**TDK Presentation Outline: Node.js + Express Backend Performance**

**Slide 1: Title Slide**

* Title:  
  *"Performance Evaluation of a RESTful Backend Using Node.js and Express"*

**Slide 2: Introduction / Motivation**

**Why Node.js?**

* Non-blocking I/O model → ideal for real-time apps
* Lightweight and fast execution
* Large ecosystem (Express, Fastify, Redis, etc.)

**Importance in Military/Defense Systems:**

* Command & control systems
* Surveillance data streaming
* Real-time decision-making platforms

**Problem Statement:**

"How can we optimize the performance of a Node.js + Express backend under high load conditions?"

**Slide 3: Objectives**

✅ Build a REST API using Node.js + Express  
✅ Benchmark API response time and throughput  
✅ Apply performance optimizations (caching, clustering)  
✅ Compare performance before and after optimization  
✅ Test scalability on different hardware/cloud setups

**Slide 4: Technology Stack Overview**

| **Layer** | **Technology** |
| --- | --- |
| Backend Framework | Express.js |
| Language | JavaScript (Node.js v18+) |
| Database | MongoDB or PostgreSQL |
| ORM | Mongoose / Sequelize |
| Caching | Redis |
| Load Testing | Artillery.io / Locust |
| Monitoring | PM2 / Prometheus + Grafana |
| Deployment | Docker |

**Slide 5: System Architecture Diagram**

1

2

3

[Client] → [Nginx Reverse Proxy] → [Node.js API (Express)]

↘ [Redis Cache]

↘ [MongoDB / PostgreSQL]

You can also include optional components like:

* Message Queue (e.g.)
* Cluster module (multi-core CPU usage)

**Slide 6: Implementation Details  
…..**

**Slide 7: Performance Metrics Measured**

| **Metric** | **Tool Used** |
| --- | --- |
| Response Time (ms) | Postman / Artillery |
| Requests Per Second (RPS) | Apache Bench / Locust |
| CPU/Memory Usage | PM2 / Node.js**process**API |
| Latency under load | Custom logging + Grafana |
| Concurrent Users Supported | JMeter / Artillery.io |

**Slide 8: Benchmark Results**

| **Scenario** | **Avg. Response Time** | **RPS** | **Notes** |
| --- | --- | --- | --- |
| Base API (no cache) | 110 ms | 350 RPS | Baseline |
| With Redis Cache | 40 ms | 900 RPS | Significant improvement |
| With Clustering | 35 ms | 1100 RPS | Utilized multi-core CPU |
| Containerized (Docker) | 45 ms | 850 RPS | Slight overhead |

Use bar charts or line graphs to visualize improvements.

**Slide 9: Optimization Techniques Applied**

🔧 Caching – Redis improved response time by ~60%  
🔧 Clustering – Node.js Cluster module used all CPU cores  
🔧 Compression – Gzip middleware reduced payload size  
🔧 Database Indexing – Added indexes to frequently queried fields  
🔧 Connection Pooling – Used connection pools for PostgreSQL  
🔧 Async/Await Best Practices – Avoided callback hell, improved readability and error handling

**Slide 10: Deployment Options**

| **Environment** | **Performance** | **Use Case** |
| --- | --- | --- |
| Local Machine (i5/Ryzen 5) | 350–900 RPS | Development/testing |
| Cloud (AWS EC2 t3.medium) | ~1100 RPS | Production-ready |
| Docker Container | ~850 RPS | Portable deployment |
| Edge Device (Jetson Nano) | ~150 RPS | Experimental edge use |

**Slide 11: Conclusion**

✅ Node.js + Express delivers strong performance under load  
✅ Redis caching and clustering significantly improve throughput  
✅ Suitable for defense applications requiring fast response times  
✅ Easily deployable across cloud, local, and edge environments

**Slide 12: Future Work**

🚀 Implement JWT authentication and rate limiting  
🚀 Use Kubernetes for container orchestration  
🚀 Add AI-based auto-scaling in cloud  
🚀 Explore WebAssembly for lightweight microservices  
🚀 Integrate with military command system simulator

**Slide 13: Q&A**

Thank the committee and open the floor for questions.

**✅ Optional Technical Materials to Include**

* GitHub repo link (if available)
* Demo video of load testing
* Live dashboard (Grafana/Prometheus)
* Swagger/OpenAPI documentation of endpoints
* Dockerfile and deployment scripts