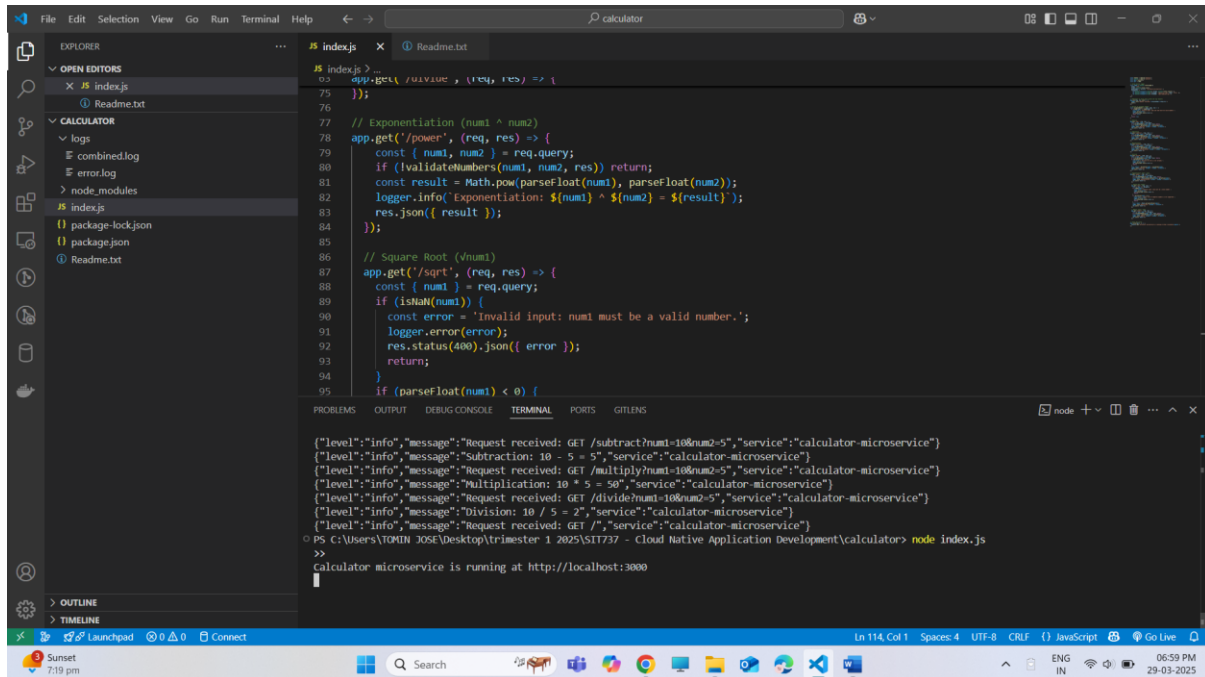


Git Repo: <https://github.com/TOMINJOSE88/sit737-2025-prac4c.git>

Part I Additional Arithmetic Operations

Running the code:



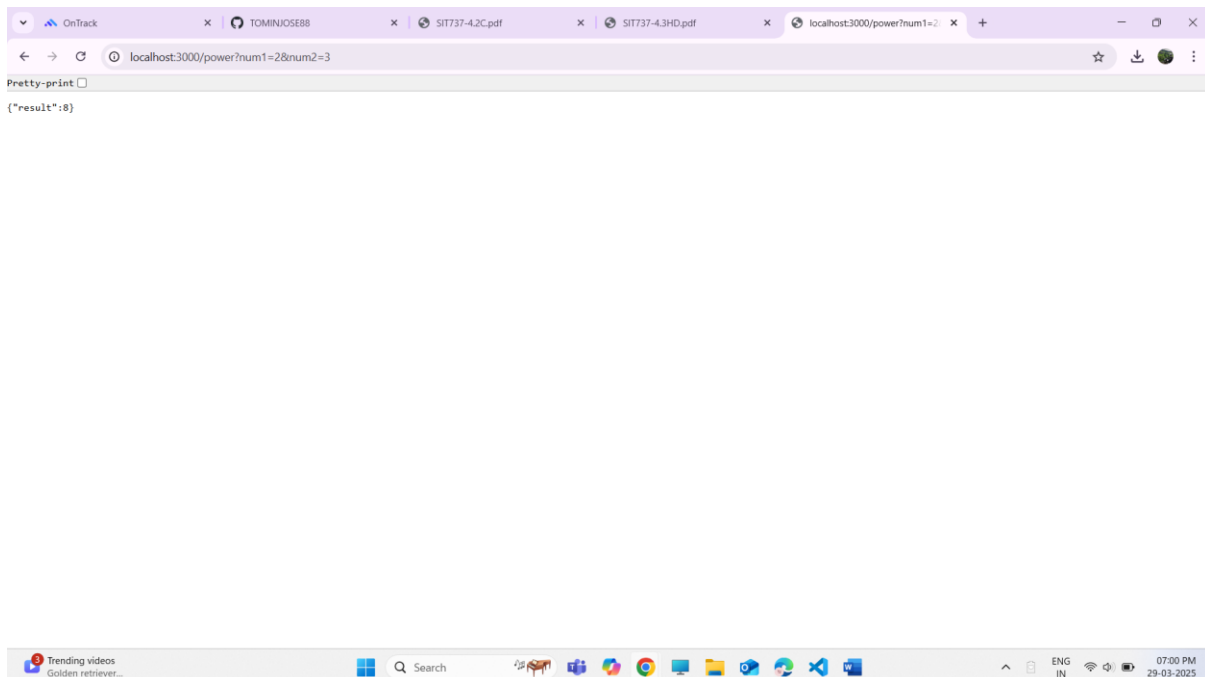
The screenshot shows the VS Code editor with the `calculator` project open. The `index.js` file is open in the editor, showing the following code:

```
1 // Exponentiation (num1 ^ num2)
2 app.get('/power', (req, res) => {
3   const { num1, num2 } = req.query;
4   if (!validateNumbers(num1, num2, res)) return;
5   const result = Math.pow(parseFloat(num1), parseFloat(num2));
6   logger.info('Exponentiation: ${num1} ^ ${num2} = ${result}');
7   res.json({ result });
8 });
9
10 // Square Root (√num1)
11 app.get('/sqrt', (req, res) => {
12   const { num1 } = req.query;
13   if (!isNan(num1)) {
14     const error = 'Invalid input: num1 must be a valid number.';
15     logger.error(error);
16     res.status(400).json({ error });
17     return;
18   }
19   if (parseFloat(num1) < 0) {
20     const error = 'Invalid input: num1 must be a non-negative number.';
21     logger.error(error);
22     res.status(400).json({ error });
23     return;
24   }
25   const result = Math.sqrt(parseFloat(num1));
26   logger.info('Square Root: √${num1} = ${result}');
27   res.json({ result });
28 });
```

The terminal output shows the following logs:

```
[{"level":"info","message":"Request received: GET /subtract?num1=10&num2=5","service":"calculator-microservice"}]
[{"level":"info","message":"Subtraction: 10 - 5 = 5","service":"calculator-microservice"}]
[{"level":"info","message":"Request received: GET /multiply?num1=10&num2=5","service":"calculator-microservice"}]
[{"level":"info","message":"Multiplication: 10 * 5 = 50","service":"calculator-microservice"}]
[{"level":"info","message":"Request received: GET /divide?num1=10&num2=5","service":"calculator-microservice"}]
[{"level":"info","message":"Division: 10 / 5 = 2","service":"calculator-microservice"}]
[{"level":"info","message":"Request received: GET /","service":"calculator-microservice"}]
PS C:\Users\TOMIN JOSE\Desktop\trimester 1 2025\SIT737 - Cloud Native Application Development\calculator> node index.js
>>
Calculator microservice is running at http://localhost:3000
```

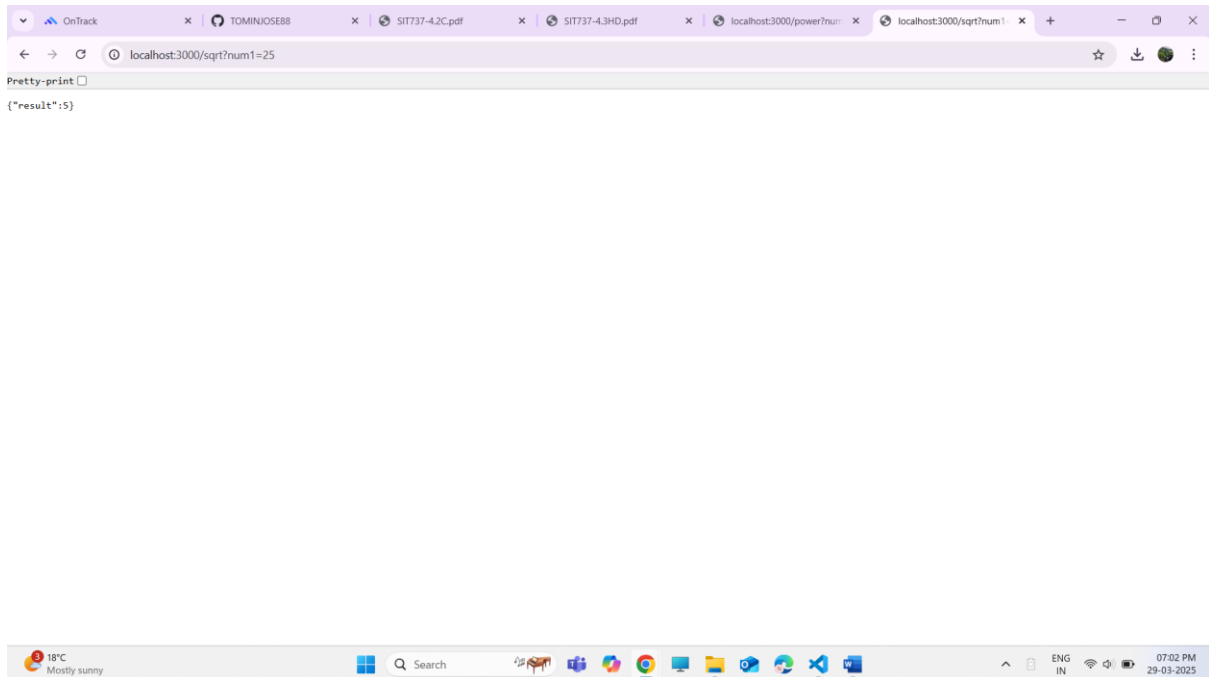
Exponentiation:



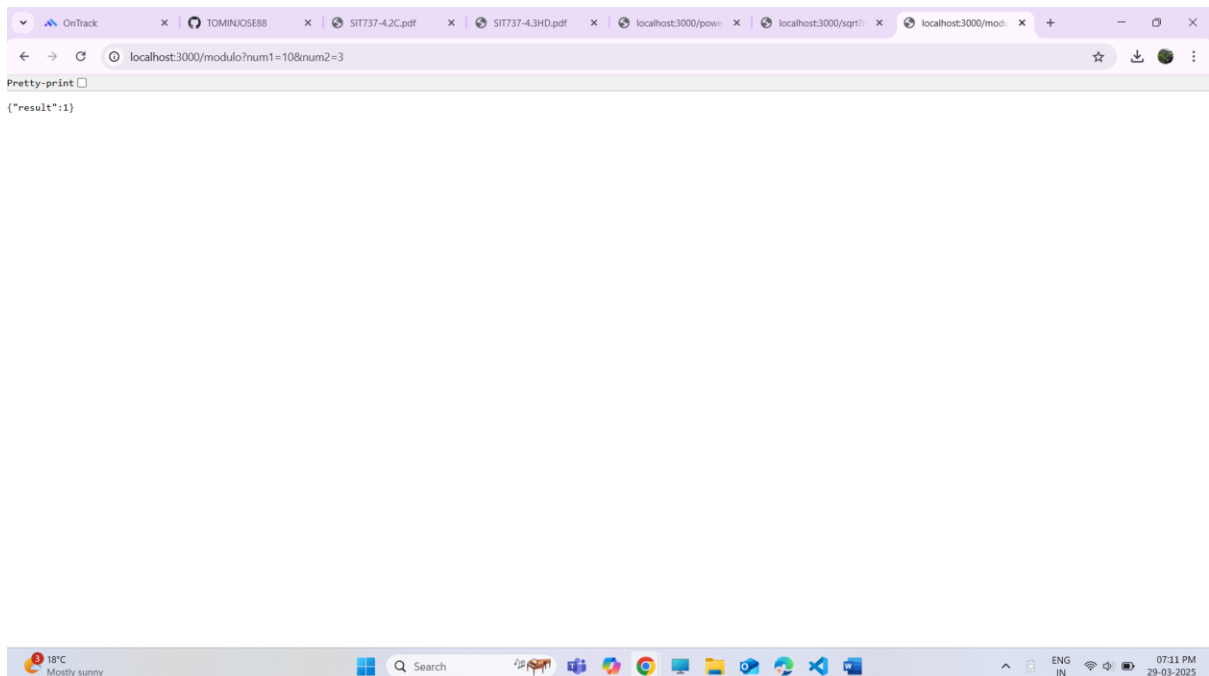
The screenshot shows a web browser with the address bar displaying `localhost:3000/power?num1=2&num2=3`. The page content shows the result of the exponentiation:

```
{"result":8}
```

Square Root:



Modulo:



Part 2

Error Handling Strategies in Microservices Architecture

In a microservices-based environment, individual services interact over networks, often relying on one another for functionality. This distributed nature increases the chances of failure due to network issues, timeouts, or service unavailability. To build resilient systems, it is essential to adopt effective error-handling strategies that enhance reliability without compromising user experience. Common strategies include the Circuit Breaker pattern, Retry pattern, and Fallback mechanisms.

Circuit Breaker Pattern:

This pattern prevents a service from repeatedly invoking a failing operation. It monitors calls to external services and “breaks” the circuit (stops further attempts) if failures cross a threshold. The circuit then enters a cooldown period before testing if the service has recovered. This prevents system overload, conserves resources, and improves responsiveness. Tools like Netflix Hystrix and Resilience4j implement this pattern effectively.

Retry Pattern:

The Retry strategy automatically reattempts a failed request a certain number of times with delays in between. It is useful for handling transient failures such as temporary network glitches or brief service downtime. However, without limits or delays (e.g., exponential backoff), retrying can worsen system performance and strain the failing service.

Fallback Mechanism:

Fallback provides a default response or alternate workflow when a service fails or is unreachable. This ensures continuity in user experience by gracefully handling errors. For example, if a recommendation service is unavailable, a fallback can serve cached or default suggestions instead of showing an error.

Implications on Reliability and User Experience:

Adopting these patterns increases fault tolerance and ensures smooth recovery from partial failures, which is critical in modern microservices. Circuit Breakers protect the system from cascading failures. Retry patterns improve reliability in the face of transient issues, while fallback mechanisms enhance user satisfaction by providing a seamless experience even during outages. However, incorrect usage—like excessive retries or over-reliance on fallbacks—can lead to performance degradation or hidden failures.

Conclusion:

Using robust error-handling strategies is vital for maintaining a resilient microservices architecture. Circuit Breakers, Retry, and Fallbacks, when correctly implemented,

provide a balanced approach to fault tolerance, improving overall service stability and user trust.

References:

1. Microsoft, 2023. *Retry pattern - Cloud Design Patterns*. [online] Microsoft Learn. Available at: <https://learn.microsoft.com/en-us/azure/architecture/patterns/retry> [Accessed 29 Mar. 2025].
2. Microsoft, 2023. *Circuit Breaker pattern - Cloud Design Patterns*. [online] Microsoft Learn. Available at: <https://learn.microsoft.com/en-us/azure/architecture/patterns/circuit-breaker> [Accessed 29 Mar. 2025].
3. Resilience4j, 2024. *Resilience4j User Guide*. [online] Available at: <https://resilience4j.readme.io/docs/getting-started-3> [Accessed 29 Mar. 2025].