Escalabilidade e Otimização de APIs com Redis: Implementação e Validação por Testes de Carga

Bruno Bonifácio

Gianluca Andrade Silvestre

Luis Filipe Damiani Colombo

Murilo Salvan

Roger Balcevicz

2025

Redis em Docker

Para viabilizar o mecanismo de cache implementado na API, foi utilizado o **Redis** em um contêiner Docker. Essa abordagem simplifica a configuração do ambiente e garante portabilidade, permitindo que qualquer membro da equipe inicialize o serviço de cache de forma padronizada.

A execução foi feita a partir da imagem oficial **redis:7**, com o seguinte comando:

docker run -d --name classapi-redis -p 6379:6379 redis:7

- **--name classapi-redis**: nome atribuído ao contêiner.
- **-p 6379:6379**: porta padrão do Redis (6379) do contêiner para a máquina local.
- **redis:7**: imagem oficial da versão 7 do Redis.

Após a execução, o log do contêiner indicou a mensagem:

Ready to accept connections tcp

confirmando que o serviço estava em funcionamento e pronto para receber conexões.

A aplicação **NestJS** foi configurada com as variáveis de ambiente:

REDIS_HOST=localhost

REDIS_PORT=6379

Essas variáveis são utilizadas pelo serviço RedisCacheService, responsável por armazenar e recuperar os dados de cache. Assim, ao receber requisições em endpoints como GET /products/{id}, a API verifica primeiramente no Redis se existe um valor armazenado. Caso não exista, a informação é buscada da fonte de dados original, armazenada no Redis e devolvida ao cliente.

O uso do Redis via Docker trouxe os seguintes benefícios:

- **Padronização** do ambiente de cache, independente da máquina de desenvolvimento.
- **Isolamento** do serviço de cache, sem necessidade de instalação local.
- **Portabilidade**, facilitando a replicação do ambiente em produção ou em outros times.

Testes de Carga com Artillery

Para validar o impacto das otimizações aplicadas na API, especialmente a introdução do cache Redis, foi utilizada a ferramenta **Artillery**, que permite simular requisições concorrentes e medir métricas de desempenho como taxa de requisições por segundo (RPS) e tempo de resposta (latência).

O Artillery foi instalado como dependência global via **npm**:

npm install -g artillery

Com a ferramenta disponível, foram definidos dois cenários principais de teste, descritos em arquivos .yml:

1. **Baseline (test-baseline.yml)**
    - Objetivo: simular um tráfego misto, com 70% de requisições para o endpoint GET /products e 30% para GET /products/{id}.
    - IDs aleatórios entre 1 e 8 foram utilizados para refletir acessos reais.
    - O teste foi configurado para crescer gradualmente de 10 até 100 requisições por segundo, em fases de aquecimento, rampa e pico.
    - Métricas obtidas:
        - **p95 ≈ 3 ms** para respostas de sucesso.
        - **p99 ≈ 5 s** apenas em misses iniciais de cache, devido ao atraso proposital configurado no serviço.
        - **0 falhas** durante toda a execução.
2. **Cache (test-cache.yml)**
    - Objetivo: medir especificamente o ganho do cache no endpoint GET /products/1.
    - O cenário consiste em múltiplas requisições consecutivas para o mesmo ID.
    - Resultados observados:
        - Primeira chamada: **~5 s** (miss).
        - Chamadas seguintes: **p95 ≈ 4 ms**, **p99 < 10 ms**.
        - Sustentação de **~438 req/s** sem falhas.

Os testes foram executados com os seguintes comandos:

artillery run test/test-baseline.yml -o test/baseline_pos.json

artillery run test/test-cache.yml -o test/cache_pos.json

O uso do Artillery permitiu comprovar, de forma prática e mensurável, os ganhos de desempenho e escalabilidade da solução proposta.

Conclusão

Os testes comprovaram o impacto positivo da adoção do **cache Redis** no endpoint GET /products/{id}. No cenário com Cache, o tempo de resposta caiu de aproximadamente **5 s (miss inicial)** para **p95 ≈ 4 ms** e **p99 < 10 ms** em acessos subsequentes, com throughput sustentado de **~438 req/s** sem falhas. No cenário Baseline, as requisições para /products e /products/{id} apresentaram **p95 ≈ 3 ms**, com o **p99 ≈ 5 s** apenas em misses pontuais de cache. Assim, conclui-se que a integração com Redis reduziu significativamente a latência e aumentou a escalabilidade da API.

Link GitHub: https://github.com/GiaNinWorld/class-api.git