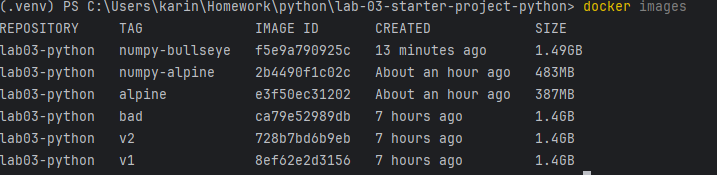
**Python-застосунок**

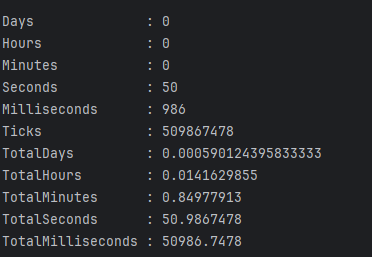
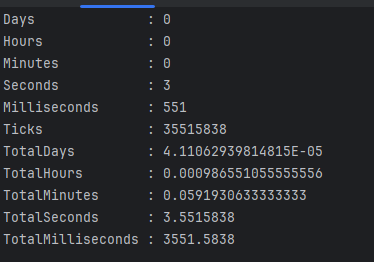
У цій частині лабораторної роботи було виконано серію експериментів з кодом:

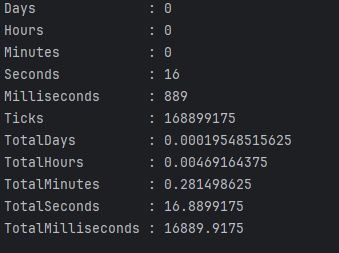
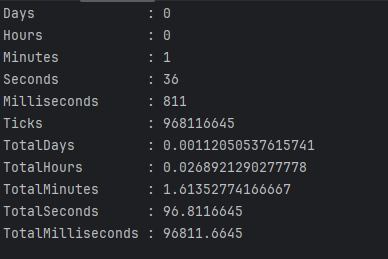
* -Збірка базового образу FastAPI-застосунку (lab03-python:v1) на основі python:3.10-bullseye
* -Повторна збірка після незначної зміни в коді (lab03-python:v2)
* -Збірка з “поганим” Dockerfile (lab03-python:bad) - спочатку копіюється код, а потім ставляться залежності
* -Збірка легкого образу на основі python:3.10-alpine без numpy
* -Додавання numpy + реалізація ендпоінту /api/matrices для перемноження матриць
* -Порівняння numpy-збірок на Alpine і Bullseye

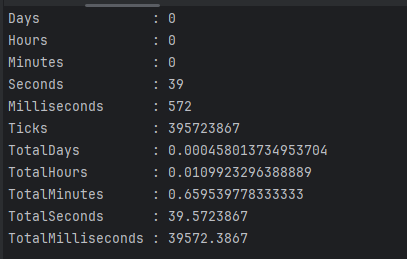
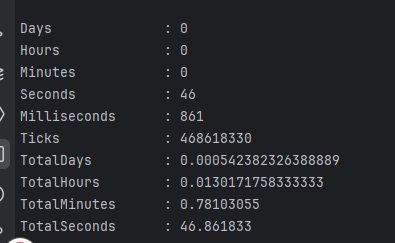
**Порівняльна таблиця**

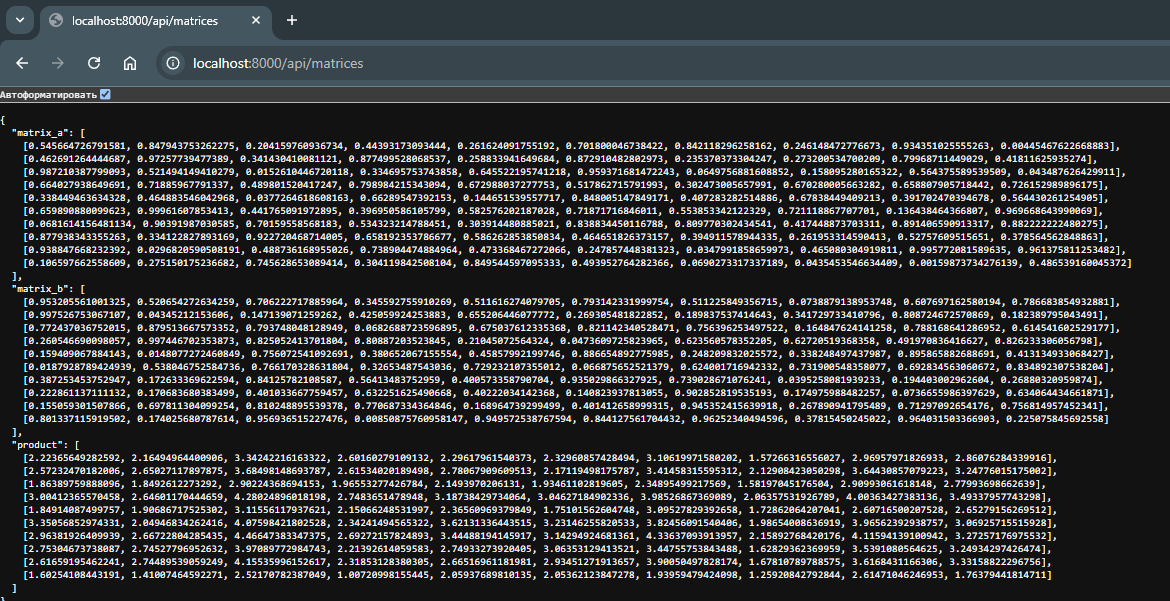
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Образ | Базовий образ | Час збірки | Розмір | Примітки |
| lab03-python:v1 | python:3.10-bullseye | 50.99 с | 1.4 GB | перша повна збірка без кешу |
| lab03-python:v2 | python:3.10-bullseye | 3.55 с | 1.4 GB | лише зміни в коді, кеш використано |
| lab03-python:bad | python:3.10-bullseye | 16.89 с | 1.4 GB | "поганий" Dockerfile, втрачено кеш для pip install |
| lab03-python:alpine | python:3.10-alpine | 96.81 с | 387 MB | без numpy, довга збірка через build tools |
| lab03-python:numpy-alpine | python:3.10-alpine | 39.57 с | 483 MB | з numpy, збірка коротша, бо кеш працює |
| lab03-python:numpy-bullseye | python:3.10-bullseye | 46.86 с | 1.49 GB | стабільна збірка з numpy і /api/matrices |

****

** **

** **

** **



**Особливості та труднощі**

* **Файл** requirements/backend.in мав нестандартне розташування, тому Dockerfile довелось адаптувати
* -**Docker не знаходив** app у main.py, бо забула експортувати об'єкт FastAPI
* У **Alpine** збірка numpy займала багато часу через необхідність компіляції і потребувала встановлення build tools
* PyCharm **не бачив імпорти**, поки не додала кореневу папку як Sources Root і не встановила залежності локально

**Висновки**

* -Порядок інструкцій у Dockerfile критично впливає на кешування. Краще спочатку копіювати requirements.txt, а лише потім код - це дозволяє зберегти шари кешу при зміні лише коду.
* -Bullseye простіший у використанні, хоч і важчий - але встановлює залежності швидше.
* -Кеш дає величезний виграш у швидкості, якщо правильно використаний.

**Golang-застосунок**

У цій частині лабораторної роботи було зроблено:

-Побудова повного образу на основі debian:bullseye-slim

-Багатоетапна збірка з подальшим створенням образів:

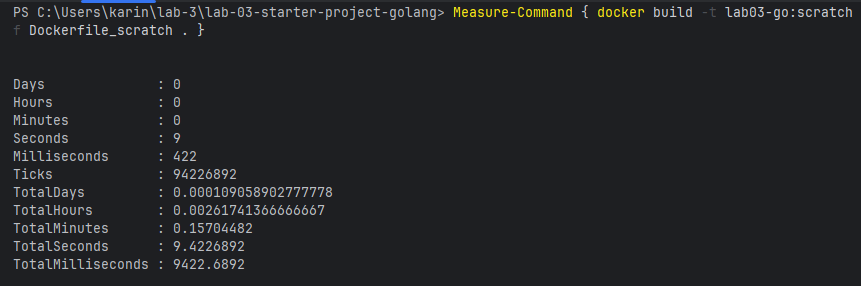
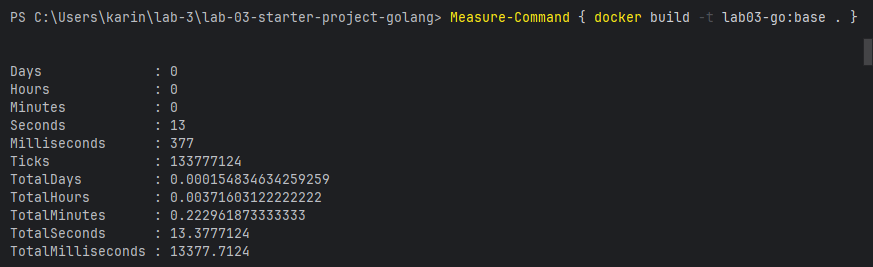
на основі scratch

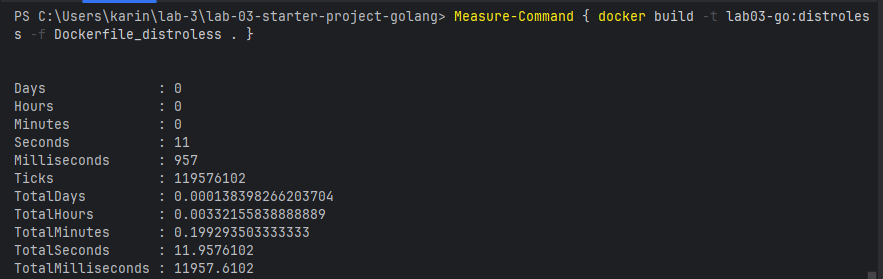
на основі distroless/static

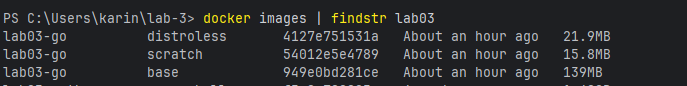
На кожному етапі було виміряно час збірки, розмір результату, а також проаналізовано переваги/недоліки способу.

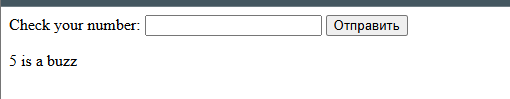
**Порівняльна таблиця**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Образ | Базовий образ | Час збірки | Розмір | Shell | Зручність налагодження |
| lab03-go:base | debian:bullseye-slim | 13.4 c | 139 МB | Так | Так |
| lab03-go:scratch | Scratch | 9.4 c | 15.8 MB | Ні | Ні |
| lab03-go:distroless | distroless/static | 11.9 c | 21.9 MB | Ні | Ні |









**Особливості та труднощі**

* У lab03-go:base виникла помилка GLIBC\_2.34 not found - вирішено статичною збіркою (CGO\_ENABLED=0).
* Не працював веб-інтерфейс - не вистачало templates/index.html. Додано COPY templates ./templates.
* Образ scratch не має shell - не можна увійти в контейнер для налагодження.
* distroless також без shell, але краще підходить для продакшену.
* Усі мінімальні образи вимагають статичної Go-збірки.

**Висновки**

* lab03-go:base зручний для розробки та тестування. Має shell, але є найважчим за розміром.
* **l**ab03-go:scratch - найбільш мінімальний, займає лише 15.8 МБ, але не придатний до відлагодження. Потребує повністю статичної збірки.
* lab03-go:distroless - компромісний варіант для продакшену: розмір невеликий, shell відсутній, але безпечний та стабільний.

**Node.js-застосунок**

У цій частині лабораторної роботи було зроблено:

* Розроблено простий веб-застосунок на Node.js з використанням Express та MongoDB, який реалізує CRUD API для управління бібліотекою книг.
* Впроваджено використання сторонніх бібліотек: express для веб-сервера, mongoose для роботи з MongoDB.
* Створено два Docker-образи для застосунку:

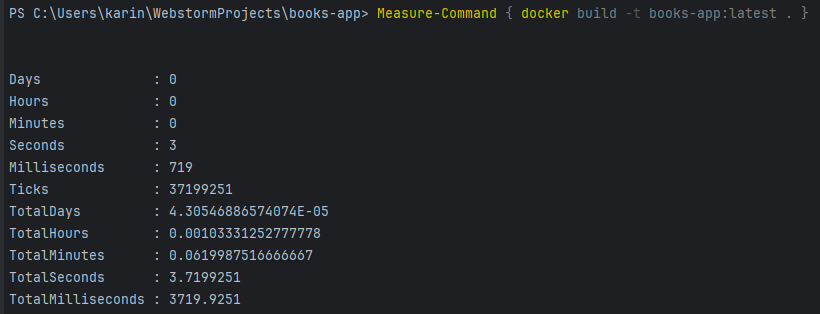
Базовий образ - на основі стандартного Node.js образу.

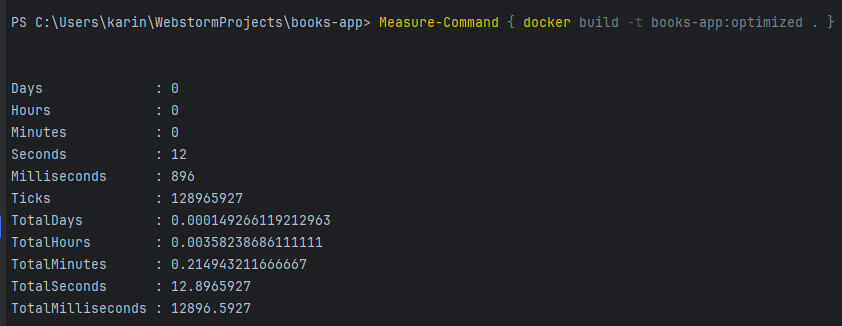
Оптимізований образ - з оптимізаціями для зменшення розміру.

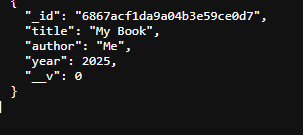
* Налаштовано docker-compose.yml для запуску двох сервісів: застосунку і MongoDB.
* Забезпечено збереження даних MongoDB між перезапусками за допомогою Docker volumes.

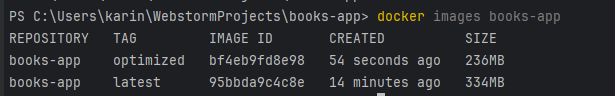
**Порівняльна таблиця**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Образ | Час збірки | Розмір | Збереження даних | Працездатність API |
| Базовий  (node:18-slim) | 3.7 c | 334 МB | Забезпечено через volume mongo\_data | Повністю функціональний CRUD |
| Оптимізований  (node:18-alpine) | 12.9 c | 236 MB | Забезпечено через volume mongo\_data | Повністю функціональний CRUD |

****

****

****

****

**Особливості та труднощі**

* **Час збірки** оптимізованого образу збільшився приблизно в 3.5 раза у порівнянні з базовим. Це пояснюється додатковими кроками оптимізації, наприклад, використанням багатошарової збірки, установки образу (Alpine), видалення зайвих залежностей.
* **Зменшення розміру образу** на 100 МБ є значним плюсом - менший образ швидше передається і розгортається.
* У docker-compose.yml з’явилось попередження про застарілу директиву version. Це пов’язано з тим, що у нових версіях Docker Compose поле version не потрібно, тому я його прибрала.
* Налаштування volume для MongoDB дозволило зберігати дані між перезапусками, що є важливим для реальних застосунків.

**Висновки**

* **Час збірки Docker-образу** напряму залежить від складності конфігурації та базового образу: оптимізований образ збирається довше (приблизно 13 секунд проти 4 секунд у базового), що пов’язано з додатковими кроками оптимізації.
* **Розмір образу** значно зменшується при використанні легких базових образів та багатошарової збірки: оптимізований образ займає 236 МБ, тоді як базовий — 334 МБ, що на 30% менше.
* **Збереження даних MongoDB** забезпечується коректно за допомогою Docker volumes, що гарантує неперервність роботи та відсутність втрати інформації при перезапусках.
* **Використання сторонніх бібліотек (express, mongoose)** спростило розробку веб-застосунку і показало хорошу сумісність із Docker-середовищем.