.NET Mentoring Program Intermediate+ Practice_Q2_2024

TROUBLESHOOTING & LOGGING

What are the differences between performance, load, and stress testing?

Aspect	Performance Testing	Load Testing	Stress Testing
Definition	Evaluates the speed, responsiveness, and stability of a system under a given workload.	Measures the system's behavior under expected user loads to determine if it can handle the expected number of users.	Determines the system's stability and robustness by testing beyond normal operational capacity, often to the point of breaking.
Purpose	To ensure the system meets performance criteria (e.g., response time, throughput).	To verify the system's ability to handle expected peak loads.	To identify the system's breaking point and its behavior under extreme conditions.
Key Metrics	Response time, throughput, resource utilization (CPU, memory, disk I/O, etc.).	Response time, throughput, error rate under varying load conditions.	Response time, system crashes, data corruption, and recovery time under extreme load conditions.
Scenario	Simulating different user interactions and measuring performance criteria.	Gradually increasing the number of users and measuring system behavior until the expected peak load is reached.	Continuously increasing the load until the system fails or shows significant performance degradation.
Typical Questions Answered	- How fast does the system respond? - How many users can it handle efficiently?	- Can the system handle the expected number of concurrent users? - How does it behave under peak load?	- What is the system's breaking point? - How does it recover after failure?
When Conducted	Throughout the development lifecycle, particularly before release.	Before the system goes live to ensure it can handle expected traffic.	Before the system goes live to ensure it can handle unexpected extreme conditions.
Tools Used	JMeter, LoadRunner, NeoLoad, AppDynamics, New Relic.	JMeter, LoadRunner, NeoLoad, BlazeMeter.	JMeter, LoadRunner, NeoLoad, Chaos Monkey.
Benefits	Ensures the system meets performance standards, improves user satisfaction, and identifies performance bottlenecks.	Validates the system’s ability to handle peak usage, ensures reliability, and helps plan capacity.	Helps in understanding system limits, ensures robustness, and prepares for extreme conditions.

When would you prefer vertical scaling over horizontal?

Vertical scaling (scaling up) and horizontal scaling (scaling out) are two approaches to enhancing the capacity and performance of a system. Here are some scenarios where vertical scaling might be preferred over horizontal scaling:
Scenarios for Preferring Vertical Scaling

- Simplicity of Implementation
  - Scenario: You need a quick and straightforward way to improve performance.
  - Reason: Vertical scaling involves adding more resources (CPU, RAM, etc.) to an existing server, which

is often simpler and faster to implement compared to horizontal scaling, which might require changes to the application architecture and more complex configurations.

- Cost Considerations
  - Scenario: The cost of adding resources to an existing machine is lower than adding more machines.
  - Reason: In some cases, it might be more cost-effective to upgrade the existing server hardware rather than purchasing and maintaining additional servers.
- Application Design Constraints
  - Scenario: The application is not designed to run in a distributed manner.
  - Reason: Some legacy applications or systems may not support horizontal scaling well. Vertical scaling can be a more feasible option when the application is monolithic or tightly coupled.
- Licensing and Software Constraints
  - Scenario: Software licensing costs increase with the number of instances.
  - Reason: Some software licenses are based on the number of server instances rather than the resources of a single server. In such cases, vertical scaling can be more economical.
- Consistent Performance Needs
  - Scenario: You require consistent and predictable performance improvements.
  - Reason: Vertical scaling often results in more predictable performance improvements since you are enhancing the capacity of a single machine, avoiding the potential complexities and variability introduced by distributed systems.
- Data Consistency and Integrity
  - Scenario: Maintaining data consistency and integrity is critical, and the system is not designed for distributed transactions.
  - Reason: Vertical scaling keeps all data operations within a single system, simplifying transaction management and consistency.
- Infrastructure Constraints
  - Scenario: Physical or logistical constraints limit the ability to add more servers.
  - Reason: In environments with limited space or where adding new servers is impractical, upgrading existing machines can be a more viable option.
- Initial Phase of Development
  - Scenario: The application is in the initial development phase and doesn't yet require the scalability that a distributed system provides.
  - Reason: Vertical scaling can meet the immediate needs and is often simpler during the early stages of development, allowing the focus to remain on building core functionality.
- Examples
  - Small to Medium-Sized Businesses: A small online retailer experiencing increased traffic during a sale might choose to vertically scale their database server to handle the surge, as it's a quick way to improve performance without complex changes.
  - Legacy Systems: An enterprise running a legacy ERP system that is not designed for distributed environments may opt to upgrade the existing server's hardware to handle more transactions rather than attempting a costly and risky re-architecture for horizontal scaling.
  - Single-Instance Applications: Applications that are inherently single-instance, such as certain analytics or batch processing applications, can benefit more from vertical scaling to speed up processing times.

Does ASP.NET Core API support horizontal scaling? Explain your answer

Yes, ASP.NET Core API supports horizontal scaling. Here's an explanation of how and why it supports horizontal scaling:

Horizontal Scaling in ASP.NET Core API

Horizontal scaling, also known as scaling out, involves adding more instances of an application to distribute the load.

ASP.NET Core is designed with features that facilitate horizontal scaling, making it a robust choice for building scalable web applications and APIs.

Key Features Supporting Horizontal Scaling

- Statelessness:
  - Explanation: ASP.NET Core encourages the development of stateless applications, where each request is independent and does not rely on server-specific data. Statelessness allows multiple instances of the application to handle requests independently.
  - Benefit: You can add or remove instances without affecting the ongoing requests, making it easy to scale out.
- Distributed Caching:
  - Explanation: ASP.NET Core supports distributed caching mechanisms (e.g., Redis, SQL Server) which allow caching data across multiple instances.
  - Benefit: Caching can be shared among instances, reducing the load on the database and improving performance.
- Session State Providers:
  - Explanation: For scenarios where session state is required, ASP.NET Core provides distributed session state providers that store session data in a centralized store (e.g., SQL Server, Redis).
  - Benefit: Session state can be maintained across multiple instances, enabling horizontal scaling.
- Load Balancing:
  - Explanation: ASP.NET Core applications can be deployed behind load balancers that distribute incoming HTTP requests across multiple instances.
  - Benefit: Load balancers ensure even distribution of traffic, improving the application's ability to handle high loads.
- Microservices Architecture:
  - Explanation: ASP.NET Core supports the development of microservices, where the application is divided into smaller, independently deployable services.
  - Benefit: Each microservice can be scaled independently based on its load, providing fine-grained control over scaling.
- Cloud-Native Features:
  - Explanation: ASP.NET Core is well-suited for cloud environments (e.g., Azure, AWS), which provide native support for horizontal scaling through features like auto-scaling groups, container orchestration (e.g., Kubernetes), and serverless computing (e.g., Azure Functions).
  - Benefit: Cloud platforms can automatically scale the number of instances based on demand, providing elasticity.

Real-World Example

Imagine you have an ASP.NET Core API that serves an e-commerce platform. During peak shopping seasons, the number of incoming requests can significantly increase. Here's how you can horizontally scale your ASP.NET Core API:

Deploy Multiple Instances: Deploy multiple instances of your ASP.NET Core API application on multiple servers or virtual machines.

Load Balancer: Use a load balancer (e.g., Azure Load Balancer, AWS Elastic Load Balancing) to distribute incoming requests evenly across these instances.

Distributed Cache: Implement a distributed cache using Redis to store frequently accessed data, reducing database load and ensuring cache consistency across instances.

Auto-Scaling: Configure auto-scaling rules in your cloud environment to automatically add or remove instances based on metrics such as CPU usage, memory usage, or request count.