Mastering Observability

Building Resilient Personal Banking Systems

From Logs to Actionable Insights

Training Agenda

01 Why Observability Matters: The "Three Pillars" in Banking

02 Pillar 1: Centralized Logging - Unifying the Narrative

03 Logging Best Practices: Writing Logs That Tell a Story

04 Exercise #1: Basic Transaction Logging

05 Pillar 2: Metrics Instrumentation - Measuring What Matters

06 Tools: Micrometer, Prometheus, AWS X-Ray

07 Exercise #2: Tracking Login Attempts

08 Pillar 3: Health Checks & Alerting - Proactive Problem Solving

09 Exercise #3: Building a Health Check Endpoint

10 Conclusion & Key Takeaways

Why Observability in Banking?

Banking systems demand the highest level of reliability, security, and performance.
When a user can't transfer money or view their balance, we need to know *why*, *where*, and *how to fix it*—fast.

Observability isn't just about monitoring; it's about asking questions about your system you didn't know you needed to ask.

Logs

What happened?

Detailed, event-level records

Metrics

How is the system performing?

Aggregated, numerical data over time

Traces

Where did it happen?

End-to-end journey of a request

Pillar 1: Centralised Logging

What is it?

In a modern banking application (composed of microservices for accounts, transfers, authentication, etc.), logs are scattered everywhere. **Centralised logging** collects all these logs into a single, searchable location.

Why is it critical for Personal Banking?

- ✓ **Trace a Transaction:** Follow a user's fund transfer request from mobile app, through transfer service, to accounts service, and into the database.
- ✓ **Security Audits:** Quickly search for all login attempts for a specific user ID across all services.
- ✓ **Troubleshooting:** When a loan application fails, see the exact error message from the credit check service without SSH-ing into a server.

Popular Tools

🔍 **ELK Stack**
Elasticsearch, Logstash, Kibana

💼 **Splunk**
Enterprise-grade log analysis platform

🌟 **Graylog**
Open-source alternative

Logging Best Practices

Good logs are the foundation of good debugging.

1. Use Structured Logging (JSON)

Instead of plain text, log in a machine-readable format. This makes searching and filtering a breeze.

❌ Bad Example

```
User 123 failed to transfer 500 to 456
```

✅ Good Example

```
{"level": "ERROR", "timestamp": "...", "service": "transfers",
"userId": "123", "sourceAccount": "...", "targetAccount": "...",
"amount": 500, "message": "Insufficient funds"}
```

Logging Best Practices

2. Log with Context

Always include relevant IDs (userId, transactionId, accountId). A **correlationId** is a pro-level tip to trace one request across multiple services.

3. Use Appropriate Log Levels

- ✓ **INFO:** Standard operations (e.g., "User logged in," "Transaction initiated")
- ✓ **WARN:** Potential issues that don't break functionality
- ✓ **ERROR:** Failures that stop a process
- ✓ **DEBUG:** Verbose information for development only

⚠ **NEVER Log Sensitive Data:** NEVER log passwords, full credit card numbers, or personally identifiable information (PII) in plain text.

Exercise #1: Basic Transaction Logging in Python

What is Structured Logging?

Instead of logging plain text strings, structured logging uses a consistent format like JSON to record data.

❌ **Before (Unstructured)**

```
Transaction failed for user-987.
Reason: Exceeds limit.
```

✅ **After (Structured)**

```
{"level": "ERROR",
"userId": "user-987",
"reason": "Exceeds transfer limit"}
```

Why use it? 🤔

Structured logs are **machine-readable**, making them easy to search, filter, and analyse in modern log management systems (Splunk, Datadog, and Elasticsearch).

Key Python Modules

🔋 This exercise uses Python's powerful built-in **("batteries-included")** modules. No external installation needed!

logging	json	uuid
The standard library for logging events in Python	Format Python dictionaries into JSON strings	Generate unique IDs for each transaction

Benefits of Structured Logging

- **Searchability:** Quickly find specific transactions by ID, user, or amount
- **Analysis:** Aggregate data to identify patterns and trends
- **Integration:** Seamlessly connect with monitoring and alerting tools
- **Debugging:** Trace issues with complete context in one log entry

Python Implementation (Part 1)

```
import logging
import uuid
import json

# Basic logger configuration to print logs to the console
logging.basicConfig(level=logging.INFO, format='%(message)s')

def transfer_funds(user_id: str, from_account: str,
                   to_account: str, amount: float):
    """
    Simulates a fund transfer and logs the transaction details.
    """
    # 1. Generate a unique ID for this specific transaction
    transaction_id = str(uuid.uuid4())

    # 2. Create a dictionary to hold all structured log data
    log_data = {
        "transactionId": transaction_id,
        "userId": user_id,
        "fromAccount": from_account,
        "toAccount": to_account,
        "amount": amount,
    }
```

Key Points:

- Each transaction gets a **unique ID** using uuid module
- All relevant data is stored in a **dictionary** for structure
- Type hints make the code more readable and maintainable

Python Implementation (Part 2) & Output

```
    # 3. Log the start of the transaction
    logging.info(json.dumps({"message": "Transaction initiated",
                             **log_data}))

    # 4. Simulate a business logic check
    if amount > 10000:
        log_data["reason"] = "Exceeds transfer limit"
        logging.error(json.dumps({"message": "Transaction failed",
                                  **log_data}))
    else:
        log_data["status"] = "Completed"
        logging.info(json.dumps({"message": "Transaction successful",
                                 **log_data}))

# Example Run
transfer_funds("user-123", "ACCT-A-456", "ACCT-B-789", 5000.00)
transfer_funds("user-987", "ACCT-C-111", "ACCT-D-222", 15000.00)
```

Expected Output

```
--- Successful Transaction ---
{"message": "Transaction initiated", "transactionId": "abc-123", ...}
{"message": "Transaction successful", "status": "Completed", ...}

--- Failed Transaction ---
{"message": "Transaction initiated", "transactionId": "def-456", ...}
{"message": "Transaction failed", "reason": "Exceeds transfer limit", ...}
```

Pillar 2: Metrics Instrumentation

What are Metrics?

Metrics are numerical representations of data measured over time. While logs tell you *what happened*, metrics tell you *how much* and *how often*.

Counter

Value that only goes up
(e.g., logins.total, transactions.failed)

Gauge

Value that can go up or down
(e.g., cpu.usage, active.users)

Timer/Histogram

Measures event duration
(e.g., api.response.time)

Why is it Critical for Personal Banking?

- ✓ **Performance:** "What is the average API response time for the 'get balance' endpoint?"
- ✓ **Business KPIs:** "How many fund transfers are happening per minute?"
- ✓ **Resource Usage:** "Is the database CPU usage approaching its limit during peak hours?"

Python Metrics & Monitoring Stack

Prometheus Client

Official Python client library for instrumenting Python applications with Prometheus metrics.

- Counter, Gauge, Histogram, Summary metrics
- Built-in HTTP server for metrics export
- Thread-safe and process-safe

Benefit: Native Prometheus integration

StatsD / DogStatsD

Lightweight metrics aggregation daemon with Python client support for sending custom metrics.

- Simple UDP-based protocol
- Low overhead, fire-and-forget
- Works with Datadog, Graphite

Benefit: Minimal performance impact

OpenTelemetry

Unified observability framework providing metrics, traces, and logs for Python applications.

- Vendor-neutral instrumentation
- Auto-instrumentation for frameworks
- Export to multiple backends

Benefit: Complete observability solution

Recommended Python Stack: Use **Prometheus Client** for application metrics instrumentation, **Prometheus** for storage/querying with PromQL, and **OpenTelemetry** for distributed tracing and unified observability across your Python microservices.

Exercise #2: Building an Observable API

Combine structured logging and Prometheus metrics in a real FastAPI web application to create a truly observable service 🚀

Goal: Learn to implement comprehensive observability in modern Python web services

This exercise builds on previous logging concepts and introduces metrics for production-ready applications.

Logging vs. Metrics: What's the Difference?

In our previous exercises, we used logging. This new code adds metrics. It's important to understand how they work together.

🪵 Structured Logging (The Diary)

What it is: Records specific, individual events with rich context (like a transactionId or userId).

Use Case: Answering "What happened on this specific request?" Perfect for debugging a single failure.

📊 Metrics (The Dashboard)

What it is: Aggregated, numerical data over time (e.g., a count of login_attempts). Metrics are cheap to store and fast to query.

Use Case: Answering "How many logins failed in the last hour?" Perfect for dashboards, alerting, and seeing trends.

Understanding the Analogy

Logging is a detailed diary of your car trip, writing down every turn and stop.

Metrics are the car's dashboard, showing your current speed, fuel level, and engine temperature.

You need both to understand the full story! ✨

Key Tools in This Example

FastAPI: A modern, high-performance Python framework for building APIs. It's the foundation of our web service.

Prometheus: A powerful open-source tool for collecting and storing metrics. The prometheus-client library lets our Python app create and expose these metrics.

> These tools work together to provide complete observability, allowing you to monitor, debug, and optimize your applications effectively.

Python Code: Tracking Logins with Metrics

Let's look at the /login endpoint. Notice how it increments a Prometheus Counter.

```python
#
# Part 1: Setting up the metric
#
from prometheus_client import Counter, CollectorRegistry

# A registry to hold all our metrics
registry = CollectorRegistry()

# Define a Counter metric.
# It has a name, a description, and a "label" called 'status'.
LOGIN_ATTEMPTS = Counter(
    "login_attempts_total",
    "Total number of login attempts",
    ["status"],  # This label lets us tag logins as "success" or "failure"
    registry=registry
)
```

```python
#
# Part 2: Using the metric in our FastAPI app
#
@app.post("/login")
async def login(req: LoginRequest):
    if req.username == "admin" and req.password == "password":
        # If login is correct, increment the counter with the 'success' label
        LOGIN_ATTEMPTS.labels(status="success").inc()
        return {"message": "Login successful"}
    else:
        # Otherwise, increment the counter with the 'failure' label
        LOGIN_ATTEMPTS.labels(status="failure").inc()
        return JSONResponse({"message": "Login failed"}, status_code=401)
```

How Do We See the Metrics?

The code creates a special /metrics endpoint. When you visit this endpoint, Prometheus provides the raw metrics in a simple text format.

Example output from the /metrics endpoint:

```
# HELP login_attempts_total Total number of login attempts # TYPE
login_attempts_total counter login_attempts_total{status="success"} 1.0
login_attempts_total{status="failure"} 1.0
```

A Prometheus server scrapes this endpoint periodically, and you can then use a tool like **Grafana** to create dashboards and alerts from this data, turning simple numbers into powerful insights. ✨

Pillar 3: Health Checks & Alerting

What are Health Checks?

A simple endpoint (e.g., /health) that your application exposes to report its current status. Orchestration tools (like Kubernetes) use this endpoint to know if your application is running correctly.

Basic Health Check Response

```
{"status": "UP"}
```

Advanced Health Check (with Dependencies)

```
{
  "status": "UP",
  "components": {
    "database": {"status": "UP"},
    "paymentGateway": {"status": "UP"},
    "diskSpace": {"status": "UP", "details": {"free": "85%"}}
  }
}
```

Pillar 3: Health Checks & Alerting

What is Alerting?

Automatically notifying the team when a problem occurs. Alerts are triggered by rules defined on your metrics or logs.

- ✓ **Metric Alert:** IF (average(api.response.time) > 2s for 5 minutes) THEN page on-call engineer
- ✓ **Security Alert:** IF (sum(login.attempts{status="failure"}) > 100 in 1 minute) THEN trigger security alert
- ✓ **Business Alert:** IF (count(ERROR logs with "Insufficient funds") > 50 in 10 minutes) THEN notify business team

⚠️ **Pro Tip:** Your alerts must be *actionable*. If an alert fires and the team doesn't know what to do, it's just noise. Every alert should be linked to a runbook.

Exercise #3: Python Health Check Endpoint

Goal: Create a FastAPI endpoint that checks database connection and reports application health.

```
@app.get("/health", response_model=HealthResponse)
async def health_check():
    db_healthy, db_message = await check_database_connection()

    if db_healthy:
        return HealthResponse(
            status="UP",
            components={
                "database": ComponentHealth(
                    status="UP",
                    details={"message": db_message}
                )
            }
        )
    else:
        raise HTTPException(
            status_code=status.HTTP_503_SERVICE_UNAVAILABLE,
            detail=HealthResponse(
                status="DOWN",
                components={
                    "database": ComponentHealth(
                        status="DOWN",
                        error=db_message
                    )
                }
            ).dict()
        )
```

Exercise #3:
The Application Health Check

Goal: Create a FastAPI endpoint that checks our application's health and reports its status

What is a Health Check?

- It's like a doctor checking a patient's vital signs 🩺
- It's a special URL (e.g., /health) that other services can call
- It answers one simple question: "*Are you okay to receive work?*"
- This is fundamental for building reliable and stable applications

Why is it So Important?

Health checks enable automated systems to manage our application effectively:

- **Monitoring & Alerting:** Automatically detect if the application goes down
- **Load Balancing:** A load balancer will stop sending traffic to an unhealthy server
- **Auto-Healing:** Systems like Kubernetes can automatically restart an unhealthy application instance

Code Breakdown - The Endpoint

```
@app.get("/health", response_model=HealthResponse)
async def health_check():
    # ... logic goes here ...
```

- @app.get("/health"): Creates a new API endpoint accessible via a GET request at the /health URL path
- async def health_check(): The function that executes whenever the /health endpoint is called

Code Breakdown - The Core Logic

```
db_healthy, db_message = await check_database_connection()

if db_healthy:
    # Return a "Healthy" response
else:
    # Return an "Unhealthy" response
```

- An application isn't healthy if its critical dependencies aren't working
- Here, we check the most common dependency: **the database**
- The result of this check determines our application's status

The "Healthy" Response

200 OK

When everything is working correctly:

```
# if db_healthy:
return HealthResponse(
    status="UP",
    components={
        "database": ComponentHealth(
            status="UP",
            details={"message": db_message}
        )
    }
)
```

Meaning: "I'm alive and ready for work!"

The "Unhealthy" Response

503 Service Unavailable

When the database connection fails:

```
# else:
raise HTTPException(
    status_code=status.HTTP_503_SERVICE_UNAVAILABLE,
    detail=HealthResponse(...)
)
```

This is crucial! The 503 status code is a standard signal that tells other systems:

"I'm temporarily broken. Do NOT send me any user traffic!"

Questions & Answers

Q1: What is the main purpose of a health check endpoint?

To provide a simple, automated way for other systems to know if an application is running correctly and is ready to handle requests.

Q2: Why check the database connection?

Because the database is a critical dependency, if the application can't connect to its database, it can't perform its main functions.

Q3: What HTTP status code for unhealthy apps, and why?

503 Service Unavailable. This specific code tells automated systems like load balancers to stop sending traffic immediately.

Q4: Load balancer scenario: 3 servers, Server #2 fails health check. What happens?

The load balancer will detect the unhealthy response from Server #2 and stop sending traffic to it. It will distribute traffic between Server #1 and #3.

Questions & Answers (cont.)

Q5: Besides a database, what else might you check?

You might check:

- Connection to another required microservice
- An external API (like a payment gateway)
- A message queue system (like RabbitMQ or Kafka)

Conclusion

Observability is not a tool, it's a culture.

It's about instrumenting your code to provide the insights you need to build and maintain a world-class Personal Banking System.

Centralized Logging

Gives you the power to trace any action

Metrics

10,000-foot view of system performance and business KPIs

Health Checks & Alerting

Turn you from reactive firefighter into proactive problem-solver

By mastering these pillars, you build trust with your users, protect their data, and ensure the system is always there when they need it.

Key Takeaways

- ✓ **Instrument Everything:** If it's important, it should be logged or measured.
- ✓ **Log in JSON:** Your future self (at 3 AM) will thank you.
- ✓ **Use Correlation IDs:** They are the single most powerful tool for debugging microservices.
- ✓ **Metrics Tell a Story:** Use dashboards to visualize login rates, transfer volumes, and API performance.
- ✓ **Alerts Must Be Actionable:** If an alert isn't important enough to wake someone up, it's noise. Tune your alerts aggressively.
- ✓ **Start Simple:** You don't need a complex system from day one. Start with good logging practices and build from there.

Thank You!

Q & A?