CartOperationDTO operation) {

   Span currentSpan = Span.current();
   currentSpan.setAttribute(“app.user-id”, userId);
   currentSpan.setAttribute("app.product-id", productId);
   currentSpan.setAttribute("app.product-quantity", operation.getQuantity());
   currentSpan.setAttribute(“app.product-unit-price”,
                         operation.getUnitPrice());
   return cartService.updateQuantityInCart(
            userId, productId, operation.getQuantity()
   );
}
```

# Enriching a span with additional information

```
// from a Spring service bean below the controller
@Transactional
public Cart updateQuantityInCart(
                        String userId, Long productId, Integer quantity) {

    // Grab the existing span (from the controller)
    Span span = Span.current();

    ...
    if (quantity == 0) {
        cart.getItems().remove(hydratedItem);
        span.setAttribute("app.item.removed", true);
        return cartRepository.save(cart);
    }
    ...
}
```

# The Spring Boot Starter

- Uses Spring’s configuration, annotations, etc.
- Works with GraalVM binary compiled applications
- Can configure in Spring application configuration files

```
# application.yaml

otel:
  propagators: tracecontext
  resource:
    attributes:
      service:
        name: ecommerce-service
  instrumentation:
    # logback-appender:
    #   enabled: false
    # slf4j-simple:
    #   enabled: false
    common:
      experimental:
        controller:
          controller-telemetry: enabled
```

# Spring Boot OpenTelemetry Starter

- Add the relevant otel repository location
- Install the OpenTelemetry BOM
- Add OpenTelemetry Spring Boot starter
- Configure `application.properties|yaml` to taste

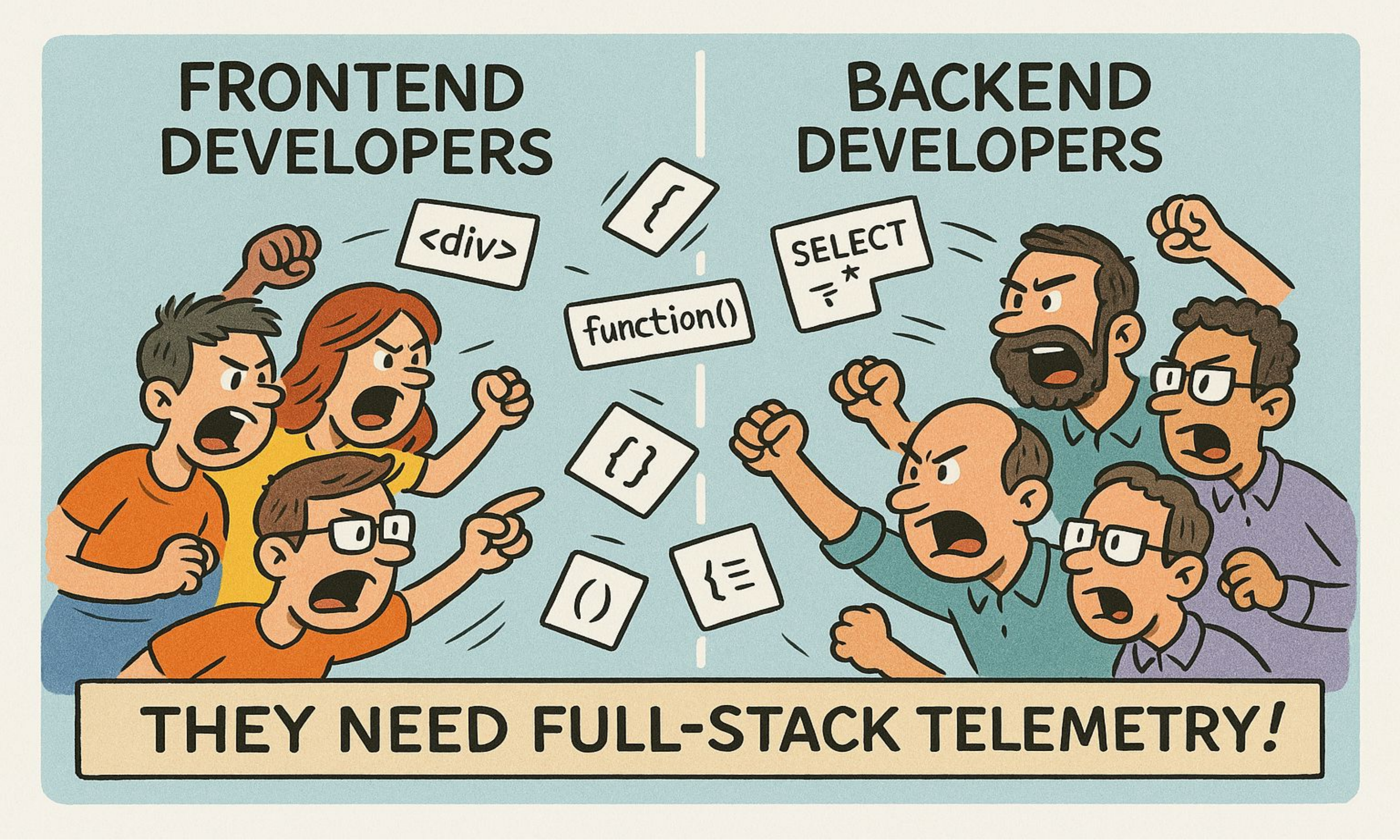
```yaml
# application.yaml

otel:
  propagators: tracecontext
  resource:
    attributes:
      service:
        name: ecommerce-service
  instrumentation:
    # logback-appender:
    #   enabled: false
    # slf4j-simple:
    #   enabled: false
    common:
      experimental:
        controller:
          controller-telemetry: enabled
```

# Otel JavaAgent -vs- Spring Boot Starter

| Approach | Pros | Cons |
|---|---|---|
| OTEL Java Agent | ● Good if you don’t own the source code<br>● Doesn’t require Spring<br>● Lots of instrumentation enabled by default | ● Not a native Spring experience<br>● Can only run one JavaAgent at a time |
| Spring Boot Starter | ● Native Spring setup and management<br>● Can run on GraalVMs<br>● No external agent code | ● Needs to be built<br>● Requires coding changes even to install |

# Frontend Observability


FRONTEND DEVELOPERS
BACKEND DEVELOPERS
<div>
{
function()
SELECT *
{}
()
{☰
THEY NEED FULL-STACK TELEMETRY!

# Instrumenting Browser Applications

- Install Honeycomb's OpenTelemetry library wrapper SDK
  - https://github.com/honeycombio/honeycomb-opentelemetry-web
  - Wraps the OpenTelemetry JavaScript SDK
  - Provides lots of helpful telemetry out of the box, including
    - Core Web Vitals
    - Browser Settings
    - Generated browser session IDs
    - Global catch-all error reporting
- Saves a lot of manual configuration, but still can be customized

# A simple example

```
const sdk = new HoneycombWebSDK({
    serviceName: ‘frontend-web’,
    instrumentations: [
        getWebAutoInstrumentations(),
    ],
});

sdk.start();
```

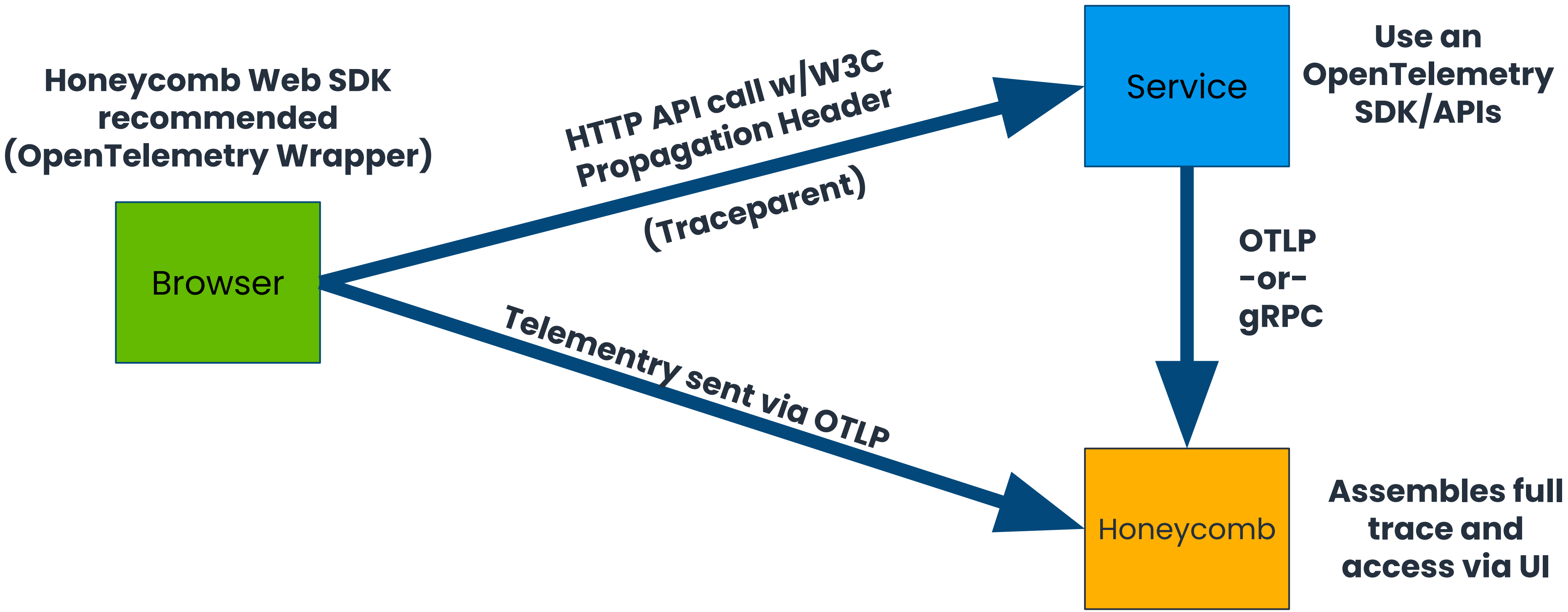
# Trace Propagation and Network Diagnostics

```
// configure settings for auto-instrumentation
// (except user-events)
const configDefaults = {
   ignoreNetworkEvents: true,
   propagateTraceHeaderCorsUrls: [  /.*/g  ]
}
```

# Applying defaults to instrumentation

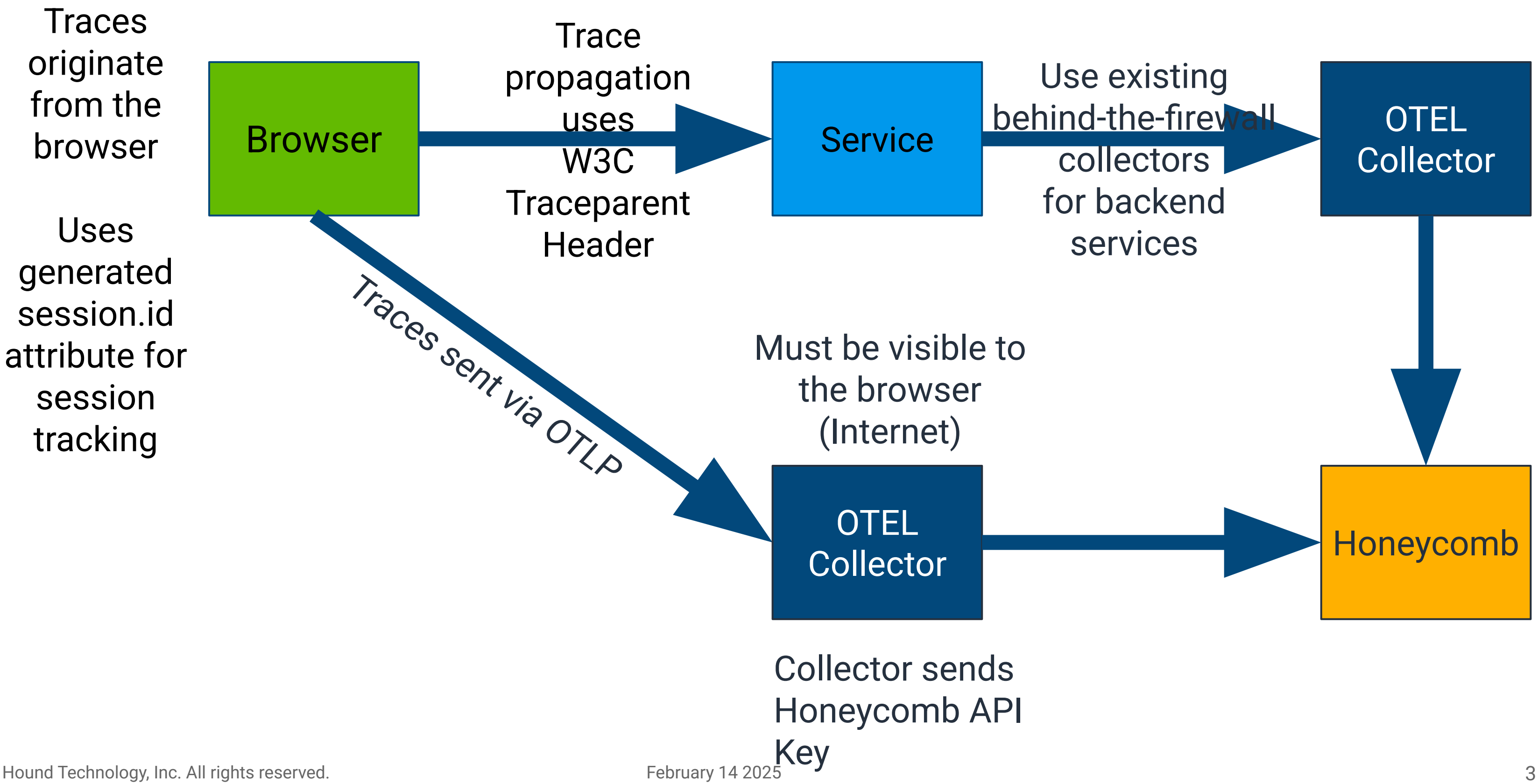
```
const sdk = new HoneycombWebSDK({
    serviceName: ‘frontend-web’,
    instrumentations: [
        getWebAutoInstrumentations({
            ‘@opentelemetry/instrumentation-fetch’: configDefaults,
            ‘@opentelemetry/instrumentation-document-load’, configDefaults,
            ‘@opentelemetry/instrumentation-xml-http-request’, configDefaults,
            ‘@opentelemetry/instrumentation-user-interaction’, {
                 enabled: true, eventNames: [‘click’, ‘submit’, ‘reset’]
            }
        }),
    ],
});

sdk.start();
```


Honeycomb Web SDK recommended (OpenTelemetry Wrapper)
Browser
HTTP API call w/W3C Propagation Header (Traceparent)
Service
Use an OpenTelemetry SDK/APIs
OTLP -or- gRPC
Telementry sent via OTLP
Honeycomb
Assembles full trace and access via UI

A full-stack trace with Honeycomb Frontend Observability

# Sending data to an OpenTelemetry Collector



Traces originate from the browser

Uses generated session.id attribute for session tracking

Browser

Trace propagation uses W3C Traceparent Header

Service

Use existing behind-the-firewall collectors for backend services

OTEL Collector

Traces sent via OTLP

Must be visible to the browser (Internet)

OTEL Collector

Honeycomb

Collector sends Honeycomb API Key

honeycomb.io

See everything. Solve anything.

honeycomb.io