1 OTCA

Generated on: 2026-01-13 15:04:57 **Version:** 1.0

1.1 Table of Contents

1.2 Overview

The **OpenTelemetry Certified Associate (OTCA)** exam demonstrates knowledge of observability concepts and OpenTelemetry implementation.

1.3 Exam Overview

Detail	Information
Exam Format	Multiple Choice
Number of Questions	60
Duration	90 minutes
Passing Score	75%
Certification Validity	3 years
Cost	$250 USD
Retake Policy	1 free retake

1.4 Exam Domains & Weights

Domain	Weight
Observability Concepts	16%
OpenTelemetry API	24%
OpenTelemetry SDK	16%
OpenTelemetry Collector	24%
Instrumentation	20%

1.5 Key Topics

1.5.1 Signals

- Traces, Metrics, Logs
- Context propagation
- Baggage

1.5.2 Collector

- Receivers, Processors, Exporters
- Pipeline configuration
- Deployment patterns

1.5.3 Instrumentation

- Auto-instrumentation
- Manual instrumentation
- SDK configuration

1.6 Study Resources

- OpenTelemetry Documentation
- OTCA Curriculum
- OpenTelemetry Demo

1.7 Navigation

- Next: Sample Questions →

1.8 OpenTelemetry Fundamentals

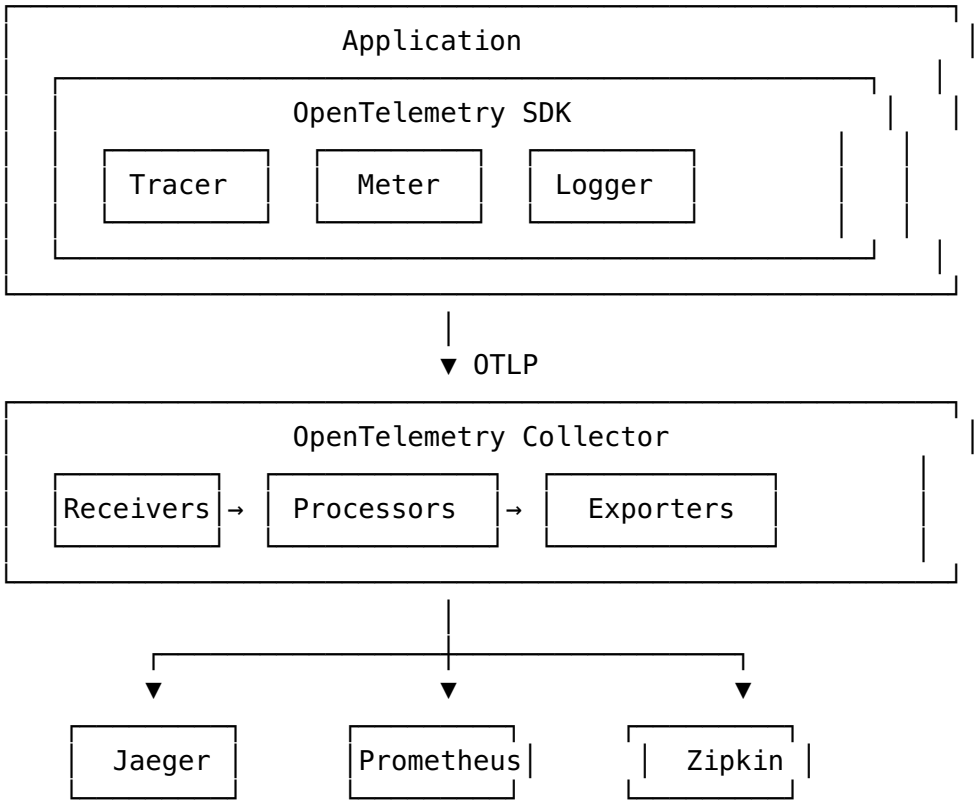
Comprehensive guide to OpenTelemetry for OTCA certification.

1.9 Overview

OpenTelemetry is a collection of tools, APIs, and SDKs for:

- **Traces** - Distributed tracing across services
- **Metrics** - Numerical measurements over time
- **Logs** - Structured log data (emerging)

1.10 Architecture



1.11 Key Concepts

1.11.1 Traces

- **Trace** - End-to-end request flow
- **Span** - Single operation within a trace
- **SpanContext** - Trace ID, Span ID, flags
- **Attributes** - Key-value pairs on spans

1.11.2 Metrics

- **Counter** - Monotonically increasing value
- **Gauge** - Current value at a point in time
- **Histogram** - Distribution of values

1.11.3 Context Propagation

- **W3C Trace Context** - Standard propagation format
- **Baggage** - User-defined key-value pairs

1.12 OpenTelemetry Collector

1.12.1 Configuration

```yaml
receivers:
  otlp:
    protocols:
      grpc:
        endpoint: 0.0.0.0:4317
      http:
        endpoint: 0.0.0.0:4318

processors:
  batch:
    timeout: 1s
    send_batch_size: 1024
  memory_limiter:
    check_interval: 1s
    limit_mib: 1000

exporters:
  logging:
    loglevel: debug
  jaeger:
    endpoint: jaeger:14250
    tls:
      insecure: true
  prometheus:
    endpoint: "0.0.0.0:8889"

service:
  pipelines:
    traces:
      receivers: [otlp]
      processors: [memory_limiter, batch]
      exporters: [jaeger, logging]
    metrics:
      receivers: [otlp]
      processors: [batch]
      exporters: [prometheus]
```

1.12.2 Deployment

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: otel-collector
spec:
  replicas: 1
  selector:
    matchLabels:
```

```
      app: otel-collector
  template:
    metadata:
      labels:
        app: otel-collector
    spec:
      containers:
      - name: collector
        image: otel/opentelemetry-collector:latest
        ports:
        - containerPort: 4317  # OTLP gRPC
        - containerPort: 4318  # OTLP HTTP
        - containerPort: 8889  # Prometheus metrics
        volumeMounts:
        - name: config
          mountPath: /etc/otel
      volumes:
      - name: config
        configMap:
          name: otel-collector-config
```

1.13 Instrumentation

1.13.1 Python

```
from opentelemetry import trace
from opentelemetry.sdk.trace import TracerProvider
from opentelemetry.sdk.trace.export import BatchSpanProcessor
from opentelemetry.exporter.otlp.proto.grpc.trace_exporter import
        OTLPSpanExporter

# Setup
trace.set_tracer_provider(TracerProvider())
tracer = trace.get_tracer(__name__)

# Export to collector
otlp_exporter = OTLPSpanExporter(endpoint="localhost:4317",
        insecure=True)
span_processor = BatchSpanProcessor(otlp_exporter)
trace.get_tracer_provider().add_span_processor(span_processor)

# Create spans
with tracer.start_as_current_span("main") as span:
    span.set_attribute("user.id", "12345")
    with tracer.start_as_current_span("child"):
        # Do work
        pass
```

1.13.2 Go

```go
package main

import (
    "context"
    "go.opentelemetry.io/otel"
    "go.opentelemetry.io/otel/exporters/otlp/otlptrace/
        otlptracegrpc"
    "go.opentelemetry.io/otel/sdk/trace"
)

func main() {
    ctx := context.Background()

    // Create exporter
    exporter, _ := otlptracegrpc.New(ctx,
        otlptracegrpc.WithEndpoint("localhost:4317"),
        otlptracegrpc.WithInsecure(),
    )

    // Create provider
    tp := trace.NewTracerProvider(
        trace.WithBatcher(exporter),
    )
    otel.SetTracerProvider(tp)

    // Create tracer
    tracer := otel.Tracer("myapp")

    // Create span
    ctx, span := tracer.Start(ctx, "operation")
    defer span.End()
}
```

1.13.3 Auto-instrumentation

```
# Python
pip install opentelemetry-distro opentelemetry-exporter-otlp
opentelemetry-bootstrap -a install
opentelemetry-instrument python myapp.py

# Java
java -javaagent:opentelemetry-javaagent.jar \
  -Dotel.service.name=myapp \
  -Dotel.exporter.otlp.endpoint=http://localhost:4317 \
  -jar myapp.jar
```

1.14 Semantic Conventions

Standard attribute names:

```
# Service
service.name: myapp
service.version: 1.0.0

# HTTP
http.method: GET
http.url: https://example.com/api
http.status_code: 200

# Database
db.system: postgresql
db.name: mydb
db.statement: SELECT * FROM users

# Messaging
messaging.system: kafka
messaging.destination: my-topic
```

1.15 Best Practices

1. **Use semantic conventions** for consistent attributes
2. **Sample appropriately** to manage data volume
3. **Propagate context** across service boundaries
4. **Add meaningful attributes** for debugging
5. **Use batch processors** for efficiency

← Back to OTCA Overview

1.16 Sample Practice Questions

1.17 Practice Resources

- OpenTelemetry Documentation
- OpenTelemetry Demo

1.18 Observability Concepts (16%)

1.18.1 Question 1

What are the three pillars of observability in OpenTelemetry?

Show Solution

1. **Traces** - Distributed tracing showing request flow
2. **Metrics** - Numerical measurements over time
3. **Logs** - Timestamped text records of events

OpenTelemetry also supports **Baggage** for context propagation.

1.18.2 Question 2

What is the difference between a Trace and a Span?

Show Solution

- **Trace** - Complete journey of a request through a distributed system, identified by a unique trace ID
- **Span** - Single operation within a trace, with start time, duration, and attributes

A trace contains multiple spans in a parent-child hierarchy:

```
Trace (trace_id: abc123)
├── Span A (root span)
│   ├── Span B (child of A)
│   └── Span C (child of A)
│       └── Span D (child of C)
```

1.18.3 Question 3

What is context propagation?

Show Solution

Context propagation passes trace context between services to correlate spans across service boundaries.

Components: - **Context** - Carries trace ID, span ID, trace flags - **Propagators** - Inject/extract context from carriers (HTTP headers) - **W3C Trace Context** - Standard format for propagation

Example headers:

```
traceparent: 00-<trace-id>-<span-id>-<flags>
tracestate: vendor1=value1,vendor2=value2
```

1.19 OpenTelemetry API (24%)

1.19.1 Question 4

How do you create a span manually?

Show Solution

```python
from opentelemetry import trace

tracer = trace.get_tracer(__name__)

with tracer.start_as_current_span("my-operation") as span:
    span.set_attribute("key", "value")
    span.add_event("processing started")
    # Do work
    span.set_status(trace.Status(trace.StatusCode.OK))
```

```java
Tracer tracer = GlobalOpenTelemetry.getTracer("my-app");
Span span = tracer.spanBuilder("my-operation").startSpan();
try (Scope scope = span.makeCurrent()) {
    span.setAttribute("key", "value");
    // Do work
} finally {
    span.end();
}
```

1.19.2 Question 5

How do you record metrics with OpenTelemetry?

Show Solution

```python
from opentelemetry import metrics

meter = metrics.get_meter(__name__)

# Counter - only increases
counter = meter.create_counter("requests_total")
counter.add(1, {"method": "GET"})

# UpDownCounter - can increase or decrease
gauge = meter.create_up_down_counter("active_connections")
gauge.add(1)
gauge.add(-1)

# Histogram - distribution of values
histogram = meter.create_histogram("request_duration")
histogram.record(0.5, {"endpoint": "/api"})
```

1.19.3 Question 6

What are span attributes vs span events?

Show Solution

Attributes - Key-value pairs describing the span - Set once, describe the operation - Examples: http.method, http.url, db.system

Events - Timestamped logs within a span - Can have multiple events per span - Examples: “cache miss”, “retry attempt”

```
span.set_attribute("http.method", "GET")
span.add_event("cache_miss", {"key": "user:123"})
```

1.20 OpenTelemetry SDK (16%)

1.20.1 Question 7

What are the main SDK components?

Show Solution

Trace SDK: - TracerProvider - Creates tracers - SpanProcessor - Processes spans (batch, simple) - SpanExporter - Exports spans to backends

Metrics SDK: - MeterProvider - Creates meters - MetricReader - Reads metrics periodically - MetricExporter - Exports metrics

Common: - Resource - Describes the entity producing telemetry - Sampler - Decides which traces to sample

1.20.2 Question 8

Configure the OpenTelemetry SDK with OTLP exporter.

Show Solution

```
from opentelemetry import trace
from opentelemetry.sdk.trace import TracerProvider
from opentelemetry.sdk.trace.export import BatchSpanProcessor
from opentelemetry.exporter.otlp.proto.grpc.trace_exporter import
        OTLPSpanExporter
from opentelemetry.sdk.resources import Resource

resource = Resource.create({"service.name": "my-service"})

provider = TracerProvider(resource=resource)
processor =
        BatchSpanProcessor(OTLPSpanExporter(endpoint="localhost:4317"))
```

```
provider.add_span_processor(processor)
trace.set_tracer_provider(provider)
```

1.20.3 Question 9

What is sampling and what are the sampling strategies?

Show Solution

Sampling decides which traces to record to reduce overhead.

Strategies: - **AlwaysOn** - Record all traces - **AlwaysOff** - Record no traces - **TraceIdRatioBased** - Sample based on trace ID ratio - **ParentBased** - Follow parent's sampling decision

```
from opentelemetry.sdk.trace.sampling import TraceIdRatioBased

sampler = TraceIdRatioBased(0.1)  # Sample 10%
provider = TracerProvider(sampler=sampler)
```

1.21 OpenTelemetry Collector (24%)

1.21.1 Question 10

What are the main components of the OTel Collector?

Show Solution

1. **Receivers** - Accept data (OTLP, Jaeger, Prometheus)
2. **Processors** - Transform data (batch, filter, attributes)
3. **Exporters** - Send data to backends (OTLP, Jaeger, Prometheus)
4. **Extensions** - Additional capabilities (health check, pprof)
5. **Connectors** - Connect pipelines (count spans to metrics)

Pipeline: Receivers → Processors → Exporters

1.21.2 Question 11

Write a basic Collector configuration.

Show Solution

```
receivers:
  otlp:
    protocols:
      grpc:
        endpoint: 0.0.0.0:4317
      http:
        endpoint: 0.0.0.0:4318
```

```yaml
processors:
  batch:
    timeout: 10s
    send_batch_size: 1000
  memory_limiter:
    limit_mib: 512

exporters:
  otlp:
    endpoint: jaeger:4317
    tls:
      insecure: true
  prometheus:
    endpoint: 0.0.0.0:8889

service:
  pipelines:
    traces:
      receivers: [otlp]
      processors: [memory_limiter, batch]
      exporters: [otlp]
    metrics:
      receivers: [otlp]
      processors: [batch]
      exporters: [prometheus]
```

1.21.3 Question 12

What are common Collector deployment patterns?

Show Solution

1. **No Collector (Direct)** - Apps export directly to backend
2. **Agent** - Collector as sidecar/daemonset per node
3. **Gateway** - Centralized Collector cluster
4. **Agent + Gateway** - Two-tier architecture

Best practices: - Use Agent for collection, Gateway for processing - Enable memory_limiter processor - Use batch processor for efficiency

---

1.22 Instrumentation (20%)

1.22.1 Question 13

What is the difference between auto and manual instrumentation?

Show Solution

Auto-instrumentation: - Automatic, no code changes - Uses agents or libraries - Covers common frameworks - Less control

Manual instrumentation: - Requires code changes - Full control over spans/metrics - Custom business logic - More effort

Best practice: Use auto-instrumentation as base, add manual for business logic.

1.22.2 Question 14

How do you enable auto-instrumentation in Python?

Show Solution

```
# Install
pip install opentelemetry-distro opentelemetry-exporter-otlp
opentelemetry-bootstrap -a install

# Run with auto-instrumentation
opentelemetry-instrument \
  --service_name my-service \
  --traces_exporter otlp \
  --metrics_exporter otlp \
  --exporter_otlp_endpoint http://localhost:4317 \
  python app.py
```

Or with environment variables:

```
export OTEL_SERVICE_NAME=my-service
export OTEL_EXPORTER_OTLP_ENDPOINT=http://localhost:4317
opentelemetry-instrument python app.py
```

1.22.3 Question 15

What semantic conventions should you follow?

Show Solution

Semantic conventions standardize attribute names:

HTTP: - `http.method` - GET, POST - `http.url` - Full URL - `http.status_code` - 200, 404

Database: - `db.system` - mysql, postgresql - `db.statement` - SQL query - `db.name` - Database name

Service: - `service.name` - Service identifier - `service.version` - Version string

Following conventions enables better correlation across tools.

1.23 Exam Tips

1. **Know the three signals** - Traces, Metrics, Logs
2. **Understand Collector architecture** - Receivers, Processors, Exporters

3. **Know SDK components** - Providers, Processors, Exporters
4. **Practice configuration** - Collector YAML, SDK setup
5. **Understand context propagation** - W3C Trace Context

← Back to OTCA Overview
