ANÁLISE DE PERFORMANCE E SEGURANÇA EM APIs RESTful: UM ESTUDO DE CASO SOBRE AUTENTICAÇÃO STATELESS COM JWT E ESTRATÉGIAS DE CACHE COM REDIS

Rubens Lavor[1], Romulo Castillo C[2]

[1]*Univesidade Federal de Santa Catarina, Joinville,Brasil(heryson07@gmail.com)*
[2]*Univesidade Federal de Santa Catarina, Joinville ,Brasil(romulo.castillo@ufsc.br)*

*Resumo:*Este estudo apresenta a keep-lite, uma API RESTful de gerenciamento de notas construída para avaliar segurança e performance. Implementou-se autenticação stateless com JWT, usando uma deny-list no Redis para revogação de tokens, e uma camada de cache Redis. Em testes de carga com k6 e 120 usuários virtuais em ambiente Docker, o cache reduziu a latência média de leitura em 95% e elevou o throughput em 113,5%, mostrando que segurança e caching estratégico viabilizam APIs escaláveis.

Palavras-chave API RESTful; JWT; Redis; cache; performance

INTRODUÇÃO

As APIs RESTful consolidaram-se como padrão para integração de sistemas distribuídos e aplicações web. A capacidade de escalar horizontalmente para atender milhares de requisições simultâneas é um requisito não-funcional crítico (Newman, 2021). Nesse contexto, dois desafios são fundamentais: garantir segurança sem comprometer a performance e minimizar a latência no acesso aos dados.

Arquiteturas modernas privilegiam o modelo stateless, no qual o servidor não armazena sessões de usuário. O padrão de mercado para isso é o JSON Web Token (JWT), que permite autenticação distribuída. Contudo, o JWT introduz um problema conhecido: a dificuldade de revogar um token antes de sua expiração. Para solucionar esse impasse, propõe-se o uso do Redis, banco de dados em memória, para gerenciar uma lista de tokens revogados (deny-list) com alta velocidade.

Paralelamente, a performance de leitura é frequentemente o gargalo de sistemas web. O acesso repetitivo a bancos em disco, como o MongoDB, pode prejudicar o tempo de resposta sob altas cargas (Kleppmann, 2017). Como solução, investiga-se o uso do Redis também como camada de cache, armazenando dados temporários em memória para acesso imediato.

Para validar essas abordagens, foi desenvolvida a keep-lite, uma API de gerenciamento de notas submetida a experimentos controlados em ambiente Docker, comparando seu desempenho com e sem as otimizações propostas. Os objetivos são: (i) implementar autenticação stateless com JWT e deny-list em Redis; e (ii) avaliar experimentalmente o impacto de uma camada de cache em memória sobre a latência e o throughput da API.

A relevância do estudo está em unir, em um mesmo objeto experimental, dois requisitos não-funcionais frequentemente tratados de forma isolada na literatura: segurança e performance. Ao quantificar tanto o ganho de desempenho proporcionado pelo cache quanto o custo da camada de segurança, o trabalho oferece evidências práticas para decisões arquiteturais em APIs RESTful. Este artigo está organizado em fundamentação e metodologia (Material e Métodos), apresentação e discussão dos resultados quantitativos, conclusões e, ao final, apêndices com os principais artefatos de implementação que asseguram a reprodutibilidade dos experimentos.

MATERIAL E MÉTODOS

Fundamentação

A arquitetura de software compreende as decisões estruturais que definem a organização de um sistema, seus componentes e relações, impactando manutenibilidade, escalabilidade, segurança e desempenho (Newman, 2021). Em sistemas distribuídos, ela determina como os serviços se comunicam e se integram. As APIs assumiram papel central nessa integração, permitindo comunicação padronizada e independente de plataforma, movimento acelerado pelos microsserviços, nos quais cada serviço expõe funcionalidades por uma API bem definida.

O estilo arquitetural REST (Representational State Transfer) foi proposto por Fielding (2000) e tornou-se o paradigma dominante para APIs modernas. REST não é um protocolo, mas um conjunto de seis restrições aplicadas sobre protocolos existentes, como o HTTP: (i) Client-Server, separando interface e processamento; (ii) Stateless, exigindo que cada requisição contenha toda a

informação necessária; (iii) Cacheable, permitindo reutilização de respostas; (iv) Uniform Interface, padronizando a comunicação; (v) Layered System, admitindo camadas intermediárias; e (vi) Code on Demand (opcional). Entre elas, a restrição stateless é central para este estudo, pois força mecanismos de autenticação autocontidos, como o JWT, e viabiliza a escalabilidade horizontal, fator que conecta diretamente o modelo ao uso de contêineres e orquestradores como Docker e Kubernetes (Docker Inc., 2025).

Para avaliar o grau de aderência ao REST, utiliza-se o Modelo de Maturidade de Richardson (Fowler, 2010), descrito na Tabela 1. A maioria das APIs em produção situa-se no Nível 2 (Richardson e Ruby, 2007), equilíbrio pragmático entre pureza do modelo e simplicidade de implementação. A keep-lite almeja esse nível.

Tabela 1. Níveis de maturidade do Modelo de Richardson.

Nível	Descrição
0	HTTP apenas como transporte (um único endpoint).
1	Recursos com URIs distintas, sem verbos HTTP.
2	Uso correto de verbos HTTP e códigos de status.
3	HATEOAS: respostas guiam as próximas ações por links.

A segurança em sistemas distribuídos envolve autenticação (“quem você é?”) e autorização (“o que você pode fazer?”) (Hardt, 2012). O modelo tradicional de sessões stateful, em que o servidor guarda os dados do usuário e devolve um identificador de sessão, conflita com a restrição stateless e exige mecanismos complexos de replicação em arquiteturas escaláveis (Newman, 2021; OWASP Foundation, 2024). A alternativa é a autenticação por tokens autocontidos (Jones; Bradley; Sakimura, 2015): após o login, o servidor gera o token e o cliente o envia em cada requisição, validado sem consulta a um armazenamento central (Auth0, 2025).

O JWT é uma string em três partes, cabeçalho, payload (claims) e assinatura. codificadas em Base64Url (OWASP Foundation, 2024). A assinatura garante integridade: o servidor a recalcula para verificar se o token foi modificado. O algoritmo influencia a validação: o HS256 usa uma chave única (mais rápido, mas compartilhada), enquanto o RS256 usa par de chaves (assinatura privada, validação pública), reduzindo o risco em ambientes com vários componentes. Para limitar a exposição, separam-se access tokens de curta duração, enviados a cada requisição, de refresh tokens de duração maior, guardados de forma restrita (ex.: cookie HttpOnly) e usados apenas para renovar o access token (Hardt, 2012).

Mesmo assim, o JWT mantém o problema da invalidação: por ser stateless, permanece válido até a expiração (exp), mesmo após logout. Para contornar isso, adota-se a deny-list: no logout, um identificador único do token (claim jti) é registrado em uma lista consultada a cada requisição. A implementação em Redis armazena cada token revogado com um TTL idêntico ao tempo de expiração restante do JWT, garantindo autolimpeza eficiente e verificação em memória sem comprometer a escalabilidade (Redis Labs, 2023).

A performance é avaliada por latência (tempo de uma requisição) e throughput (requisições por unidade de tempo) (Kleppmann, 2017). O cache explora a localidade de referência, o acesso repetido aos mesmos dados, mantendo-os em memória de acesso rápido. A diferença de desempenho entre RAM (microssegundos) e disco (milissegundos) torna bancos em memória eficientes para baixa latência (Redis Labs, 2023). Utiliza-se o padrão Cache-Aside (Microsoft Corporation, 2021): a aplicação consulta o cache; em caso de cache miss, busca no banco e armazena com TTL; nas próximas requisições o dado vem do cache (cache hit). O Redis, sistema de estruturas de dados em memória de código aberto, desempenha duplo papel: cache de leitura e infraestrutura da deny-list.

Arquitetura da Solução

A keep-lite fornece uma API RESTful simplificada para gerenciamento de notas, servindo de base para avaliar, em ambiente controlado, o impacto do cache com Redis e da autenticação via JWT. A arquitetura segue o estilo hexagonal (ports and adapters), mantendo baixo acoplamento e facilitando a troca de componentes de infraestrutura sem alterar a lógica central, o que simplifica comparar o acesso direto ao banco com o uso de cache, já que o restante da aplicação permanece igual.

A aplicação foi escrita em Kotlin sobre Java 21. O Spring Boot cuida da inicialização e dos endpoints REST, e o Spring Security, integrado ao JWT, valida cada requisição protegida. O armazenamento usa MongoDB, escolhido pela simplicidade com documentos, e o Redis atua em dois papéis: cache para leituras frequentes e repositório temporário da deny-list. A Figura 1 ilustra a organização hexagonal, na qual o domínio permanece isolado das tecnologias de infraestrutura. O gerenciamento de dependências usa Gradle, e a documentação segue OpenAPI/Swagger (Figura 2). Toda a aplicação, com Redis e MongoDB, foi empacotada em contêineres Docker, garantindo um ambiente previsível e reprodutível. O desenvolvimento ocorreu no IntelliJ IDEA Community Edition, com bibliotecas como Spring Web e Spring Validation.

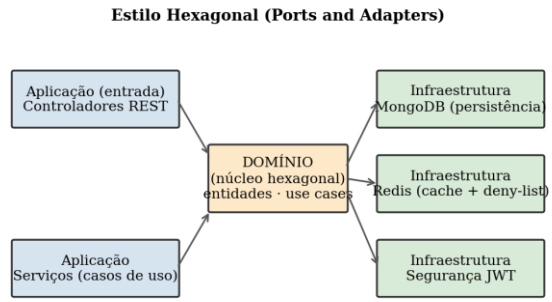

Figura 1. Arquitetura hexagonal (ports and adapters) da keep-lite.

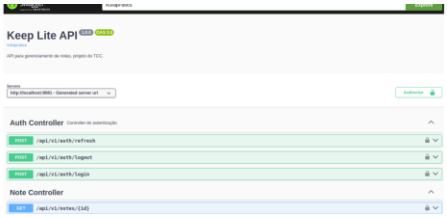

Figura 2. Documentação OpenAPI/Swagger da keep-lite, com os endpoints de autenticação, notas e usuário.

Ambiente e Procedimento Experimental

Os testes foram executados em uma máquina local (Intel Core i7-10875H @ 2.30 GHz, 8 núcleos e 16 threads; 8 GB DDR4; Ubuntu 22.04 LTS), com Docker Engine 20.10.7, Docker Compose 1.29.2, Grafana k6 v0.50.0, JDK Eclipse Temurin 21, Spring Boot 3.3.0, MongoDB 7.0 e Redis 7.2-alpine. O k6 e os serviços em Docker compartilham os mesmos recursos; como a condição foi idêntica para todos os cenários, essa disputa não compromete a comparação.

A massa de dados foi gerada por scripts em k6: o primeiro criou 120 usuários de teste; o segundo autenticou cada um e gerou 50 notas por conta, totalizando 6.000 notas no MongoDB. Essa mesma massa foi utilizada em todas as execuções, garantindo consistência.

Foram definidos dois cenários, variando apenas a habilitação do cache. No Cenário 1 (Baseline), a aplicação executa com as anotações de cache (@Cacheable, @CacheEvict) desabilitadas, servindo todas as leituras (GET /notes) diretamente pelo MongoDB; funciona como grupo de controle. No Cenário 2 (Otimizado), habilitam-se @Cacheable na listagem de notas, @CacheEvict nas escritas e a configuração do CacheManager para serialização JSON, com a camada Redis ativa.

O mesmo script de carga foi aplicado a ambos, simulando leitura intensiva. Cada um dos 120 usuários virtuais (VUs) faz login uma vez (POST /auth/login), executa um loop de 10 leituras (GET /notes) e simula um tempo de pensamento com sleep antes de repetir — proporção desenhada para maximizar cache hits no Cenário 2. A carga foi aplicada em três estágios: rampa de 0 a 120 VUs em 30 s, 120 VUs sustentados por 2 min e descida a 0 em 10 s. Coletaram-se latência (média e p95), throughput (req/s) e taxa de erro (percentual de respostas HTTP 4xx/5xx).

Declaração de uso de IA: *durante a elaboração deste trabalho utilizou-se o assistente Claude (Anthropic) como ferramenta de apoio, restrita à correção gramatical e ortográfica e ao resumo de trechos redigidos pelo autor. A concepção, o desenvolvimento da aplicação, a execução dos experimentos, a análise dos resultados e as conclusões são de autoria integral do autor.*

RESULTADOS E DISCUSSÃO

Os dois cenários foram executados sob carga de 120 VUs em ambiente Docker controlado. A Tabela 1 consolida os resultados. A inclusão do Redis alterou de forma significativa o comportamento da API sob carga, sobretudo em leitura intensiva.

Tabela 2. Comparativo de performance: Baseline (sem cache) vs. Otimizado (com cache).

Métrica	Baseline	Com Cache	Melhoria
Taxa de Falha	0,00%	0,00%	N/A
RPS (req/s)	153,76	328,39	+113,5%
Latência média	376,54 ms	18,67 ms	-95,0%
Latência p(95)	844,36 ms	70,70 ms	-91,6%

Impacto na Latência

No Baseline, a API acessa o MongoDB a cada requisição; sob carga, a latência cresce progressivamente, indicando saturação do banco — comportamento esperado, pois operações em disco dependem de I/O físico e controle de concorrência. A operação GET /notes apresentou latência média de 376,54 ms e p95 de 844,36 ms, evidenciando o banco como gargalo principal.

Com o cache Redis ativo, após o primeiro acesso (cache miss) as requisições passam a ser atendidas em memória. A latência média caiu para 18,67 ms (redução de 95%) e o p95 para 70,70 ms (melhoria de 91,6%), passando da ordem de centenas para dezenas de milissegundos e atingindo o threshold de 100 ms estabelecido para a pesquisa (Figura 3). A redução da cauda de latência melhora a previsibilidade e a experiência do usuário, confirmando que mecanismos de cache podem reduzir a latência em duas ordens de grandeza (Kleppmann, 2017).

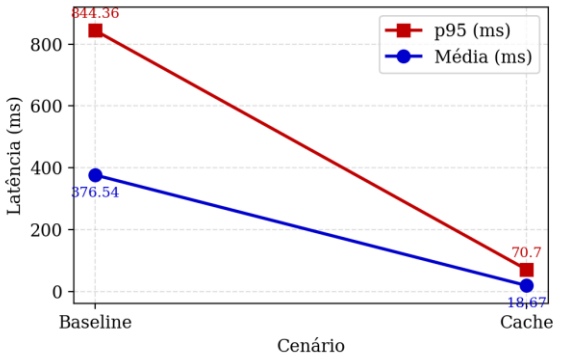

Figura 3. Latência por cenário (média e p95) durante leituras GET /notes.

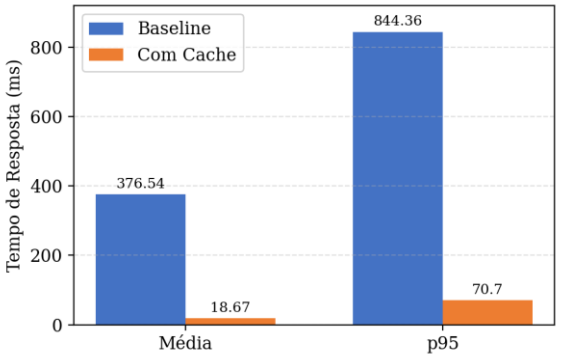

Figura 4. Comparativo de latência (média e p95) entre os cenários.

Impacto no Throughput

A redução da latência permitiu que os mesmos 120 VUs completassem mais ciclos no mesmo intervalo. O throughput mais que dobrou, de 153,76 req/s (Baseline) para 328,39 req/s (Com Cache), aumento de 113,5% (Figura 5). Isso indica que o cache eliminou o gargalo do banco e liberou recursos para um volume maior de requisições, coerente com a literatura (Amazon Web Services, 2024).

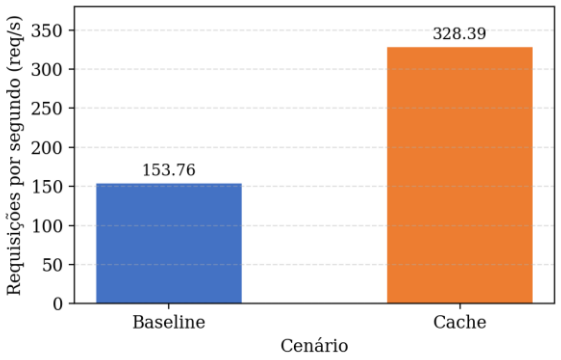

Figura 5. Throughput (req/s) da API com e sem cache Redis.

Overhead de Autenticação

A arquitetura introduz um overhead em toda requisição autenticada: a validação criptográfica da assinatura do JWT (CPU-bound) e uma consulta chave-valor à deny-list no Redis. Como leituras em memória no Redis têm latência de microssegundos a poucos milissegundos (Redis Labs, 2023) e a validação de assinatura é uma operação de CPU muito rápida, o overhead total de segurança é de ordem de grandeza muito inferior ao acesso ao banco, não representando gargalo e justificando seu uso em troca da capacidade de revogação de tokens. A taxa de erro permaneceu em 0,00% em todos os cenários.

A Tabela 3 detalha as métricas brutas de tempo de resposta (em ms) extraídas do k6 para a operação GET /notes, evidenciando a redução em toda a distribuição — da mediana ao percentil 95 — e não apenas na média. Observa-se também a queda do valor máximo, de 2.558 ms para 327 ms, o que reforça a eliminação da cauda de latência associada à saturação do MongoDB.

Tabela 3. Métricas detalhadas de tempo de resposta (ms) da operação GET /notes.

Métrica	**Baseline**	**Com Cache**
Média	376,54	18,67
Mínimo	6,74	2,36
Mediana	332,55	9,21
p(90)	710,12	51,17
p(95)	844,36	70,70
Máximo	2.558,01	327,17

Quanto à execução global, o Baseline completou 25.090 verificações a uma taxa de 153,76/s, enquanto o cenário com cache completou 53.440 a 328,39/s no mesmo intervalo — confirmando, do ponto de vista da vazão agregada, o ganho de 113,5%. Em ambos, todas as verificações foram bem-sucedidas (100%), sem requisições falhas, o que indica estabilidade da aplicação mesmo sob a carga máxima de 120 VUs.

De modo geral, os resultados indicam que o aumento de complexidade arquitetural decorrente da introdução do Redis é amplamente compensado pelos ganhos de performance e escalabilidade. O padrão Cache-Aside mostrou-se adequado ao perfil de leitura intensiva da aplicação, e o estilo hexagonal permitiu alternar entre os cenários sem alterar a lógica de negócio, reforçando o valor da separação entre domínio e infraestrutura para experimentação controlada. Assim, a hipótese inicial é confirmada: o cache em memória reduz significativamente a latência e eleva o throughput sem comprometer a natureza stateless proporcionada pelo JWT.

Os ganhos observados são consistentes com a literatura. Kleppmann (2017) aponta que mecanismos de cache podem reduzir a latência em até duas ordens de grandeza ao eliminar operações de I/O e bloqueios de concorrência — o que se confirma na queda da latência média de 376,54 ms para 18,67 ms, redução de aproximadamente 20 vezes. De forma análoga, a

literatura de boas práticas de cache (Amazon Web Services, 2024) destaca que o alívio de pressão sobre o banco libera recursos computacionais para atender maior volume de requisições, comportamento evidenciado pela duplicação do throughput. Cabe ressaltar que esses ganhos se manifestam de modo mais expressivo em cargas de leitura intensiva; em cenários com proporção elevada de escritas, a invalidação frequente do cache (@CacheEvict) tende a reduzir a taxa de acerto, o que constitui uma direção relevante para investigações futuras.

CONCLUSÃO

Este trabalho investigou a combinação de autenticação stateless com JWT e estratégias de cache em memória com Redis sobre a segurança, a performance e a escalabilidade de uma API RESTful, validada pela keep-lite. Demonstrou-se a viabilidade do JWT em ambientes stateless e, para sua limitação de revogação, implementou-se uma deny-list de access tokens em Redis, com revogação imediata e eliminação de estado no servidor.

Os experimentos com 120 usuários virtuais mostraram ganhos expressivos: a latência média de leitura reduziu 95% e o p95 caiu de 844,36 ms para 70,70 ms (-91,6%), enquanto o throughput aumentou 113,5% (de 153,76 para 328,39 req/s), sem falhas. O MongoDB deixou de ser gargalo em leitura intensiva. Conclui-se que a combinação de autenticação segura (JWT com deny-list em Redis) e caching estratégico é eficaz para construir APIs RESTful seguras, performáticas e escaláveis, evidenciando que performance e segurança são pilares complementares.

Como limitações, os testes ocorreram em ambiente local único, o foco foi em leitura, e o overhead de segurança foi analisado teoricamente. Como trabalhos futuros, sugerem-se experimentos em ambiente escalável (Kubernetes), avaliação de cenários de escrita intensiva com invalidação de cache (@CacheEvict) e comparação entre estratégias de serialização (JSON, MessagePack, Protobuf).

AGRADECIMENTOS

O autor agradece aos professores e colegas que contribuíram com discussões e revisões ao longo do desenvolvimento deste trabalho.

REFERÊNCIAS

AMAZON WEB SERVICES. Caching Strategies and Best Practices. 2024. Disponível em: https://docs.aws.amazon.com/whitepapers/latest/caching-best-practices.

AUTH0. JWTs in Practice. 2025. Disponível em: https://auth0.com.

AWAD, A. et al. Performance evaluation of redis in-memory data store. In: IEEE International Conference on Computing. IEEE, 2020.

DOCKER INC. Docker Documentation. 2025. Disponível em: https://docs.docker.com.

FIELDING, R. T. Architectural Styles and the Design of Network-based Software Architectures. Tese (Doutorado) — University of California, Irvine, 2000.

FOWLER, M. Richardson Maturity Model. 2010. Disponível em: https://martinfowler.com/articles/richardsonMaturityModel.html.

HARDT, D. RFC 6749: The OAuth 2.0 Authorization Framework. IETF, 2012.

JONES, M.; BRADLEY, J.; SAKIMURA, N. RFC 7519: JSON Web Token (JWT). IETF, 2015.

KLEPPMANN, M. Designing Data-Intensive Applications. O'Reilly Media, 2017.

MICROSOFT CORPORATION. Cache-Aside Pattern. 2021. Disponível em: https://learn.microsoft.com/azure/architecture/patterns/cache-aside.

NEWMAN, S. Building Microservices. 2. ed. O'Reilly Media, 2021.

OWASP FOUNDATION. OWASP Cheat Sheet Series: JSON Web Token Security. 2024. Disponível em: https://cheatsheetseries.owasp.org.

REDIS LABS. Redis Documentation. 2023. Disponível em: https://redis.io/docs.

RICHARDSON, L.; RUBY, S. RESTful Web Services. O'Reilly Media, 2007.

APÊNDICE A – ESTRUTURA DE DIRETÓRIOS

A organização dos diretórios reflete o estilo hexagonal (ports and adapters), separando as camadas de aplicação, domínio e infraestrutura, conforme a estrutura a seguir.

Estrutura de pastas

```
keep-lite/
  build.gradle.kts
  docker-compose.yml
  k6-scripts/
  src/main/kotlin/.../keeplite/
    application/        (Ports de entrada)
      rest/  (auth/ note/ user/)
      service/          (casos de uso)
    domain/             (logica de negocio)
      note/  (entity/ repository/ usecase/)
      user/  (entity/ repository/ usecase/)
    infrastructure/     (Adapters de saida)
      config/  (Spring Security, Redis...)
      persistence/mongo/ (document/ mapper/)
      security/jwt/
```

APÊNDICE B – DOCKER COMPOSE

A composição a seguir orquestra os três serviços (MongoDB, Redis e a API) em uma rede dedicada, com volumes nomeados para persistência.

docker-compose.yml

```
services:
  mongo-db:
    image: mongo:7.0
    container_name: keeplite-mongo
    ports: ["27017:27017"]
    environment:
      - MONGO_INITDB_ROOT_USERNAME=root
      - MONGO_INITDB_ROOT_PASSWORD=password
    volumes: [mongo_data:/data/db]
    networks: [keeplite-network]
  redis-cache:
    image: redis:7.2-alpine
    container_name: keeplite-redis
```

```
    ports: ["6379:6379"]
    command: redis-server --requirepass redispassword
    volumes: [redis_data:/data]
    networks: [keeplite-network]
  keep-lite-api:
    build: .
    container_name: keeplite-api
    depends_on: [mongo-db, redis-cache]
    ports: ["8080:8080"]
    environment:
      - SPRING_PROFILES_ACTIVE=docker
      - SPRING_DATA_MONGODB_URI=mongodb://root:
        password@mongo-db:27017/keeplite
      - SPRING_DATA_REDIS_HOST=redis-cache
      - SPRING_DATA_REDIS_PORT=6379
      - SPRING_DATA_REDIS_PASSWORD=redispassword
    networks: [keeplite-network]
volumes: {mongo_data:, redis_data:}
networks: {keeplite-network: {driver: bridge}}
```

APÊNDICE C – SCRIPT DE TESTE DE CARGA (k6)

Trecho central do script load-test-read-intensive.js, que executa o loop de leitura intensiva por VU, reutilizando o token de acesso entre iterações.

load-test-read-intensive.js

```
export const options = {
  stages: [
    { duration: '30s', target: 120 },
    { duration: '2m',  target: 120 },
    { duration: '10s', target: 0 },
  ],
  thresholds: {
    'http_req_failed': ['rate<0.01'],
    'get_notes_duration': ['p(95)<100'],
  },
};
export default function () {
  // login uma vez por VU; reusa o token
  // loop de 10 leituras GET /notes
  for (let i = 0; i < 10; i++) {
    const r = http.get(`${baseUrl}/notes`,
                       authHeaders);
    check(r, {'ok': (r)=> r.status===200});
    sleep(0.1);
  }
  sleep(Math.random()*2 + 1); // think time
}
```

Os estágios definidos nas opções do k6 controlam a rampa de carga (subida, sustentação e descida), enquanto os thresholds estabelecem critérios automáticos de aprovação: taxa de falha inferior a 1% e p95 da operação de leitura abaixo de 100 ms. A reutilização do token de acesso entre as iterações de cada usuário virtual reproduz o comportamento de uma sessão autenticada real, na qual o login ocorre uma única vez e as leituras subsequentes apenas apresentam o token já obtido. Os scripts de geração da massa de dados (criação de usuários e de notas) seguem estrutura análoga e estão omitidos por brevidade.

O conjunto de artefatos apresentado nos Apêndices A a C foi reunido com o propósito de assegurar a reprodutibilidade dos experimentos. A combinação da estrutura modular (Apêndice A), da definição declarativa do ambiente em contêineres (Apêndice B) e dos scripts de carga parametrizados (Apêndice C) permite que terceiros reconstruam o cenário de testes e verifiquem os resultados relatados. Esse cuidado com a reprodutibilidade é particularmente importante em estudos de performance, nos quais pequenas variações de ambiente — versões de bibliotecas, recursos de hardware ou configuração de rede — podem alterar de forma significativa as medições de latência e throughput. Ao fixar explicitamente as versões de cada serviço e isolar a execução em contêineres, busca-se reduzir tais variações e tornar a comparação entre os cenários a mais justa possível.