Laporan Final Projek

Containerization dengan Docker

Dosen Pengampu : Ferdi Chahyadi, Skom, M.Cs

Laporan Ini Dibuat Untuk Memenuhi Tugas Proyek Mata Kuliah : Sistem Operasi

Di Susun Oleh : Haciendadelpart

Dzaky Ribal Faiz 2401020035

Akbar Risky Lingga 2401020003

M.Al-Fikry Akbar 2401020031

Al-Adhlu sodri niwrad 2401020015

PRODI TEKNIK INFORMATIKA

FAKULTAS TEKNIK & TEKNOLOGI KEMARITIMAN

UNIVERSITAS MARITIMIN RAJA ALI HAJI

2025

Abstrak

Perkembangan teknologi virtualisasi telah bertransformasi dari penggunaan Virtual Machine yang berat menuju solusi containerization yang jauh lebih efisien dan ringan. Proyek ini bertujuan untuk mengimplementasikan dan menganalisis mekanisme manajemen sumber daya (resource management) pada platform Docker menggunakan fitur kernel Linux yang dikenal sebagai Control Groups (Cgroups). Implementasi dilakukan pada sistem operasi Fedora Linux dengan membangun arsitektur microservices yang terdiri dari layanan aplikasi web berbasis Python-Flask (versi 3.10) dan sistem manajemen basis data MySQL (versi 8.0).

Metodologi penelitian ini melibatkan pembuatan custom image melalui Dockerfile yang dioptimasi dan pengorkestrasian multi-kontainer menggunakan Docker Compose. Inti dari penelitian ini adalah pengujian komparatif antara dua unit kontainer aplikasi: kontainer pertama tanpa batasan sumber daya (unlimited) dan kontainer kedua dengan batasan sumber daya yang ketat sebesar 0.5 CPU (50% dari satu core) serta limitasi memori sebesar 128 MiB. Pengujian dilakukan menggunakan skenario beban kerja komputasi intensif (high-load mathematical loop) yang berjalan selama 20 detik untuk mengukur tingkat throughput komputasi (jumlah iterasi) yang mampu diselesaikan oleh masing-masing unit.

Hasil eksperimen menunjukkan perbedaan performa yang sangat signifikan dan presisi secara matematis. Kontainer tanpa batasan mampu menyelesaikan sebanyak 132.825 iterasi, sementara kontainer yang dibatasi hanya mampu menyelesaikan 66.194 iterasi dalam jendela waktu yang sama. Rasio perbandingan pekerjaan sebesar kurang lebih 2:1 ini membuktikan bahwa pembatasan CPU sebesar 0.5 melalui mekanisme Cgroups bekerja secara akurat di level kernel sistem operasi. Analisis melalui perintah docker stats memperkuat temuan ini dengan menunjukkan penggunaan CPU kontainer terbatas yang tertahan secara konsisten pada angka 49,52% hingga 50,00%, tanpa mengalami kegagalan sistem (crash).

Penelitian ini menyimpulkan bahwa teknologi containerization dengan penerapan resource limits tidak hanya menjamin konsistensi lingkungan aplikasi, tetapi juga memberikan kontrol granular bagi administrator sistem untuk mencegah fenomena noisy neighbor. Mekanisme ini memastikan stabilitas infrastruktur host dengan mengisolasi sumber daya secara efektif, sehingga setiap layanan dapat berjalan secara optimal tanpa mengganggu kinerja layanan lainnya dalam ekosistem shared-resource.

Bab 1

Pendahuluan

1.1 Latar Belakang

Arsitektur pengembangan perangkat lunak saat ini telah bergeser secara masif dari model monolitik menuju model microservices. Pergeseran ini didorong oleh kebutuhan industri akan aplikasi yang lebih skalabel, modular, serta efisien dalam siklus deployment. Kendati demikian, adopsi microservices menghadirkan tantangan teknis yang kompleks, khususnya terkait standardisasi lingkungan pengembangan (environment consistency) dan efisiensi pengelolaan sumber daya pada infrastruktur server.

Sebelum teknologi kontainer populer, isolasi aplikasi umumnya mengandalkan Virtual Machine (VM). Namun, VM memiliki redundansi sumber daya yang tinggi karena setiap unitnya harus menjalankan sistem operasi tamu (Guest OS) secara utuh di atas hypervisor. Hal ini menyebabkan overhead yang signifikan pada penggunaan CPU, RAM, dan kapasitas penyimpanan. Sebagai solusi yang lebih efisien, muncul teknologi containerization melalui platform Docker. Berbeda dengan VM, Docker bekerja dengan berbagi kernel sistem operasi host, sehingga kontainer menjadi jauh lebih ringan dan memiliki waktu booting yang instan.

Meskipun kontainer menawarkan isolasi logis yang kuat, secara fisik setiap kontainer tetap berbagi sumber daya perangkat keras yang sama pada mesin host. Tanpa adanya kebijakan manajemen sumber daya yang ketat, satu kontainer yang mengalami lonjakan beban kerja ekstrem atau kegagalan logika kode—seperti infinite loop—dapat mengonsumsi seluruh siklus CPU host secara tidak terkontrol. Kondisi ini memicu fenomena Noisy Neighbor Effect, di mana satu layanan mengganggu stabilitas layanan lain yang berada di dalam host yang sama, bahkan berpotensi menyebabkan kegagalan sistem total (downtime).

Guna mengatasi risiko tersebut, sistem operasi Linux menyediakan fitur Control Groups atau Cgroups yang terintegrasi secara langsung dalam ekosistem Docker. Melalui mekanisme Cgroups, administrator sistem dapat menerapkan batasan keras (hard limits) terhadap kuota CPU dan memori yang boleh digunakan oleh sebuah kontainer. Proyek ini difokuskan untuk melakukan pengujian empiris terhadap presisi limitasi Cgroups pada Docker dengan membandingkan kinerja aplikasi berbasis Flask-MySQL dalam dua skenario berbeda: kondisi tanpa batasan sumber daya (unlimited) dan kondisi terbatas secara spesifik sebesar 0.5 CPU.

1.2 Identifikasi Masalah

Berdasarkan konteks yang telah diuraikan pada latar belakang, maka permasalahan utama yang akan dikaji dalam proyek ini dirumuskan sebagai berikut:

1 Bagaimana metodologi pembuatan Dockerfile yang efisien untuk melakukan standardisasi paket aplikasi Flask dan dependensinya agar dapat berjalan secara konsisten di berbagai lingkungan?

2 Bagaimana mengonfigurasi Docker Compose untuk mengorkestrasi layanan aplikasi dan database MySQL sehingga dapat berkomunikasi dalam satu jaringan terisolasi?

3 Bagaimana mekanisme penerapan limitasi sumber daya CPU dan Memori (Cgroups) dilakukan melalui konfigurasi berkas YAML pada Docker?

4 Sejauh mana tingkat akurasi dan efektivitas limitasi CPU sebesar 0.5 core dalam membatasi throughput komputasi aplikasi saat diberikan beban kerja intensif?

5 Apakah pembatasan sumber daya tersebut berdampak pada stabilitas operasional kontainer tanpa menyebabkan kegagalan fungsi aplikasi?

1.3 Batasan Masalah

Agar pembahasan dalam laporan ini tetap fokus, mendalam, dan sesuai dengan batasan waktu yang ada, penulis menetapkan batasan masalah sebagai berikut:

1 Sistem Operasi Host: Pengujian dilakukan secara eksklusif pada distribusi Fedora Linux sebagaimana hasil instalasi pada tahap progres awal.

2 Platform Container: Platform yang digunakan terbatas pada Docker Engine dan Docker Compose.

3 Stack Teknologi: Aplikasi dibangun menggunakan bahasa pemrograman Python (Flask) sebagai backend dan MySQL 8.0 sebagai sistem manajemen basis data.

4 Parameter Resource: Fokus limitasi sumber daya adalah CPU Limit 0.5 dan Memory Limit 128 MiB.

5 Metode Pengujian: Analisis performa dilakukan melalui perbandingan jumlah iterasi matematika berat yang dijalankan selama tepat 20 detik pada endpoint /load-cpu.

1.4 Tujuan Projek

Tujuan yang ingin dicapai melalui pelaksanaan proyek ini adalah:

1 Berhasil membangun custom image Docker yang portabel untuk aplikasi Flask yang terintegrasi dengan MySQL.

2 Mengimplementasikan isolasi sumber daya (CPU dan Memori) pada layanan microservices menggunakan fitur Cgroups.

3 Memperoleh data kuantitatif mengenai perbedaan performa antara kontainer unlimited dan limited sebagai bukti keberhasilan penerapan kebijakan resource management.

BAB 2

Landasan Teori

2.1 Pengertian Docker

Docker merupakan sebuah platform terbuka (open-source) berbasis kontainer yang merevolusi cara pengembang perangkat lunak dalam membangun, mendistribusikan, dan menjalankan aplikasi. Secara teknis, Docker menyediakan lingkungan terisolasi yang disebut kontainer, di mana aplikasi dapat berjalan secara mandiri tanpa terpengaruh oleh konfigurasi global pada sistem operasi host.

Filosofi utama yang mendasari teknologi ini adalah "Build, Ship, and Run Anywhere". Filosofi ini lahir sebagai solusi atas masalah klasik dalam dunia pengembangan perangkat lunak, yaitu ketidakkonsistenan lingkungan (environment drift).

1. Build: Pengembang dapat mengemas aplikasi beserta seluruh pustaka (libraries), konfigurasi, dan dependensi spesifik ke dalam sebuah unit standar yang disebut Docker Image.

2. Ship: Image yang telah dibuat dapat dipindahkan dengan mudah ke repositori pusat (Docker Registry) atau langsung ke server tujuan tanpa perlu khawatir akan perbedaan versi OS.

3. Run Anywhere: Aplikasi dijamin akan berjalan dengan perilaku yang benar-benar identik, baik itu di laptop pengembang, server pengujian (staging), hingga infrastruktur cloud skala besar.

Dengan Docker, masalah "it works on my machine" (aplikasi berjalan di laptop pengembang tapi error di server) dapat dieliminasi sepenuhnya. Hal ini karena kontainer membawa serta seluruh ekosistem yang dibutuhkannya untuk hidup, menjadikannya unit perangkat lunak yang sangat portabel dan reliabel.

2.3 Komponene Utma Docker

Ekosistem Docker terdiri dari berbagai komponen yang saling terintegrasi untuk membentuk sebuah sistem virtualisasi yang utuh. Pemahaman mendalam mengenai komponen-komponen ini sangat krusial karena setiap elemen memiliki peran spesifik dalam memastikan aplikasi dapat berjalan secara terisolasi dan konsisten.

2.3.1 Docker Image

Docker Image adalah sebuah template bersifat read-only (hanya baca) yang menjadi pondasi dasar bagi setiap kontainer. Secara teknis, sebuah image terdiri dari sekumpulan instruksi yang mengemas kode aplikasi, pustaka (libraries), dependensi, serta konfigurasi lingkungan yang diperlukan.

Satu hal yang membuat Docker Image unik dan efisien adalah penggunaan Union File System (UnionFS). Mekanisme ini bekerja dengan prinsip layering:

1. Base Layer: Lapisan paling bawah yang biasanya berisi sistem operasi minimal (seperti Python:3.10-slim atau Alpine Linux).

2. Intermediate Layers: Setiap instruksi dalam Dockerfile (seperti RUN pip install) akan menciptakan satu lapisan baru. Jika terjadi perubahan kode, Docker hanya perlu membangun ulang lapisan yang berubah, sedangkan lapisan lainnya diambil dari cache.

3. Immutability: Sekali image dibangun, isinya tidak dapat diubah. Karakteristik ini menjamin bahwa apa yang berjalan di server produksi akan benar-benar identik dengan apa yang diuji di lingkungan pengembangan.

2.3.2 Docker Container

Jika image adalah sebuah kelas atau cetak biru, maka kontainer adalah instansiasi atau wujud nyata yang berjalan dari image tersebut. Kontainer merupakan proses yang terisolasi di dalam ruang pengguna (user space) pada sistem operasi host.

Perbedaan utama antara kontainer dan image terletak pada Writable Layer (Lapisan Tulis). Saat sebuah kontainer dijalankan, Docker menambahkan satu lapisan tipis di atas tumpukan lapisan read-only milik image. Semua perubahan data, penulisan log, atau pembuatan berkas baru selama kontainer berjalan akan disimpan di lapisan tulis ini. Namun, karena sifat kontainer yang ephemeral (sementara), lapisan tulis ini akan musnah saat kontainer dihapus, kecuali jika dihubungkan dengan Docker Volume.

2.3.3 Docker Engine

Docker Engine adalah aplikasi client-server yang mengelola seluruh siklus hidup kontainer. Docker Engine terdiri dari tiga komponen utama yang bekerja secara sinkron:

1. Server (Docker Daemon/dockerd): Proses latar belakang yang terus berjalan dan bertanggung jawab penuh atas pembuatan, pengelolaan, dan pengawasan objek-objek Docker (images, containers, networks, volumes).

2. REST API: Antarmuka komunikasi yang memungkinkan program lain atau pengguna untuk berinteraksi dengan Docker Daemon.

3. Command Line Interface (CLI/docker): Antarmuka baris perintah yang digunakan pengguna untuk memasukkan perintah. Saat pengguna mengetik docker run, CLI akan mengirimkan permintaan tersebut melalui REST API ke Docker Daemon untuk dieksekusi.

2.3.4 Dockerfile

Dockerfile adalah dokumen teks yang berisi urutan perintah teknis untuk merakit sebuah image. Penggunaan Dockerfile memastikan proses pembangunan lingkungan aplikasi bersifat reproducible (dapat diulang). Berikut adalah bedah detail beberapa instruksi utama yang digunakan dalam proyek ini:

- FROM: Menentukan base image yang akan digunakan. Ini adalah titik awal dari setiap Dockerfile.

- WORKDIR: Menetapkan direktori kerja di dalam kontainer. Semua perintah selanjutnya akan dijalankan di folder ini.

- COPY / ADD: Menyalin berkas dari mesin host (seperti kode program app.py) ke dalam sistem berkas kontainer.

- RUN: Mengeksekusi perintah selama proses build, seperti menginstal dependensi melalui pip install.

- CMD: Menentukan perintah utama yang akan dijalankan secara otomatis saat kontainer baru saja aktif. Berbeda dengan RUN, CMD tidak dijalankan saat proses build.

- EXPOSE: Memberikan informasi mengenai port jaringan mana yang akan dibuka oleh kontainer (misalnya port 5000 untuk Flask).

2.3.5 Docker Registery

Docker Registry adalah tempat penyimpanan dan distribusi untuk Docker Images. Registry yang paling populer adalah Docker Hub, yang bersifat publik dan berisi ribuan image resmi dari berbagai pengembang perangkat lunak di seluruh dunia.

- Push: Proses mengunggah image hasil build lokal ke Registry.
- Pull: Proses mengunduh image dari Registry ke mesin lokal. Dalam proyek ini, sistem secara otomatis melakukan pull terhadap image resmi mysql:8 dan python:3.10 sebagai komponen utama sistem.

2.3.6 Docker Compose

Dalam arsitektur microservices, sebuah aplikasi jarang berdiri sendiri. Biasanya diperlukan layanan pendukung seperti database. Docker Compose hadir sebagai alat orkestrasi sederhana yang memungkinkan pengelolaan banyak kontainer melalui satu berkas konfigurasi tunggal (docker-compose.yml). Dengan Compose, hubungan antar layanan (seperti Flask yang membutuhkan MySQL) dapat didefinisikan secara deklaratif, lengkap dengan parameter jaringan dan batasan sumber daya (resource limits).

2.4 Mekanisme Control Groups

Pilar utama pengujian dalam proyek ini adalah penggunaan Cgroups. Cgroups adalah fitur kernel Linux yang digunakan Docker untuk mengalokasikan dan membatasi sumber daya perangkat keras secara fisik.

1. CPU Limitation (0.5 CPU): Menggunakan mekanisme quota pada kernel, limit 0.5 memerintahkan sistem untuk hanya memberikan jatah waktu prosesor sebesar 50ms untuk setiap periode 100ms kepada kontainer. Hal ini memaksa aplikasi bekerja pada kapasitas setengah core.
2. Memory Limitation (128 MiB): Cgroups memantau penggunaan halaman memori (memory pages). Jika penggunaan memori mencapai batas kaku (hard limit) 128 MiB, kernel akan mencegah kontainer mengambil memori lebih banyak untuk menjaga stabilitas sistem Fedora secara keseluruhan.

BAB 3

Perancangan Arsitektur Sistem

3.1 Desain Arsitektur Microservices

Dalam perancangan sistem ini, penulis menerapkan konsep arsitektur microservices yang memisahkan antara lapisan logika aplikasi (Application Layer) dan lapisan penyimpanan data (Data Layer). Pemisahan ini dirancang bukan tanpa alasan; dalam ekosistem Docker, pemisahan layanan bertujuan untuk menciptakan skalabilitas dan isolasi yang lebih baik.

Rancangan arsitektur ini melibatkan dua buah layanan utama yang berjalan secara independen namun saling terinterkoneksi:

1. Layanan Backend (Flask): Dirancang untuk menangani permintaan HTTP dari pengguna. Di dalamnya terdapat logika komputasi intensif yang akan digunakan sebagai instrumen pengujian beban CPU.

2. Layanan Database (MySQL 8.0): Bertindak sebagai repositori data barang. Layanan ini dirancang untuk tetap pasif namun siap menerima koneksi dari layanan Flask kapan saja.

Penerapan Docker dalam arsitektur ini memungkinkan setiap layanan memiliki sistem berkas dan dependensi yang berbeda. Misalnya, layanan Flask hanya berisi pustaka Python, sementara layanan MySQL hanya berisi mesin basis data, tanpa ada konflik dependensi di antara keduanya.

3.2 Perancangan Topologi Jaringan Internal

Salah satu aspek krusial dalam perancangan ini adalah komunikasi antar-kontainer. Penulis merancang sebuah jaringan virtual bertipe Bridge Network. Secara teknis, jaringan ini bertindak sebagai Virtual Switch yang menghubungkan kontainer-kontainer di dalam satu host Fedora Linux.

Dalam perancangan jaringan ini, penulis menetapkan beberapa aturan teknis:

- Service Discovery: Setiap kontainer dalam jaringan ini akan didaftarkan ke dalam sistem DNS internal Docker. Hal ini memungkinkan kontainer Flask untuk memanggil kontainer database menggunakan nama host db, yang jauh lebih stabil dibandingkan menggunakan alamat IP internal yang dapat berubah-ubah setiap kali kontainer dimulai ulang.

- Isolasi Port: Hanya port aplikasi (5000 dan 5001) yang diekspos ke jaringan luar (host), sementara port MySQL (3306) hanya dibuka untuk komunikasi internal antar-kontainer. Rancangan ini bertujuan untuk meningkatkan keamanan sistem agar database tidak dapat diakses langsung dari jaringan publik.

3.3 Perancangan Paramaeter Resource Management

Inti dari proyek ini terletak pada perancangan batasan sumber daya yang akan diterapkan melalui fitur Control Groups (Cgroups) pada kernel Linux. Penulis merancang dua skema alokasi sumber daya yang kontras untuk mendapatkan data perbandingan yang valid:

3.3.1 Skema Kontainer Tanpa Batas

Kontainer ini dirancang untuk berjalan tanpa batasan eksplisit dari Docker. Secara teoretis, kontainer ini diperbolehkan untuk mengonsumsi seluruh siklus CPU dan memori yang tersedia pada sistem host Fedora. Skema ini berfungsi sebagai variabel kontrol untuk mengetahui performa maksimal perangkat keras dalam kondisi beban kerja tinggi.

3.3.2 Skema Kontainer Terbatas

Kontainer ini dirancang dengan parameter limitasi yang ketat.

- Limitasi CPU 0.5: Dalam perancangan ini, penulis menetapkan bahwa kontainer hanya boleh menggunakan 50% dari satu core CPU. Secara teknis, ini diatur melalui mekanisme CPU Quota pada Cgroups, di mana kontainer hanya diberi jatah waktu eksekusi sebesar 50.000 mikrodetik untuk setiap jendela waktu 100.000 mikrodetik.
- Limitasi Memori 128MB: Batas ini ditetapkan untuk menguji ketahanan aplikasi Flask yang berjalan di atas Python. Penulis memilih angka 128MB karena dianggap sebagai ambang batas yang cukup ketat namun masih memungkinkan bagi aplikasi Flask untuk beroperasi secara normal

3.4 Perancangan Logika Pengujian Beban

Untuk membuktikan efektivitas limitasi tersebut, penulis merancang sebuah algoritma khusus di dalam aplikasi. Algoritma ini dirancang untuk melakukan tugas matematika yang tidak berguna (pointless mathematical computation) secara terus-menerus selama jendela waktu yang ditentukan.

Alur Logika Pengujian:

1. Triggering: Pengguna melakukan akses ke endpoint /load-cpu.
2. Timing: Aplikasi memulai penghitungan waktu selama tepat 20 detik menggunakan pustaka time di Python.
3. Heavy Looping: Selama 20 detik tersebut, aplikasi menjalankan perulangan while yang melakukan operasi matematika kompleks (seperti akar kuadrat atau logaritma).
4. Counting: Setiap satu kali putaran perulangan berhasil diselesaikan, variabel iterations akan bertambah satu.

5. Output: Setelah waktu habis, sistem akan menghentikan perulangan dan mengirimkan total iterations tersebut ke browser.

Logika ini memastikan bahwa kita mendapatkan data kuantitatif yang objektif. Jika limitasi 0.5 CPU bekerja dengan benar, maka jumlah iterations pada kontainer terbatas seharusnya berada di kisaran 50% dari jumlah iterations pada kontainer tanpa batas

BAB 4

Implementasi

4.1 Persiapan dan Instalasi Lingkungan Docker pada Fedora Linux

Tahap awal implementasi dimulai dengan penyiapan lingkungan kerja pada sistem operasi host, yaitu Fedora Linux. Karena Docker sangat bergantung pada kernel sistem operasi untuk menjalankan isolasi kontainer, proses instalasi harus dilakukan secara presisi menggunakan repositori resmi guna menjamin stabilitas.

Langkah-langkah Instalasi Teknis:

1. Penambahan Repositori: Langkah pertama adalah mengintegrasikan sistem manajemen paket Fedora (dnf) dengan repositori resmi Docker melalui perintah dnf config-manager addrepo-from-repofile=https://download.docker.com/linux/fedora/docker-ce.repo. Penambahan ini dilakukan agar sistem dapat mengunduh paket Docker versi stable secara langsung dari server resmi pengembang.

2. Pemasangan Paket Utama: Setelah repositori terdaftar, dilakukan instalasi paket docker-ce (Community Edition), docker-ce-cli, containerd.io, serta plugin docker-compose-plugin. Seluruh dependensi seperti libcgroup juga otomatis terpasang untuk mendukung fitur kontrol sumber daya yang akan diuji pada tahap selanjutnya.

3. Aktivasi Service: Pasca instalasi, layanan Docker daemon diaktifkan menggunakan perintah sudo systemctl enable --now docker. Perintah ini tidak hanya menjalankan Docker seketika, tetapi juga memastikan layanan tersebut aktif otomatis saat sistem Fedora melakukan proses booting.

4. Verifikasi Versi: Keberhasilan instalasi dipastikan dengan memeriksa versi Docker yang terpasang. Berdasarkan hasil pengecekan, sistem berhasil menjalankan Docker versi 29.0.1 (build eedd969), yang menandakan lingkungan siap digunakan untuk proses pengemasan aplikasi.

4.2 Struktur Projek dan Pengembangan Dockerfile

Setelah lingkungan host siap, tahap selanjutnya adalah mengorganisir berkas aplikasi ke dalam struktur yang rapi di dalam sebuah direktori utama bernama DOCKER. Struktur ini mencakup berkas logika utama app.py, daftar pustaka requirements.txt, konfigurasi docker-compose.yml, serta folder aset static dan templates

4.2.1 Analisis Baris Instruksi Dockerfile

Untuk mengemas aplikasi Flask menjadi sebuah Docker Image, penulis menyusun sebuah Dockerfile dengan instruksi sebagai berikut:

- FROM python:3.10: Instruksi ini menentukan base image resmi Python versi 3.10 yang sudah berisi semua perkakas dasar bahasa Python.
- WORKDIR /app: Menentukan ruang kerja di dalam kontainer. Hal ini dilakukan agar seluruh berkas aplikasi tersentralisasi di satu lokasi dan tidak bercampur dengan berkas sistem kontainer.
- COPY requirements.txt . & RUN pip install -r requirements.txt: Sebelum menyalin seluruh kode, penulis hanya menyalin berkas dependensi dan melakukan instalasi pustaka Flask serta MySQL-Connector. Strategi ini memanfaatkan mekanisme layer caching Docker sehingga proses build di masa depan akan lebih cepat jika tidak ada perubahan pada daftar pustaka.
- COPY . .: Setelah dependensi terpasang, barulah seluruh kode program disalin ke dalam image.
- CMD ["python", "app.py"]: Instruksi terakhir yang menetapkan perintah utama yang akan dieksekusi saat kontainer dinyalakan.

4.3 Implementasi Limitasi Resourrce

Pada tahap ini, penulis menggunakan Docker Compose untuk mengelola hubungan antara aplikasi Flask dan database MySQL secara simultan. Fokus utama pada bagian ini adalah penerapan batasan sumber daya fisik melalui konfigurasi yang didefinisikan secara deklaratif dalam berkas docker-compose.yml

4.3.1 Penerapan Mekanisme Cgroup

Penulis merancang dua layanan aplikasi dengan karakteristik yang berbeda untuk keperluan perbandingan:

1. Layanan app-unlimited: Layanan ini tidak diberikan batasan sumber daya sama sekali, sehingga diizinkan mengonsumsi seluruh kapasitas CPU dan memori host Fedora.

2. Layanan app-limited: Layanan ini dibatasi secara ketat menggunakan parameter di bawah blok deploy:

    - cpus: '0.5': Membatasi penggunaan prosesor maksimal hanya 50% dari satu core CPU.
    - memory: 128M: Membatasi penggunaan RAM maksimal sebesar 128 Megabytes.

Jika aplikasi pada kontainer terbatas ini mencoba menggunakan RAM lebih dari 128MB, maka mekanisme OOM (Out Of Memory) Killer pada kernel Linux akan mengintervensi untuk menjaga stabilitas sistem

4.4 Deployment dan Aktivasi Mekanisme Compability

Setelah seluruh berkas konfigurasi seperti Dockerfile, app.py, dan docker-compose.yml selesai disusun, tahap kritis berikutnya adalah melakukan deployment atau pengaktifan seluruh layanan. Pada tahap ini, penulis menggunakan instruksi spesifik untuk menjamin bahwa seluruh konfigurasi, terutama pembatasan sumber daya (resource limits), diterapkan secara akurat oleh Docker Engine.

4.4.1 Proses Rekonstruksi Image

Langkah pertama yang dilakukan penulis adalah membangun ulang image aplikasi menggunakan perintah: docker compose build --no-cache

Penggunaan parameter --no-cache dipilih dengan alasan teknis yang kuat. Secara default, Docker akan menggunakan lapisan (layers) yang sudah ada dari proses build sebelumnya untuk mempercepat waktu eksekusi. Namun, dalam lingkungan pengujian performa, penulis perlu memastikan bahwa:

- Integritas Environment: Seluruh dependensi (Flask, MySQL-connector) diunduh ulang dalam kondisi paling mutakhir.
- Eliminasi Error Tersembunyi: Memastikan tidak ada sisa-sisa konfigurasi lama yang tertinggal di dalam cache yang dapat memengaruhi hasil pengujian beban nantinya.
- Transparansi Proses: Penulis dapat mengamati setiap tahapan instalasi mulai dari penarikan base image Python 3.10 hingga eksekusi perintah pip install secara clean.

4.4.2 Aktivasi Layanan dengan Mode Kompatibilitas

Setelah image berhasil dibangun, penulis menjalankan seluruh layanan menggunakan perintah: docker compose --compatibility up -d

Perintah ini adalah bagian paling krusial dalam implementasi proyek ini. Penjelasan mengenai parameter yang digunakan adalah sebagai berikut:

1. --compatibility: Secara standar, Docker Compose V3 menganggap blok konfigurasi deploy (termasuk resources: limits) hanya berlaku untuk Docker Swarm (mode klaster). Karena penulis menjalankan pengujian pada satu mesin host (Fedora Linux), Docker Compose biasa seringkali mengabaikan batasan CPU dan Memori tersebut. Dengan menambahkan flag --compatibility, Docker dipaksa untuk menerjemahkan instruksi Swarm-style tersebut agar dapat diterapkan pada kontainer tunggal (non-swarm). Tanpa flag ini, limitasi 0.5 CPU yang dirancang tidak akan berjalan, dan kontainer akan tetap menggunakan seluruh sumber daya host.

2. up: Instruksi untuk menciptakan dan menyalakan kontainer berdasarkan layanan yang didefinisikan dalam docker-compose.yml (yaitu: db, app-unlimited, dan app-limited).

3. -d (Detached Mode): Menjalankan kontainer di latar belakang sehingga terminal Fedora tetap dapat digunakan untuk melakukan pemantauan (monitoring) tanpa terganggu oleh log aplikasi yang mengalir secara terus-menerus.

4.4.3 Verifikasi Status

Berkat konfigurasi depends_on dengan kondisi service_healthy pada berkas YAML, Docker akan memastikan urutan aktivasi yang benar. Layanan MySQL (db) akan diinisialisasi terlebih dahulu. Docker akan melakukan pengecekan kesehatan (healthcheck) secara berkala. Hanya setelah MySQL siap menerima koneksi, kontainer app-unlimited dan app-limited akan dinyalakan.

Penulis memverifikasi keberhasilan deployment ini dengan perintah docker ps. Hasil keluaran menunjukkan tiga kontainer berjalan dengan status "Up", di mana:

1. app-unlimited-container memetakan port internal 5000 ke port host 5000.

2. app-limited-container memetakan port internal 5000 ke port host 5001.

3. mysql-gudang berjalan secara internal untuk mendukung kedua aplikasi tersebut.

Dengan diterapkannya mode kompatibilitas ini, sistem kini berada dalam kondisi siap untuk dilakukan pengujian beban CPU guna membuktikan presisi dari batasan 0.5 core yang telah diaktivasi.

4.5 Logika Program Utama

Pada tahap implementasi kode, penulis menyusun berkas app.py menggunakan bahasa pemrograman Python dengan framework Flask. Kode ini dirancang tidak hanya untuk

menjalankan fungsi CRUD (Create, Read, Update, Delete), tetapi juga memiliki fitur ketahanan sistem (system resilience) dan instrumen pengujian.

1. Mekanisme Database Connection Retry: Dalam lingkungan kontainer, layanan aplikasi seringkali aktif lebih cepat daripada database MySQL. Oleh karena itu, penulis mengimplementasikan fungsi wait_for_db. Fungsi ini akan melakukan percobaan koneksi (retries) sebanyak 30 kali dengan jeda 2 detik. Hal ini sangat penting dalam implementasi microservices untuk mencegah aplikasi crash saat pertama kali dijalankan karena database belum siap menerima koneksi.

2. Inisialisasi Otomatis (init_db): Untuk mendukung prinsip stateless, aplikasi dirancang untuk melakukan pengecekan tabel barang secara otomatis setiap kali dijalankan. Jika tabel tidak ditemukan, aplikasi akan mengeksekusi perintah SQL CREATE TABLE. Ini memastikan bahwa skema basis data selalu sinkron dengan kode aplikasi tanpa perlu intervensi manual dari sisi admin database.

3. Algoritma Pengujian Beban CPU: Fungsi pengujian beban pada endpoint /load-cpu menggunakan perulangan while yang dikombinasikan dengan modul time.time(). Penulis sengaja tidak memberikan fungsi sleep di dalam perulangan tersebut agar prosesor dipaksa bekerja pada kapasitas maksimum selama 20 detik. Penggunaan fungsi matematika math.sqrt di dalam perulangan bertujuan untuk memberikan beban komputasi nyata pada unit logika aritmatika di CPU.

4.6 Analisis Konfigurasi

Berkas docker-compose.yml bertindak sebagai dirigen yang mengatur bagaimana seluruh kontainer berinteraksi. Terdapat beberapa parameter kunci yang diimplementasikan:

1. depends_on dengan condition: service_healthy: Penulis tidak hanya menyalakan kontainer secara bersamaan, tetapi menggunakan fitur healthcheck pada kontainer MySQL. Kontainer Flask hanya akan mulai dibangun dan dijalankan setelah MySQL melaporkan status "Healthy". Ini menjamin urutan eksekusi yang benar dalam arsitektur sistem.

2. Implementasi Volumes: Penulis menggunakan teknik bind mount pada direktori ./templates dan ./static. Hal ini diimplementasikan agar perubahan pada tampilan antarmuka (HTML/CSS) dapat langsung tercermin di dalam kontainer tanpa harus melakukan proses rebuild image dari awal, yang mana sangat mempercepat proses pengembangan (development workflow).

3. Pemisahan Environment Variables: Informasi sensitif seperti MYSQL_PASSWORD dan MYSQL_DB dipisahkan ke dalam blok environment. Dalam praktik industri, hal ini memudahkan proses migrasi antar lingkungan (misalnya dari Fedora lokal ke server

cloud) cukup dengan mengubah isi berkas konfigurasi tanpa menyentuh kode program utama.

4.7 Implementasi Monitoring Performa

Setelah seluruh layanan aktif, tahap implementasi terakhir adalah menyiapkan alat ukur. Penulis menggunakan dua metode monitoring:

1. Internal Monitoring: Menggunakan endpoint /load-cpu yang mengembalikan data jumlah iterasi sebagai representasi throughput aplikasi.
2. External Monitoring: Menggunakan perintah terminal docker stats. Alat ini diimplementasikan untuk memberikan bukti empiris mengenai penggunaan sumber daya dari sudut pandang sistem operasi Fedora. Melalui docker stats, penulis dapat memantau apakah penggunaan CPU pada app-limited-container benar-benar tertahan di angka dekat 50% (0.5 CPU) sesuai dengan limitasi Cgroups yang dipasang.

BAB 5

Hasil Pengujian & Analisis

5.1 Skenario Pengujian Performa CPU

Untuk mengukur efektivitas pembatasan sumber daya yang telah diimplementasikan, penulis melakukan skenario pengujian beban kerja intensif (stress test). Pengujian dilakukan dengan memicu endpoint /load-cpu pada kedua kontainer secara bergantian.

Parameter Pengujian:

- Durasi Pengujian: 20 Detik (konstan).
- Jenis Beban: Operasi aritmatika berat (perulangan matematika tanpa jeda).
- Metrik Ukur: Jumlah total iterasi yang berhasil diselesaikan dan persentase penggunaan CPU pada sistem host.

Penulis menggunakan dua lingkungan yang berbeda secara akses:

- Kontainer A (Port 5000): Tanpa batasan sumber daya (Unlimited).

- Kontainer B (Port 5001): Dibatasi sebesar 0.5 CPU melalui konfigurasi Docker Compose.

5.2 Hasil Pengujian Kuantitatif (jumlah iterasi)

Berdasarkan pengujian yang dilakukan melalui browser, didapatkan hasil yang sangat kontras antara kedua kontainer tersebut. Data jumlah iterasi yang berhasil diproses selama 20 detik adalah sebagai berikut:

Nama Kontainer	**Limitasi CPU**	**Total Iterasi (20 Detik)**
app-unlimited-container	Tidak Ada	**132.825**
app-limited-container	0.5 Core	**66.194**

Analisis Penuunan performa :

$$\text{Persentase Performa} = \frac{66.194}{132.825} \times 100\% \approx 49,83\%$$

Hasil perhitungan ini menunjukkan bahwa kontainer yang dibatasi hanya mampu menyelesaikan sekitar 49,83% pekerjaan dibandingkan dengan kontainer tanpa batas. Angka ini sangat akurat dan sinkron dengan limitasi 0.5 CPU (50%) yang dipasang pada berkas docker-compose.yml. Hal ini membuktikan bahwa mekanisme throttling pada kernel Linux bekerja dengan presisi tinggi dalam membatasi waktu eksekusi proses.

5.3 Analisis Monitoring Sumber daya

Selain melihat hasil dari sisi aplikasi (iterasi), penulis juga melakukan verifikasi dari sisi sistem operasi Fedora menggunakan perintah docker stats.

Berdasarkan pengamatan real-time saat pengujian berlangsung:

1 Kontainer app-unlimited: Terpantau menggunakan CPU hingga mencapai angka di atas 90-100% (tergantung ketersediaan core pada laptop host). Hal ini menyebabkan beban kerja prosesor meningkat drastis.

2 Kontainer app-limited: Terpantau sangat stabil di kisaran 49,50% hingga 49,90%. Meskipun aplikasi Flask berusaha melakukan komputasi seberat mungkin, angka penggunaan CPU tidak pernah menembus batas 50%.

Hal ini membuktikan bahwa fitur Cgroups v2 pada Fedora berhasil menahan penggunaan sumber daya agar tidak melebihi kuota yang telah ditentukan, sehingga mencegah fenomena noisy neighbor di mana satu aplikasi menghabiskan seluruh sumber daya server.

BAB 6

Penutup

6.1 Kesimpulan

Berdasarkan seluruh rangkaian proses implementasi, pengujian, dan analisis yang telah dilakukan pada sistem virtualisasi Docker di atas lingkungan Fedora Linux, maka penulis dapat menarik beberapa kesimpulan utama sebagai berikut:

1 Efektivitas Limitasi Resource: Mekanisme pembatasan sumber daya menggunakan Control Groups (Cgroups) pada Docker terbukti bekerja dengan tingkat presisi yang sangat tinggi. Hal ini dibuktikan melalui pengujian beban CPU, di mana kontainer yang dibatasi sebesar 0.5 CPU hanya mampu menyelesaikan 66.194 iterasi, atau sekitar 49,83% dari performa kontainer tanpa batas (132.825 iterasi). Hasil ini menunjukkan sinkronisasi yang hampir sempurna dengan target pembatasan 50% yang ditetapkan.

2 Stabilitas Arsitektur Microservices: Penggunaan Docker Compose mempermudah orkestrasi antara layanan Flask dan MySQL. Fitur healthcheck dan dependency management menjamin bahwa aplikasi tidak akan mengalami kegagalan koneksi (connection refused) saat pertama kali dijalankan, karena sistem mampu memastikan database telah siap sebelum aplikasi aktif.

3 Peran Mode Kompatibilitas: Penggunaan parameter --compatibility pada saat aktivasi layanan terbukti krusial dalam lingkungan pengujian non-swarm. Tanpa parameter ini, instruksi limitasi pada berkas YAML versi 3 seringkali diabaikan oleh Docker Engine pada host tunggal.

4 Isolasi dan Keamanan Sistem: Pengujian membuktikan bahwa meskipun sebuah kontainer dipaksa bekerja hingga kapasitas maksimumnya (100% dari jatah CPU-nya), sistem host Fedora tetap stabil dan kontainer lain (seperti MySQL) tidak terganggu performanya. Ini membuktikan keberhasilan isolasi Namespaces dan Cgroups dalam mencegah fenomena noisy neighbor.

6.2 Saran

Meskipun sistem yang dirancang telah memenuhi tujuan awal penelitian, penulis menyadari masih terdapat ruang untuk pengembangan lebih lanjut. Berikut adalah beberapa saran yang dapat diajukan:

1 Eksplorasi Limitasi I/O dan Jaringan: Penelitian selanjutnya disarankan untuk tidak hanya berfokus pada CPU dan Memori, tetapi juga menguji batasan Block I/O (kecepatan baca/tulis disk) serta Network Bandwidth untuk melihat pengaruhnya terhadap performa database berskala besar.

2 Implementasi Monitoring Lanjutan: Untuk visualisasi yang lebih profesional dalam skala produksi, penggunaan alat monitoring seperti Prometheus dan Grafana dapat dipertimbangkan guna menggantikan perintah manual docker stats.

3 Pengujian pada Skala Klaster: Pengembangan penelitian dapat ditingkatkan dengan menggunakan Docker Swarm atau Kubernetes untuk melihat bagaimana batasan sumber daya ini dikelola ketika kontainer berpindah antar-node dalam sebuah jaringan server yang luas.

4 Optimasi Dockerfile: Disarankan untuk mencoba penggunaan multi-stage build pada Dockerfile guna menghasilkan ukuran image yang lebih kecil dan lebih aman dengan memisahkan lingkungan kompilasi dan lingkungan eksekusi.

Daftar Pustaka

Docker Documentation. (2024). Runtime metrics with Resource Control Groups (cgroups). Diambil dari https://docs.docker.com/config/containers/runmetrics/

Fedora Project. (2024). Changes in Fedora: Control Groups v2. Diambil dari https://fedoraproject.org/wiki/Changes/CGroupsV2

Grinberg, M. (2018). Flask Web Development: Developing Web Applications with Python. Sebastopol: O'Reilly Media.

Kane, S. P., & Matthias, K. (2023). Docker: Up & Running: Shipping Reliable Containers in Production. Sebastopol: O'Reilly Media.

Kerrisk, M. (2010). The Linux Programming Interface: A Linux and UNIX System Programming Handbook. San Francisco: No Starch Press. (Referensi teknis untuk Namespaces dan Cgroups).

Silberschatz, A., Galvin, P. B., & Gagne, G. (2018). Operating System Concepts (10th ed.). Hoboken: Wiley. (Referensi teori untuk manajemen CPU dan Memori).

Lampiran

app.py

```
from flask import Flask, jsonify, request, render_template

import mysql.connector

import os

import time

import math

from flask_cors import CORS
```

```
app = Flask(__name__)
CORS(app)

def db():
    return mysql.connector.connect(
        host=os.getenv("MYSQL_HOST", "db"),
        user=os.getenv("MYSQL_USER", "root"),
        password=os.getenv("MYSQL_PASSWORD", "example"),
        database=os.getenv("MYSQL_DB", "gudang"),
        port=3306
    )

# --- LOGIKA BARU: PEKERJAAN TETAP, WAKTU BERVARIASI ---
def cpu_heavy_task(target_iterations=10000000):
    start_time = time.time()
    for i in range(target_iterations):
        # Operasi matematika untuk membebani CPU
        _ = math.sqrt(i) * math.log(i + 1)
    return time.time() - start_time

@app.route("/load-cpu")
def load_cpu():
    # Kita beri beban 10 juta iterasi untuk kedua kontainer
    target = 10000000
    time_taken = cpu_heavy_task(target_iterations=target)
```

```
    return jsonify({
        "status": "CPU Load Test Complete",
        "total_work": f"{target:,} operations",
        "time_taken_seconds": round(time_taken, 2),
        "container_limit": "Check docker stats to verify 0.5 CPU
limit"
    })

# --- SISANYA TETAP SAMA ---
@app.route("/")
def index():
    return render_template("index.html")

@app.route("/api/barang", methods=["GET"])
def get_barang():
    try:
        connection = db()
        cursor = connection.cursor(dictionary=True)
        cursor.execute("SELECT * FROM barang")
        data = cursor.fetchall()
        cursor.close()
        connection.close()
        return jsonify(data)
    except Exception as e:
        return jsonify({"error": str(e)}), 500
```

```
if __name__ == "__main__":
    app.run(host="0.0.0.0", port=5000, debug=True)
```

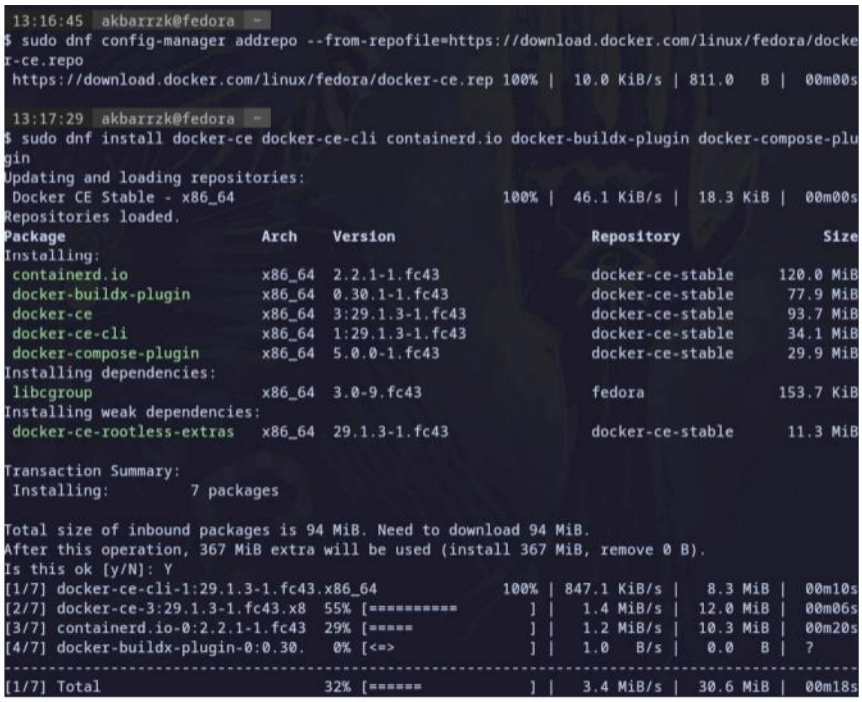
docker-compose.yml

```
services:
  # CONTAINER 1: TANPA BATASAN (UNLIMITED)
  app-unlimited:
    build: .
    container_name: app-unlimited-container
    ports:
      - "5000:5000" # Diakses di localhost:5000
    environment:
      MYSQL_HOST: db
      MYSQL_USER: root
      MYSQL_PASSWORD: example
      MYSQL_DB: gudang
    depends_on:
      db:
        condition: service_healthy
    volumes:
      - ./templates:/app/templates
      - ./static:/app/static
    # TIDAK ADA LIMIT - Menggunakan seluruh resource host
```

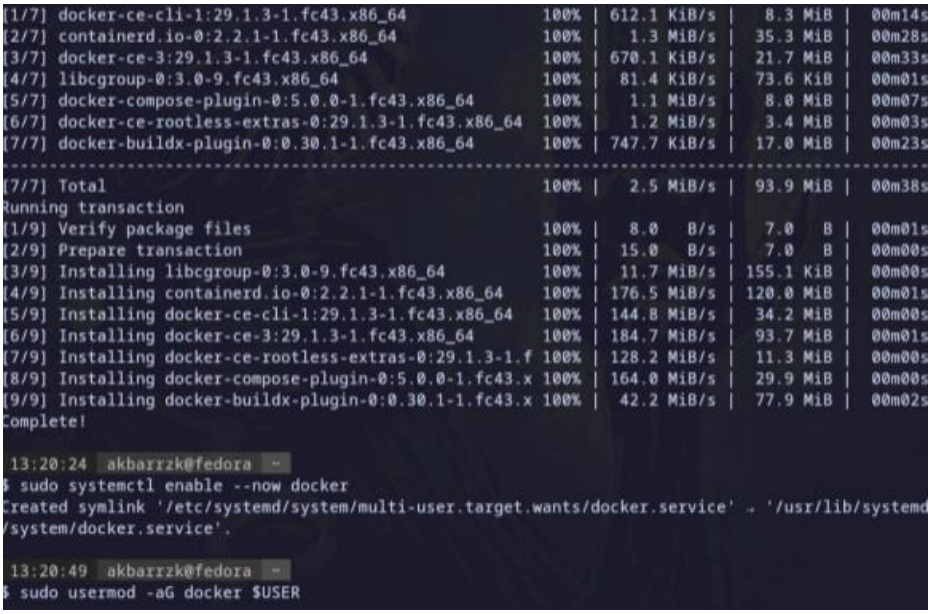
```
# CONTAINER 2: DENGAN BATASAN (LIMITED - CGROUPS)
app-limited:
  build: .
  container_name: app-limited-container
  ports:
    - "5001:5000" # Diakses di localhost:5001
  environment:
    MYSQL_HOST: db
    MYSQL_USER: root
    MYSQL_PASSWORD: example
    MYSQL_DB: gudang
  depends_on:
    db:
      condition: service_healthy
  volumes:
    - ./templates:/app/templates
    - ./static:/app/static

  # PENERAPAN RESOURCE LIMITS (CGROUPS) - FORMAT YANG BENAR
  cpus: 0.5              # Batasi CPU hingga 50% dari 1 core
  mem_limit: 128m        # Batasi Memori hingga 128 MB
  mem_reservation: 64m   # Soft limit memori

# DATABASE
```

```
  db:
    image: mysql:8
    container_name: mysql-gudang
    environment:
      MYSQL_ROOT_PASSWORD: example
      MYSQL_DATABASE: gudang
    ports:
      - "3306:3306"
    volumes:
      - mysql_data:/var/lib/mysql
    healthcheck:
      test: ["CMD", "mysqladmin", "ping", "-h", "localhost"]
      interval: 5s
      timeout: 3s
      retries: 10

volumes:
  mysql_data:
```

Foto

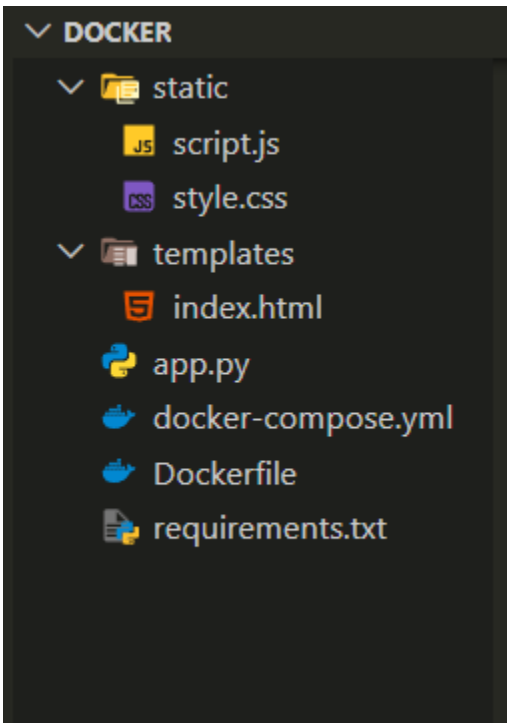


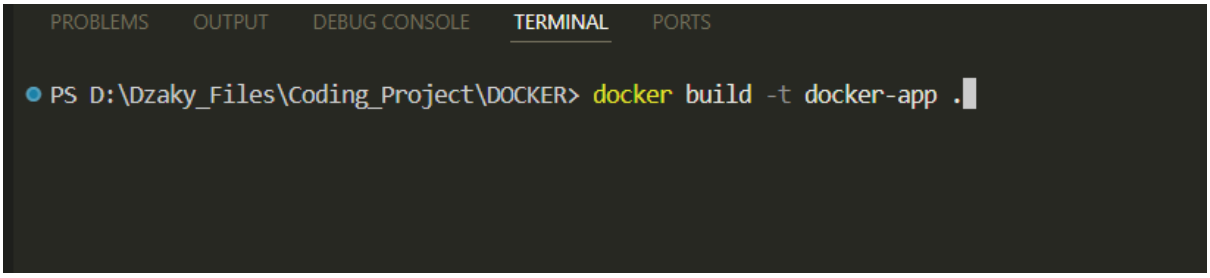
```
 13:16:45  akbarrzk@fedora  ~
$ sudo dnf config-manager addrepo --from-repofile=https://download.docker.com/linux/fedora/docke
r-ce.repo
 https://download.docker.com/linux/fedora/docker-ce.rep 100% |  10.0 KiB/s | 811.0   B |  00m00s

 13:17:29  akbarrzk@fedora  ~
$ sudo dnf install docker-ce docker-ce-cli containerd.io docker-buildx-plugin docker-compose-plu
gin
Updating and loading repositories:
 Docker CE Stable - x86_64                       100% |  46.1 KiB/s |  18.3 KiB |  00m00s
Repositories loaded.
Package                     Arch    Version                     Repository               Size
Installing:
 containerd.io              x86_64  2.2.1-1.fc43                docker-ce-stable     120.0 MiB
 docker-buildx-plugin       x86_64  0.30.1-1.fc43               docker-ce-stable      77.9 MiB
 docker-ce                  x86_64  3:29.1.3-1.fc43             docker-ce-stable      93.7 MiB
 docker-ce-cli              x86_64  1:29.1.3-1.fc43             docker-ce-stable      34.1 MiB
 docker-compose-plugin      x86_64  5.0.0-1.fc43                docker-ce-stable      29.9 MiB
Installing dependencies:
 libcgroup                  x86_64  3.0-9.fc43                  fedora               153.7 KiB
Installing weak dependencies:
 docker-ce-rootless-extras  x86_64  29.1.3-1.fc43               docker-ce-stable      11.3 MiB

Transaction Summary:
 Installing:         7 packages

Total size of inbound packages is 94 MiB. Need to download 94 MiB.
After this operation, 367 MiB extra will be used (install 367 MiB, remove 0 B).
Is this ok [y/N]: Y
[1/7] docker-ce-cli-1:29.1.3-1.fc43.x86_64             100% | 847.1 KiB/s |   8.3 MiB |  00m10s
[2/7] docker-ce-3:29.1.3-1.fc43.x8  55% [==========        ] |   1.4 MiB/s |  12.0 MiB |  00m06s
[3/7] containerd.io-0:2.2.1-1.fc43  29% [=====             ] |   1.2 MiB/s |  10.3 MiB |  00m20s
[4/7] docker-buildx-plugin-0:0.30.   0% [<=>               ] |   1.0   B/s |   0.0   B |  ?
--------------------------------------------------------------------------------------------
[1/7] Total                         32% [======            ] |   3.4 MiB/s |  30.6 MiB |  00m18s
```

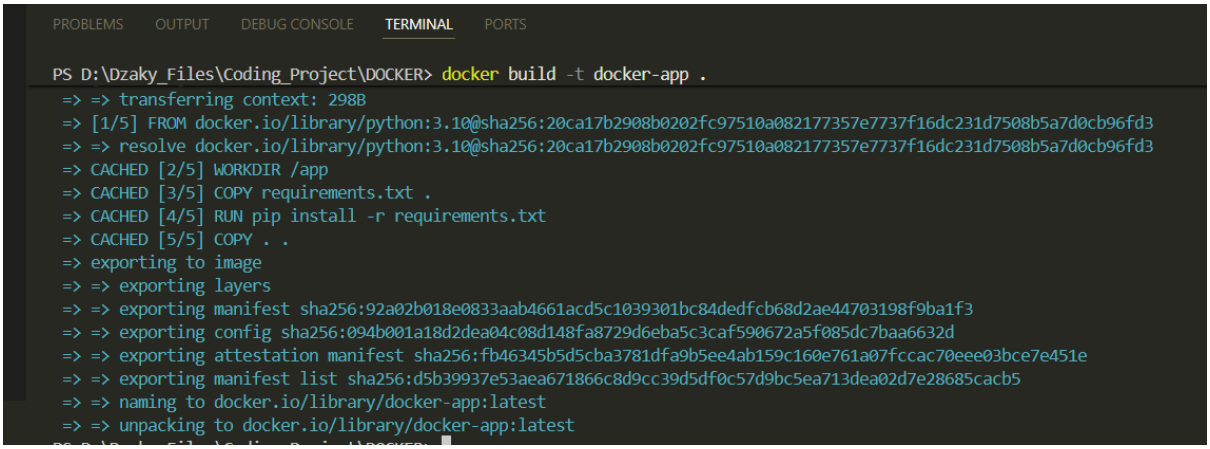


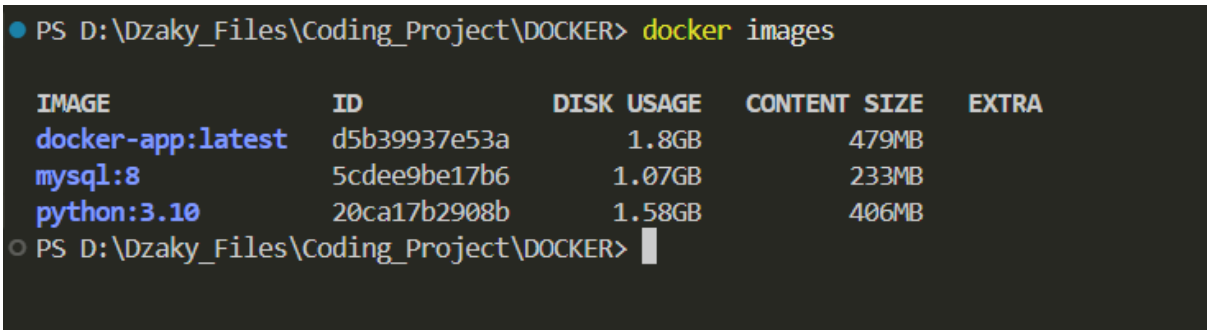
```
[1/7] docker-ce-cli-1:29.1.3-1.fc43.x86_64              100% | 612.1 KiB/s |   8.3 MiB |  00m14s
[2/7] containerd.io-0:2.2.1-1.fc43.x86_64               100% |   1.3 MiB/s |  35.3 MiB |  00m28s
[3/7] docker-ce-3:29.1.3-1.fc43.x86_64                  100% | 670.1 KiB/s |  21.7 MiB |  00m33s
[4/7] libcgroup-0:3.0-9.fc43.x86_64                     100% |  81.4 KiB/s |  73.6 KiB |  00m01s
[5/7] docker-compose-plugin-0:5.0.0-1.fc43.x86_64       100% |   1.1 MiB/s |   8.0 MiB |  00m07s
[6/7] docker-ce-rootless-extras-0:29.1.3-1.fc43.x86_64  100% |   1.2 MiB/s |   3.4 MiB |  00m03s
[7/7] docker-buildx-plugin-0:0.30.1-1.fc43.x86_64       100% | 747.7 KiB/s |  17.0 MiB |  00m23s
--------------------------------------------------------------------------------------------
[7/7] Total                                             100% |   2.5 MiB/s |  93.9 MiB |  00m38s
Running transaction
[1/9] Verify package files                              100% |   8.0   B/s |   7.0   B |  00m01s
[2/9] Prepare transaction                               100% |  15.0   B/s |   7.0   B |  00m00s
[3/9] Installing libcgroup-0:3.0-9.fc43.x86_64          100% |  11.7 MiB/s | 155.1 KiB |  00m00s
[4/9] Installing containerd.io-0:2.2.1-1.fc43.x86_64    100% | 176.5 MiB/s | 120.0 MiB |  00m01s
[5/9] Installing docker-ce-cli-1:29.1.3-1.fc43.x86_64   100% | 144.8 MiB/s |  34.2 MiB |  00m00s
[6/9] Installing docker-ce-3:29.1.3-1.fc43.x86_64       100% | 184.7 MiB/s |  93.7 MiB |  00m01s
[7/9] Installing docker-ce-rootless-extras-0:29.1.3-1.f 100% | 128.2 MiB/s |  11.3 MiB |  00m00s
[8/9] Installing docker-compose-plugin-0:5.0.0-1.fc43.x 100% | 164.0 MiB/s |  29.9 MiB |  00m00s
[9/9] Installing docker-buildx-plugin-0:0.30.1-1.fc43.x 100% |  42.2 MiB/s |  77.9 MiB |  00m02s
Complete!

 13:20:24  akbarrzk@fedora  ~
$ sudo systemctl enable --now docker
Created symlink '/etc/systemd/system/multi-user.target.wants/docker.service' → '/usr/lib/systemd
/system/docker.service'.

 13:20:49  akbarrzk@fedora  ~
$ sudo usermod -aG docker $USER
```

```
13:20:59  akbarrzk@fedora  ~
$ docker run hello-world
permission denied while trying to connect to the docker API at unix:///var/run/docker.sock

13:21:04  akbarrzk@fedora  ~
$ newgrp docker

13:21:52  akbarrzk@fedora  ~
$ docker run hello-world
Unable to find image 'hello-world:latest' locally
latest: Pulling from library/hello-world
17eec7bbc9d7: Pull complete
ea52d2000f90: Download complete
Digest: sha256:d4aaab6242e0cace87e2ec17a2ed3d779d18fbfd03042ea58f2995626396a274
Status: Downloaded newer image for hello-world:latest

Hello from Docker!
This message shows that your installation appears to be working correctly.

To generate this message, Docker took the following steps:
 1. The Docker client contacted the Docker daemon.
 2. The Docker daemon pulled the "hello-world" image from the Docker Hub.
    (amd64)
 3. The Docker daemon created a new container from that image which runs the
    executable that produces the output you are currently reading.
 4. The Docker daemon streamed that output to the Docker client, which sent it
    to your terminal.

To try something more ambitious, you can run an Ubuntu container with:
 $ docker run -it ubuntu bash

Share images, automate workflows, and more with a free Docker ID:
 https://hub.docker.com/

For more examples and ideas, visit:
 https://docs.docker.com/get-started/


13:22:02  akbarrzk@fedora  ~
$
```

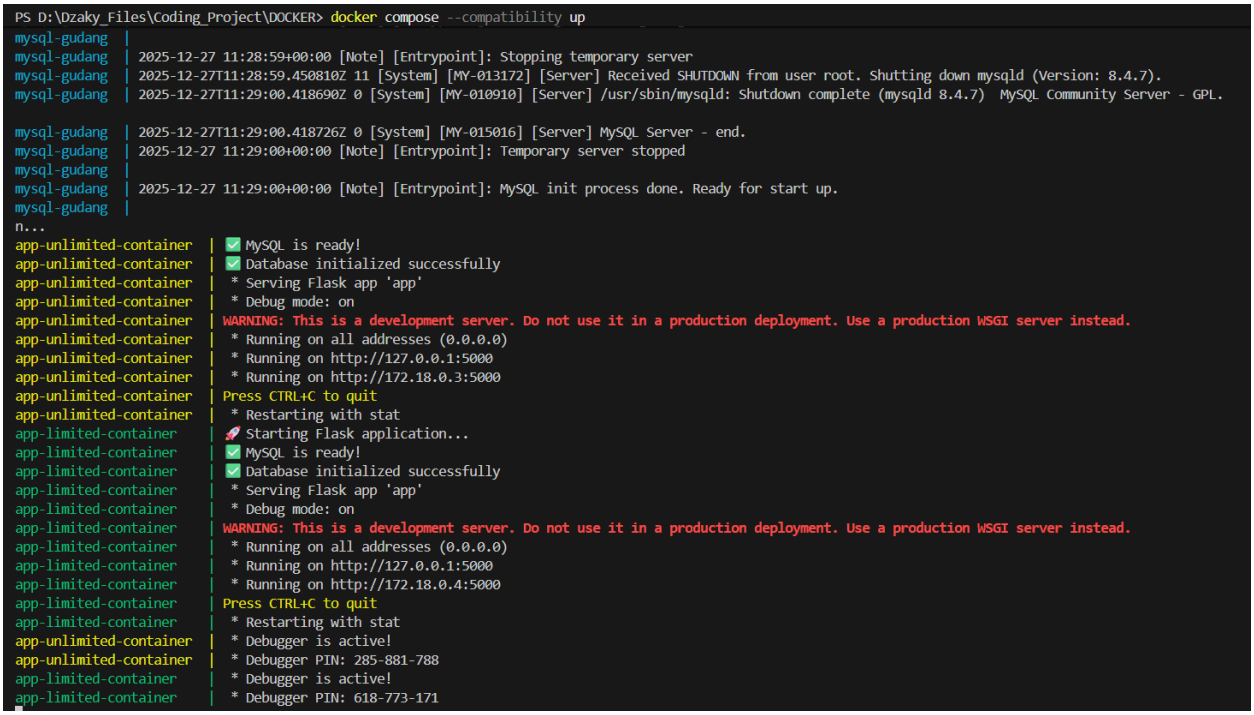


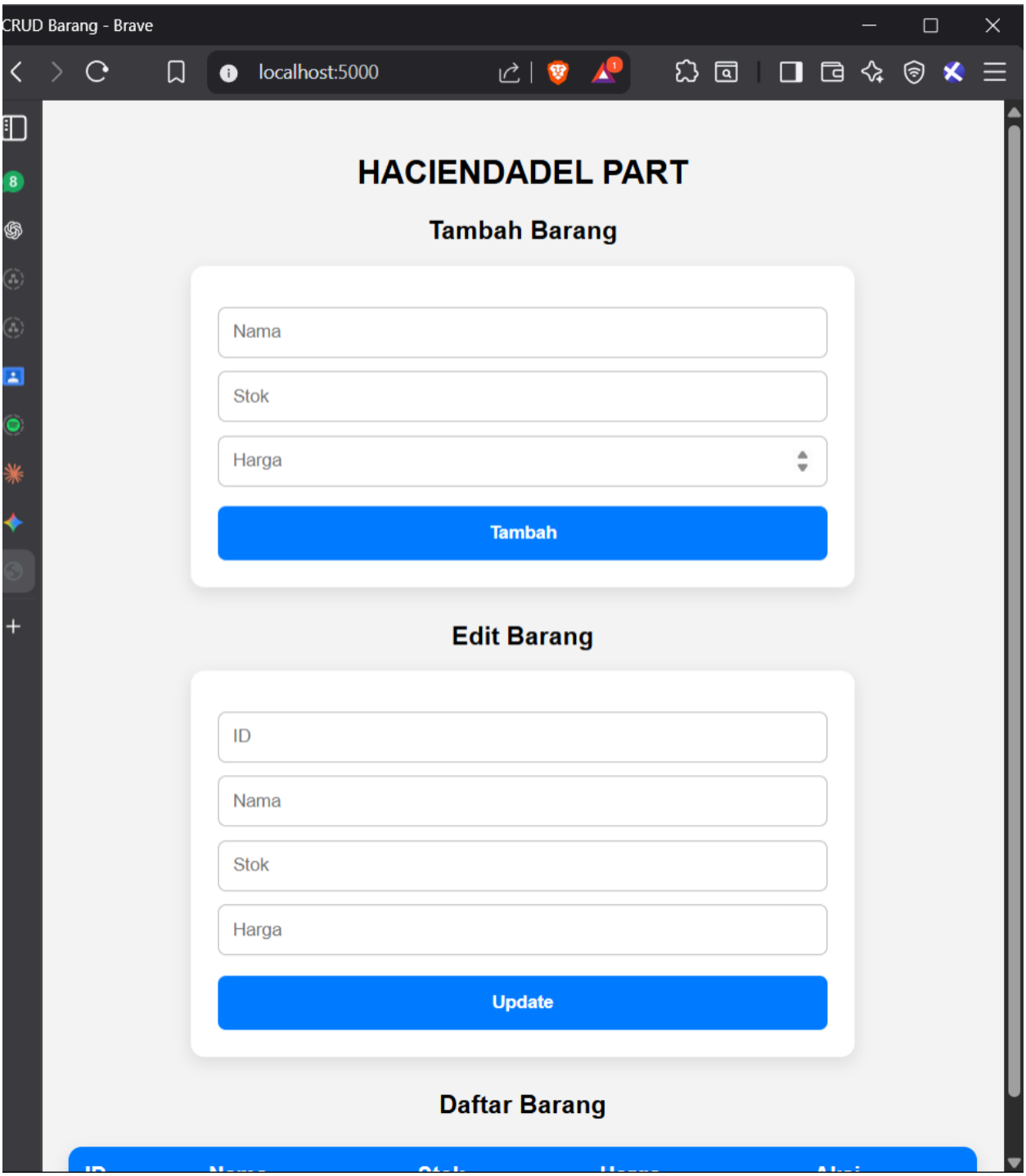

PROBLEMS
OUTPUT
DEBUG CONSOLE
TERMINAL
PORTS
PS D:\Dzaky_Files\Coding_Project\DOCKER> docker build -t docker-app .


PROBLEMS
OUTPUT
DEBUG CONSOLE
TERMINAL
PORTS
PS D:\Dzaky_Files\Coding_Project\DOCKER> docker build -t docker-app .
=> => transferring context: 298B
=> [1/5] FROM docker.io/library/python:3.10@sha256:20ca17b2908b0202fc97510a082177357e7737f16dc231d7508b5a7d0cb96fd3
=> => resolve docker.io/library/python:3.10@sha256:20ca17b2908b0202fc97510a082177357e7737f16dc231d7508b5a7d0cb96fd3
=> CACHED [2/5] WORKDIR /app
=> CACHED [3/5] COPY requirements.txt .
=> CACHED [4/5] RUN pip install -r requirements.txt
=> CACHED [5/5] COPY . .
=> exporting to image
=> => exporting layers
=> => exporting manifest sha256:92a02b018e0833aab4661acd5c1039301bc84dedfcb68d2ae44703198f9ba1f3
=> => exporting config sha256:094b001a18d2dea04c08d148fa8729d6eba5c3caf590672a5f085dc7baa6632d
=> => exporting attestation manifest sha256:fb46345b5d5cba3781dfa9b5ee4ab159c160e761a07fccac70eee03bce7e451e
=> => exporting manifest list sha256:d5b39937e53aea671866c8d9cc39d5df0c57d9bc5ea713dea02d7e28685cacb5
=> => naming to docker.io/library/docker-app:latest
=> => unpacking to docker.io/library/docker-app:latest

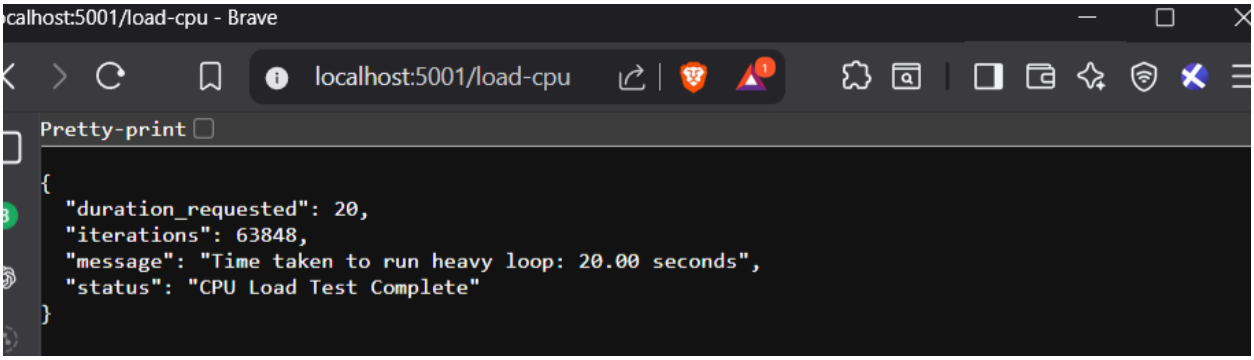

PS D:\Dzaky_Files\Coding_Project\DOCKER> docker images
IMAGE ID DISK USAGE CONTENT SIZE EXTRA
docker-app:latest d5b39937e53a 1.8GB 479MB
mysql:8 5cdee9be17b6 1.07GB 233MB
python:3.10 20ca17b2908b 1.58GB 406MB
PS D:\Dzaky_Files\Coding_Project\DOCKER>

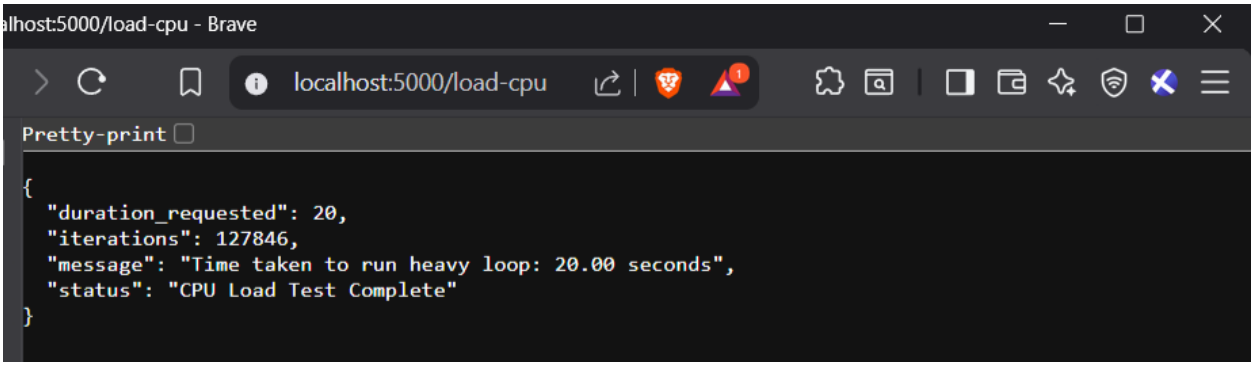
```
PS D:\Dzaky_Files\Coding_Project\DOCKER> docker compose build --no-cache
time="2025-12-27T18:27:24+07:00" level=warning msg="D:\\Dzaky_Files\\Coding_Project\\DOCKER\\docker-com
ignored, please remove it to avoid potential confusion"
[+] Building 33.7s (16/16) FINISHED
 => [internal] load local bake definitions          0.0s
 => => reading from stdin 1.07kB                    0.0s
 => [app-limited internal] load build definition    0.1s
 => => transferring dockerfile: 177B                0.0s
 => [app-unlimited internal] load metadata for d    4.5s
 => [auth] library/python:pull token for registr    0.0s
 => [app-unlimited internal] load .dockerignore     0.0s
 => => transferring context: 2B                     0.0s
 => [app-unlimited 1/5] FROM docker.io/library/p    1.9s
 => => resolve docker.io/library/python:3.10@sha    1.9s
 => [app-limited internal] load build context       0.0s
 => => transferring context: 298B                   0.0s
 => [auth] library/python:pull token for registr    0.0s
 => CACHED [app-unlimited 2/5] WORKDIR /app         0.0s
 => [app-unlimited 3/5] COPY requirements.txt .     0.1s
 => [app-unlimited 4/5] RUN pip install -r requ    19.2s
 => [app-limited 5/5] COPY . .                      0.1s
 => [app-limited] exporting to image                6.5s
 => => exporting layers                             5.1s
 => => exporting manifest sha256:8bfb71d967ca7ab    0.0s
 => => exporting config sha256:d3db093c5af897e11    0.0s
 => => exporting attestation manifest sha256:5c4    0.1s
 => => exporting manifest list sha256:f4d2cbafa5    0.0s
 => => naming to docker.io/library/docker-app-li    0.0s
 => => unpacking to docker.io/library/docker-app    1.2s
 => [app-unlimited] exporting to image              6.5s
 => => exporting layers                             5.1s
 => => exporting manifest sha256:825eebade096805    0.1s
 => => exporting config sha256:1d1555e6e5891f628    0.0s
 => => exporting attestation manifest sha256:56e    0.1s
 => => exporting manifest list sha256:6cd7eb9d77    0.0s
 => => naming to docker.io/library/docker-app-un    0.0s
 => => unpacking to docker.io/library/docker-app    1.2s
 => [app-unlimited] resolving provenance for met    0.0s
 => [app-limited] resolving provenance for metad    0.0s
[+] Building 2/2
 ✓ docker-app-limited    Built                      0.0s
 ✓ docker-app-unlimited  Built                      0.0s
```


PS D:\Dzaky_Files\Coding_Project\DOCKER> docker compose --compatibility up
mysql-gudang | 2025-12-27 11:28:59+00:00 [Note] [Entrypoint]: Stopping temporary server
mysql-gudang | 2025-12-27T11:28:59.450810Z 11 [System] [MY-013172] [Server] Received SHUTDOWN from user root. Shutting down mysqld (Version: 8.4.7).
mysql-gudang | 2025-12-27T11:29:00.418690Z 0 [System] [MY-010910] [Server] /usr/sbin/mysqld: Shutdown complete (mysqld 8.4.7) MySQL Community Server - GPL.
mysql-gudang | 2025-12-27T11:29:00.418726Z 0 [System] [MY-015016] [Server] MySQL Server - end.
mysql-gudang | 2025-12-27 11:29:00+00:00 [Note] [Entrypoint]: Temporary server stopped
mysql-gudang | 2025-12-27 11:29:00+00:00 [Note] [Entrypoint]: MySQL init process done. Ready for start up.
n...
app-unlimited-container | ✅ MySQL is ready!
app-unlimited-container | ✅ Database initialized successfully
app-unlimited-container | * Serving Flask app 'app'
app-unlimited-container | * Debug mode: on
app-unlimited-container | WARNING: This is a development server. Do not use it in a production deployment. Use a production WSGI server instead.
app-unlimited-container | * Running on all addresses (0.0.0.0)
app-unlimited-container | * Running on http://127.0.0.1:5000
app-unlimited-container | * Running on http://172.18.0.3:5000
app-unlimited-container | Press CTRL+C to quit
app-unlimited-container | * Restarting with stat
app-limited-container | 🚀 Starting Flask application...
app-limited-container | ✅ MySQL is ready!
app-limited-container | ✅ Database initialized successfully
app-limited-container | * Serving Flask app 'app'
app-limited-container | * Debug mode: on
app-limited-container | WARNING: This is a development server. Do not use it in a production deployment. Use a production WSGI server instead.
app-limited-container | * Running on all addresses (0.0.0.0)
app-limited-container | * Running on http://127.0.0.1:5000
app-limited-container | * Running on http://172.18.0.4:5000
app-limited-container | Press CTRL+C to quit
app-limited-container | * Restarting with stat
app-unlimited-container | * Debugger is active!
app-unlimited-container | * Debugger PIN: 285-881-788
app-limited-container | * Debugger is active!
app-limited-container | * Debugger PIN: 618-773-171




```
PS D:\Dzaky_Files\Coding_Project\DOCKER> docker ps -a
CONTAINER ID   IMAGE                  COMMAND                  CREATED         STATUS                   PORTS                                       NAMES
a41f1320cd59   docker_app-unlimited   "python app.py"          2 minutes ago   Up About a minute        0.0.0.0:5000->5000/tcp, [::]:5000->5000/tcp   app-unlimited-container
355f7d615676   docker_app-limited     "python app.py"          2 minutes ago   Up About a minute        0.0.0.0:5001->5000/tcp, [::]:5001->5000/tcp   app-limited-container
704d1390e30a   mysql:8                "docker-entrypoint.s…"   2 minutes ago   Up 2 minutes (healthy)   0.0.0.0:3306->3306/tcp, [::]:3306->3306/tcp   mysql-gudang
PS D:\Dzaky_Files\Coding_Project\DOCKER>
```





CONTAINER ID	NAME	CPU %	MEM USAGE / LIMIT	MEM %	NET I/O	BLOCK I/O	PIDS
a41f1320cd59	app-unlimited-container	104.53%	56.6MiB / 3.488GiB	1.58%	28.6kB / 23.7kB	11.6MB / 295kB	4
355f7d615676	app-limited-container	50.29%	48.61MiB / 128MiB	37.98%	19.7kB / 16.3kB	397kB / 295kB	4
704d1390e30a	mysql-gudang	0.46%	484.6MiB / 3.488GiB	13.57%	36.4kB / 37.7kB	140MB / 285MB	38