Calculator Microservice

Overview

This microservice provides a RESTful API for basic and advanced arithmetic operations. The service is built using **Node.js** and **Express**, with **Winston** for logging requests and errors.

Features

- Basic arithmetic operations: **Addition, Subtraction, Multiplication, Division**
- Advanced operations: **Exponentiation, Square Root, Modulo**
- Request logging using Winston (console and file logging)
- Error handling for invalid inputs and division by zero

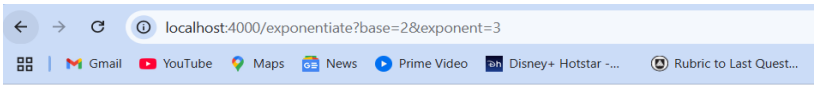
Exponentiation

- **Endpoint:** /exponentiate

```
// Exponentiation Route
app.get("/exponentiate", (req, res) => {
    const base = parseFloat(req.query.base);
    const exponent = parseFloat(req.query.exponent);
    if (isNaN(base) || isNaN(exponent)) {
        logger.error("Invalid numbers provided for exponentiation");
        return res.status(400).send("Invalid numbers provided.");
    }
    const result = Math.pow(base, exponent);
    logger.info(`Exponentiation Result: ${result}`);
    res.send(`Result: ${result}`);
});

// Square Root Route
app.get("/sqrt", (req, res) => {
    const num = parseFloat(req.query.num);
    if (isNaN(num)) {
        logger.error("Invalid number provided for square root");
        return res.status(400).send("Invalid number provided.");
    }
    if (num < 0) {
        logger.error("Attempted to calculate square root of a negative number");
        return res.status(400).send("Cannot calculate square root of a negative number.");
    }
    const result = Math.sqrt(num);
    logger.info(`Square Root Result: ${result}`);
    res.send(`Result: ${result}`);
});
```

- **Example:** http://localhost:4000/exponentiate?base=2&exponent=3

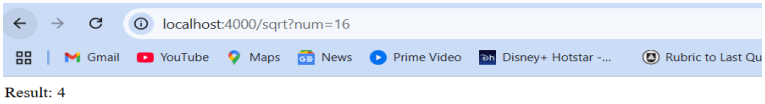


Result: 8

- **Response:** Result: 8

Square Root

- **Endpoint:** /sqrt
- **Example:** http://localhost:4000/sqrt?num=16



- **Response:** Result: 4

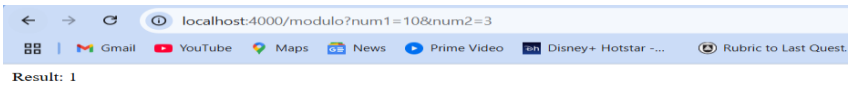
Modulo

- **Endpoint:** /modulo

```
    res.send(`Result: ${result}`);
});

// Modulo Route
app.get("/modulo", (req, res) => {
    const num1 = parseFloat(req.query.num1);
    const num2 = parseFloat(req.query.num2);
    if (isNaN(num1) || isNaN(num2)) {
        logger.error("Invalid numbers provided for modulo operation");
        return res.status(400).send("Invalid numbers provided.");
    }
    const result = num1 % num2;
    logger.info(`Modulo Result: ${result}`);
    res.send(`Result: ${result}`);
});
```

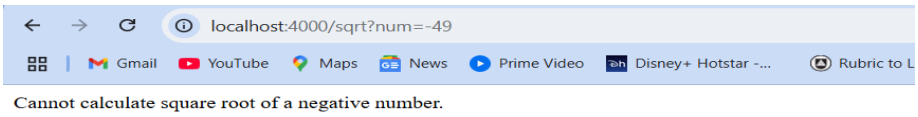
- **Example:** http://localhost:4000/modulo?num1=10&num2=3

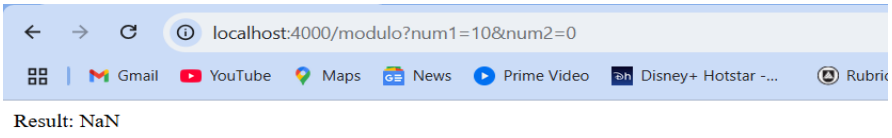


- **Response:** Result: 1

Error Handling

- Invalid inputs return a **400 Bad Request** error with an appropriate message.
- Division by zero and negative square roots are prevented.





Logging

- All requests are logged in server.log
- Errors are logged separately

Error Handling Strategies in Microservices Architecture

The distributed nature of microservices makes error handling essential, even though the architecture is built for scalability and robustness. System overload, network problems, and unavailable services can all lead to failures. Efficient error-handling techniques increase system dependability and improve user satisfaction. The Circuit Breaker pattern, Retry pattern, and Fallback mechanism are some of the important error handling techniques that are examined in this paper along with their implications for microservices-based applications.

1. Circuit Breaker Pattern

The **Circuit Breaker** pattern prevents a service from making repeated requests to a failing dependency. It works in three states:

- **Closed:** Requests are forwarded normally.
- **Open:** Requests are blocked if failures exceed a threshold.
- **Half-Open:** Limited requests are tested before resuming normal operation.

Implications:

- Improves system resilience by preventing cascading failures.
- Reduces response time by avoiding calls to a non-responsive service.
- Enhances user experience by providing faster failure detection.

Example Implementation:

Netflix's **Hystrix** was a popular Circuit Breaker library, but its successor, **Resilience4j**, is now widely used.

2. Retry Pattern

The **Retry pattern** attempts failed requests again after a brief delay, often with **exponential backoff** to reduce strain on the system.

Implications:

- Helps handle transient failures, such as temporary network glitches.
- Can degrade performance if not configured properly (e.g., too many retries increasing load).

Example Implementation:

Spring Retry and Resilience4j's retry module are commonly used in Java-based microservices.

3. Fallback Mechanism

A **Fallback mechanism** provides an alternative response when a service fails. This could be a cached response, a default value, or redirecting requests to a backup service.

Implications:

- Enhances user experience by avoiding complete service disruption.
- Useful for critical applications where some response is better than none.
- Increases complexity as it requires defining meaningful fallback strategies.

Example Implementation:

Fallbacks are often implemented using Resilience4j or service mesh solutions like Istio.

Conclusion

The dependability of microservices is greatly increased by error handling techniques like circuit breakers, retries, and fallback mechanisms. A balanced mix of these strategies guarantees that services continue to be robust, and that user experience is not adversely affected. Careful implementation is crucial because incorrect settings can result in higher latency or needless retries.

References

1. N. Joshi, *"Microservices Patterns: Circuit Breaker, Retry, and Fallback Mechanisms,"* 2023.
2. Netflix Tech Blog, *"Hystrix – Latency and Fault Tolerance for Distributed Systems,"* 2018.
3. Resilience4j Documentation, *"Fault Tolerance Library for Java Applications,"* 2024.