Módulo 3 – Tipos de Testes de Performance

Neste módulo, vamos entender os **principais tipos de testes de performance** e **quando aplicá-los na prática**.

Esses testes são fundamentais para validar a **estabilidade, escalabilidade e eficiência** de uma aplicação antes que ela chegue em produção.

---

1. Visão Geral

Os testes de performance simulam cenários reais de uso, permitindo detectar:

- Gargalos em endpoints e APIs.
- Lentidão sob alta demanda.
- Problemas de infraestrutura (CPU, memória, rede, banco de dados).
- Impactos de novas versões antes da produção.

Na prática, uma boa estratégia de performance combina diferentes tipos de testes, cada um com um propósito específico.

---

2. Teste de Fumaça (Smoke Test)

Objetivo: Verificar rapidamente se o ambiente e os endpoints principais estão respondendo corretamente.

Aplicação prática:

É utilizado antes de testes mais longos, para confirmar que o sistema está ativo e que as respostas básicas estão corretas (por exemplo, status 200).

Exemplo de cenário:

Antes de rodar um teste de carga de 30 minutos, executa-se um smoke test de 5 segundos com apenas 1 usuário para validar a disponibilidade da API.

Exemplo prático (K6):

```
import http from 'k6/http';
import { check } from 'k6';
```

```
export const options = { vus: 1, duration: '5s' };

export default function () {
  const res = http.get('http://localhost:3000/produtos');
  check(res, {
    'status é 200': (r) => r.status === 200,
  });
}
```

3. Teste de Carga (Load Test)

Objetivo: Avaliar o comportamento da aplicação sob o **volume médio esperado de usuários**.

Aplicação prática:

Usado para verificar se o sistema mantém um bom desempenho com a quantidade normal de acessos de produção, sem degradação perceptível.

Exemplo de cenário:

Um e-commerce pode simular 500 usuários simultâneos navegando e realizando compras para confirmar que o tempo médio de resposta permanece abaixo de 500 ms.

Exemplo prático (K6):

```
import http from 'k6/http';
import { check } from 'k6';

export const options = {
  vus: 50,
  duration: '3m',
};

export default function () {
  const res = http.get('http://localhost:3000/produtos');
  check(res, { 'status é 200': (r) => r.status === 200 });
}
```

4. Teste de Estresse e Pico (Stress & Spike Test)

Stress Test

Objetivo: Identificar **até onde o sistema suporta carga crescente** antes de falhar ou apresentar lentidão.

Aplicação prática:

Executado com aumento gradual de usuários para descobrir o ponto em que começam a ocorrer falhas ou respostas lentas.

Exemplo de cenário:

Em uma API bancária, o teste pode subir de 100 para 2000 usuários para medir o ponto em que o banco de dados atinge o limite de conexões.

Exemplo prático (K6):

```
import http from 'k6/http';
import { check } from 'k6';

export const options = {
  stages: [
    { duration: '1m', target: 100 },
    { duration: '2m', target: 500 },
    { duration: '2m', target: 1000 },
    { duration: '1m', target: 0 },
  ],
};

export default function () {
  const res = http.get('http://localhost:3000/usuarios');
  check(res, { 'status é 200': (r) => r.status === 200 });
}
```

Spike Test

Objetivo: Avaliar o **comportamento da aplicação diante de picos súbitos de tráfego**.

Aplicação prática:

Simula situações onde o número de usuários cresce rapidamente em poucos segundos, testando a resiliência da infraestrutura.

Exemplo de cenário:

Durante eventos como Black Friday, o número de acessos pode aumentar repentinamente de 10 para 1000 usuários em poucos segundos.

Exemplo prático (K6):

```
import http from 'k6/http';
import { check } from 'k6';

export const options = {
  stages: [
    { duration: '10s', target: 10 },
    { duration: '10s', target: 500 },
    { duration: '1m', target: 500 },
    { duration: '10s', target: 0 },
  ],
};

export default function () {
  const res = http.get('http://localhost:3000/produtos');
  check(res, { 'status é 200': (r) => r.status === 200 });
}
```

---

5. Teste de Longa Duração (Soak Test)

Objetivo: Avaliar o **comportamento do sistema em uso contínuo**, verificando possíveis degradações com o tempo.

Aplicação prática:

Ideal para identificar vazamentos de memória, lentidão progressiva ou falhas em conexões persistentes.

Exemplo de cenário:

Um sistema de monitoramento pode ser submetido a 50 usuários simultâneos por 12 horas para detectar perda gradual de performance.

Exemplo prático (K6):

```
import http from 'k6/http';
import { check } from 'k6';

export const options = {
  vus: 10,
  duration: '12h',
};

export default function () {
  const res = http.get('http://localhost:3000/produtos');
  check(res, { 'status é 200': (r) => r.status === 200 });
}
```

6. Teste de Limite (Breakpoint Test)

Objetivo: Descobrir **o ponto exato de falha** da aplicação, onde o sistema deixa de responder adequadamente.

Aplicação prática:

Utilizado para determinar o limite máximo suportado pela aplicação antes que erros passem a ocorrer de forma generalizada.

Exemplo de cenário:

Durante uma migração de servidores, o teste de limite pode ajudar a definir a capacidade necessária de usuários simultâneos no novo ambiente.

Exemplo prático (K6):

```
import http from 'k6/http';
import { check } from 'k6';

export const options = {
  stages: [
    { duration: '30s', target: 100 },
    { duration: '30s', target: 300 },
    { duration: '30s', target: 600 },
    { duration: '30s', target: 900 },
    { duration: '30s', target: 1200 },
```

```
  ],
};

export default function () {
  const res = http.get('http://localhost:3000/usuarios');
  check(res, { 'status é 200': (r) => r.status === 200 });
}
```

Conclusão

Cada tipo de teste tem um propósito específico dentro da estratégia de performance:

Tipo de Teste	Objetivo Principal
Smoke Test	Verificar se o sistema está funcional antes dos testes principais
Load Test	Medir o comportamento sob carga esperada
Stress Test	Descobrir o limite de estabilidade
Spike Test	Avaliar a resposta a picos repentinos
Soak Test	Identificar degradação ao longo do tempo
Breakpoint Test	Encontrar o ponto exato de falha

Os testes de performance não apenas validam o sistema, mas também ajudam a **prevenir falhas em produção** e a **dimensionar corretamente os recursos da infraestrutura**.