**Міністерство освіти і науки України  
Національний технічний університет України  
«Київський політехнічний інститут імені Ігоря Сікорського»  
Факультет інформатики та обчислювальної техніки  
Кафедра обчислювальної техніки**

**Лабораторна робота № 3**

з дисципліни  
«**Методології і технології розроблення програмного забезпечення**»

Виконав: Перевірив:

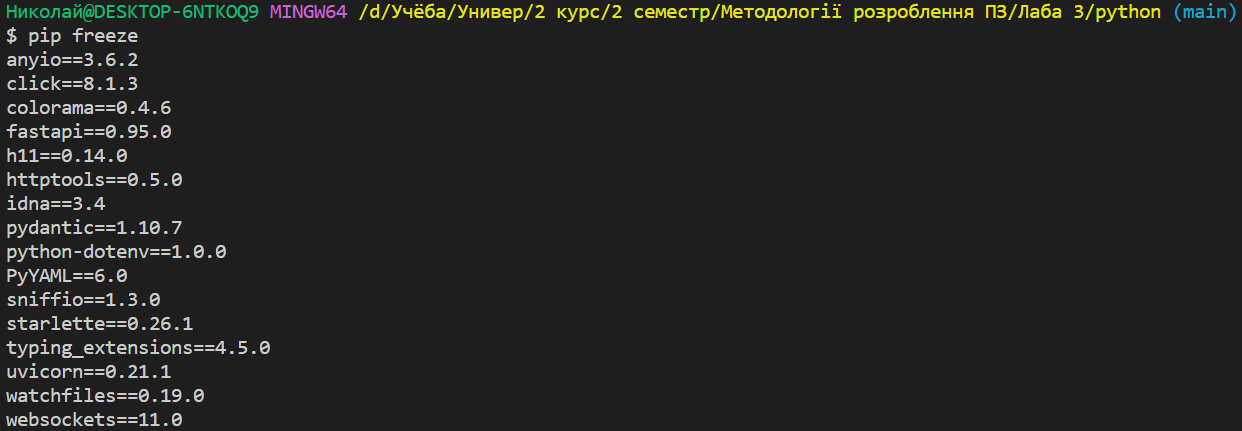
студент групи ІМ-11 [Ковальчук Олександр Миронович](https://schedule.kpi.ua/lecturers?lecturerId=478a9027-fc6f-4657-a381-9575d99d89d0)  
Царик Микола Миколайович  
номер у списку групи: 22

**Київ 2023**

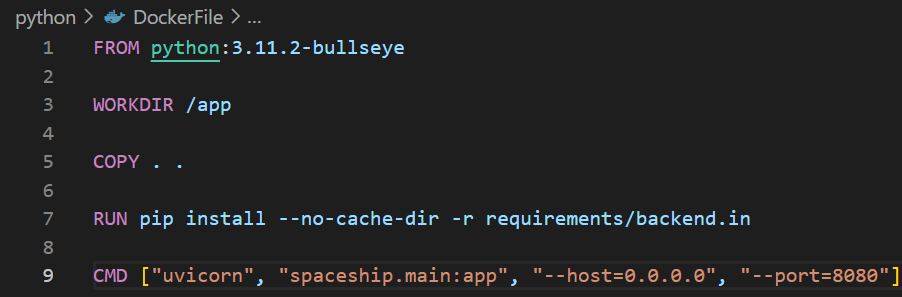
**Python**

1) Спочатку ми створимо контейнер для програми, включивши в нього код програми та всі необхідні залежності, і потім створимо образ. Для отримання інформації про необхідні залежності ми використаємо команду pip freeze

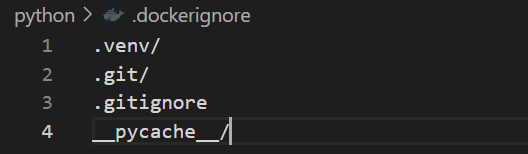
**Dependencies:**



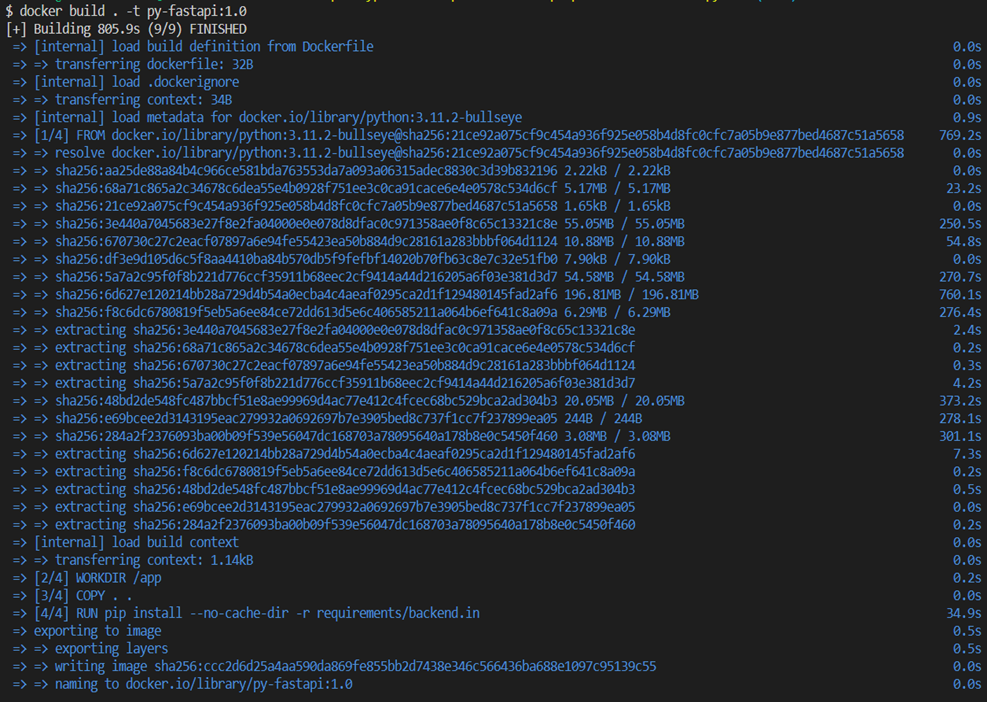
**DockerFile:**



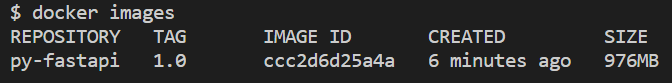
**.dockerignore:**



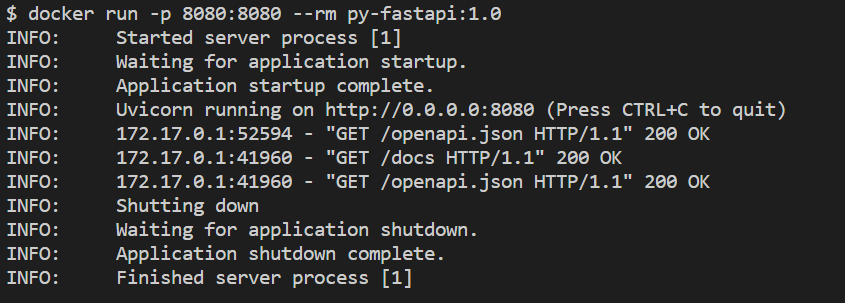
Команда збірки образу:docker build . -t py-fastapi:1.0

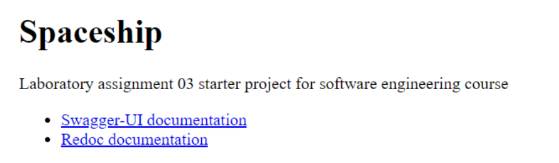


Щоб побачити image скористуємося командою docker images



Запускаємо контейнер за допомогою команди docker run -p 8080:8080 --rm py-fastapi:1.0



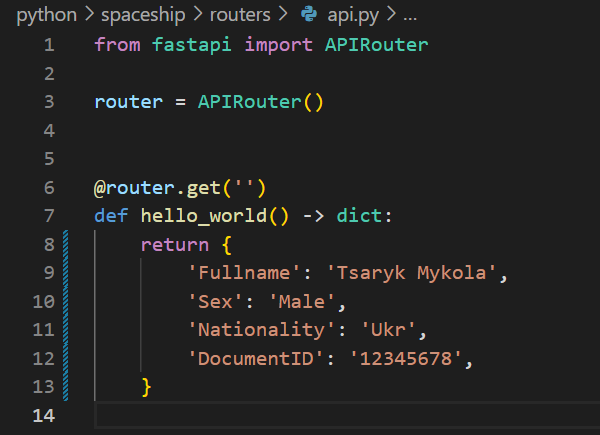
****

**Результат:**

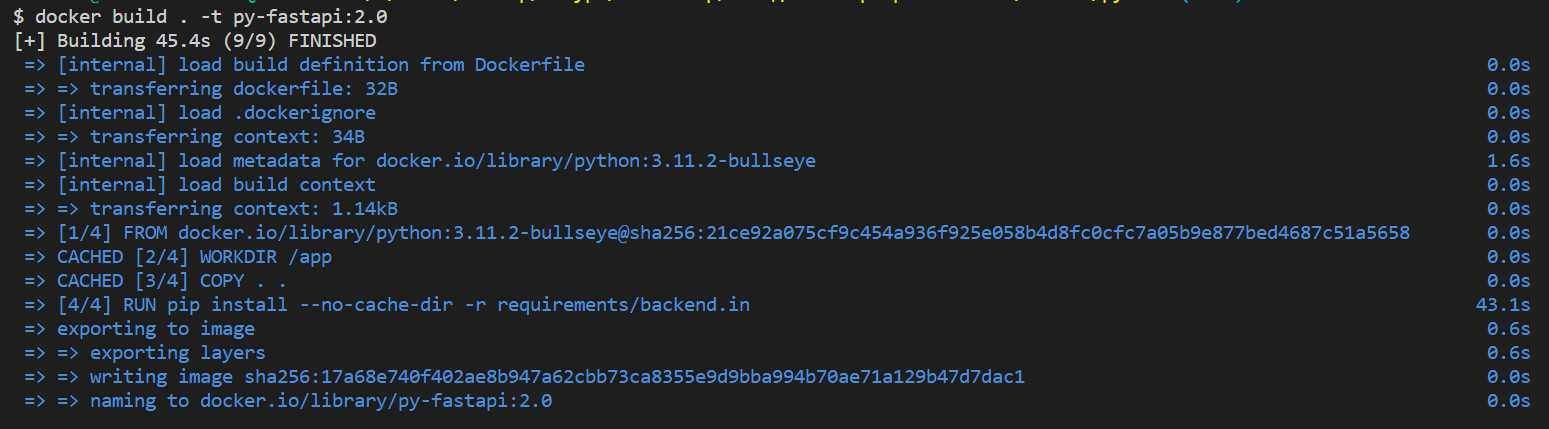
* Час збірки образу: 805.9 с (Довго через дуже погане інтернет-з’єднання)
* Розмір образу: 976 MB
* [Посилання на коміт](https://github.com/TsNikolay/-Software-development-methodologies-Labs-3/commit/10c303ba9e7079024ed347e63eeddffc09e8709a)

2) Тепер ми змінемо файл api.py та створимо образ із новою версією коду.

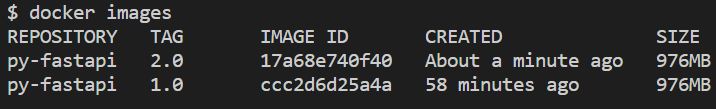
**api.py**



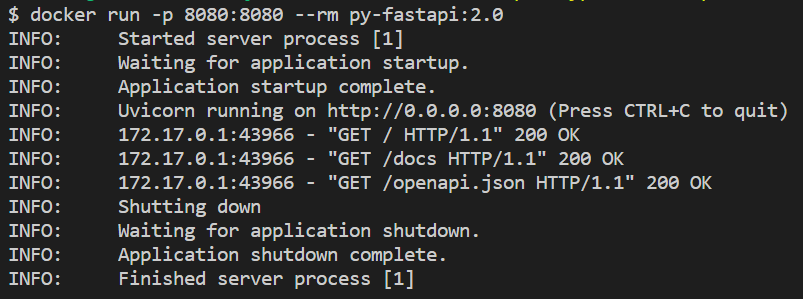
Команда збірки нового образу:docker build . -t py-fastapi:2.0



Щоб побачити image скористуємося командою docker images



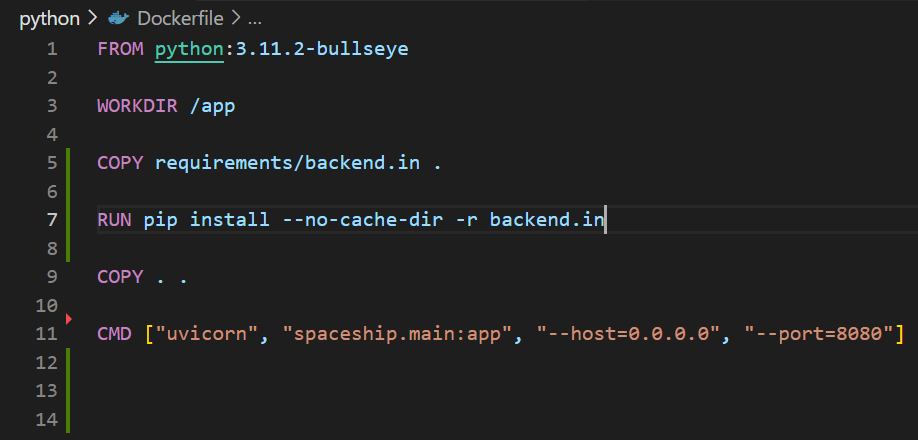
Запускаємо контейнер за допомогою команди docker run -p 8080:8080 --rm py-fastapi:2.0



**Результат:**

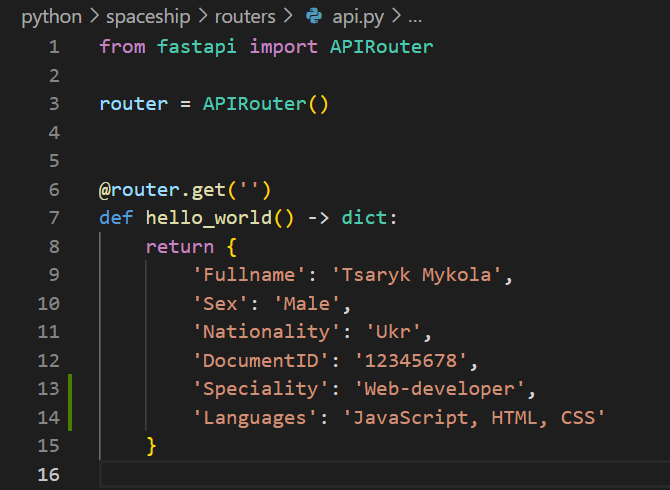
* Час збірки образу: 45.4 с (Цього разу значно швидше бо образ python:slim-bullseye вже був завантажений)
* Розмір образу: 976 MB
* [Посилання на коміт](https://github.com/TsNikolay/-Software-development-methodologies-Labs-3/commit/a23d7becdffb3c4f49a0269b73da2365ee820117)

3) Ми перепишемо dockerfile так, щоб ефективно використовувати шари для побудови образу. Для цього ми спочатку додаємо в образ те, що буде змінюватися найрідше (залежно від проекту), а ті файли, які будуть змінюватися частіше (наш код), додаємо в останню чергу.

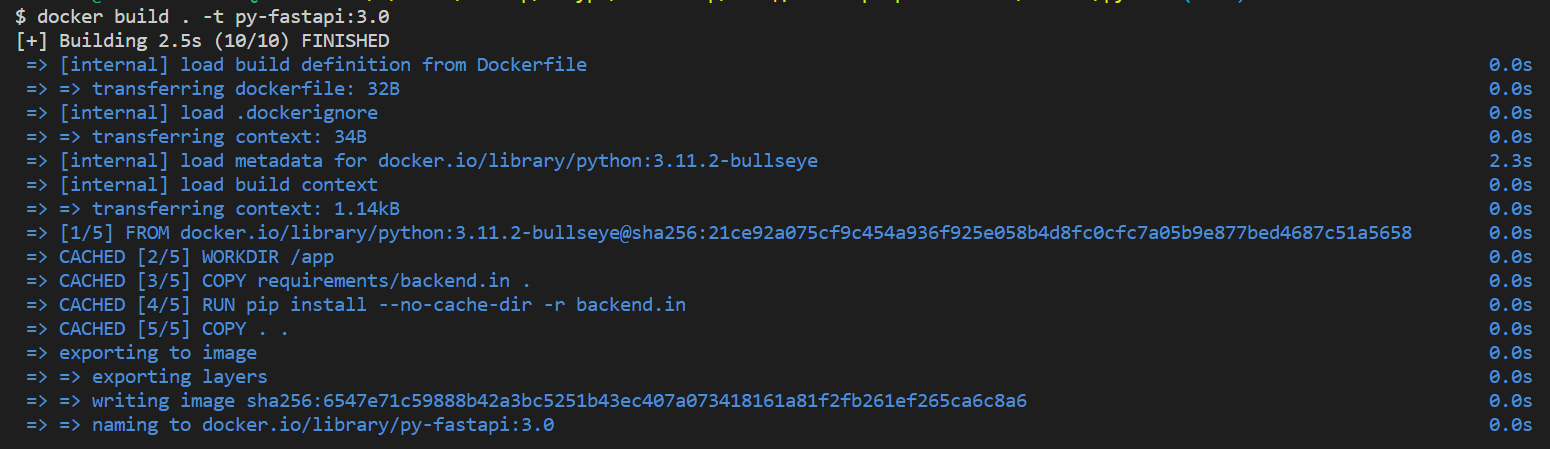


Також змінемо файл api.py щоб показати, що шар з залежностями буде використовуватися з кешу, а не копіюватиметься разом зі змістом директорії

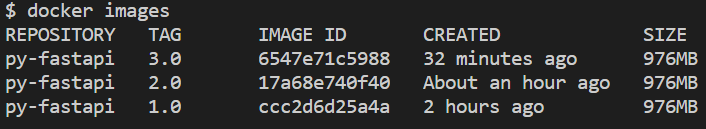
**api.py**

****

Тепер ми спочатку копіюємо файл із залежностями Python окремо від інших файлів директорії, встановлююємо залежності. Завдяки цьому Docker буде кешувати шар з залежностями, а це означає що якщо залежності не змінюються, то цей шар буде використовуватися з кешу у наступних збірках. Це суттєво скоротить час збірки образу.

Команда збірки нового образу:docker build . -t py-fastapi:3.0

Щоб побачити image скористуємося командою docker images



Запускаємо контейнер за допомогою команди docker run -p 8080:8080 --rm py-fastapi:3.0



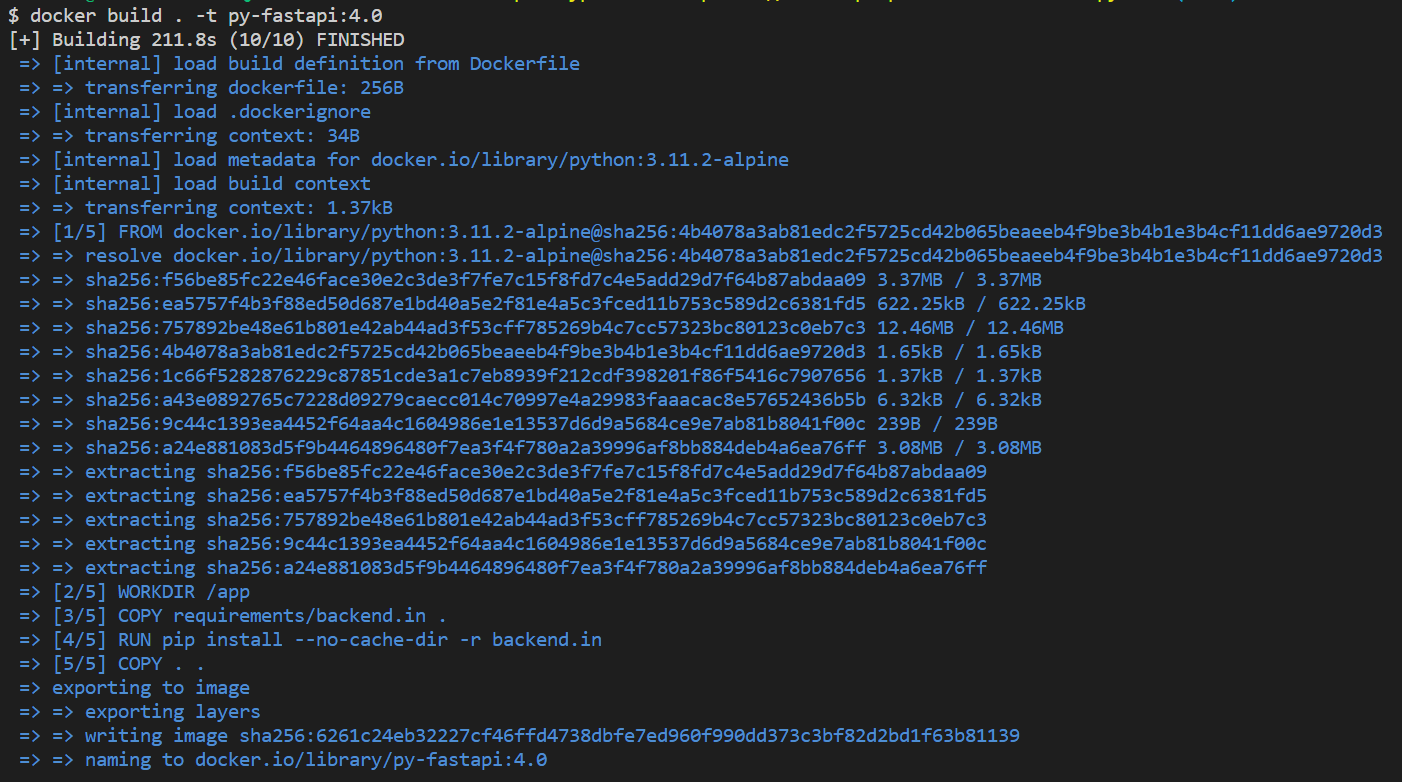
**Результат:**

* Час збірки образу: 2.5 с
* Розмір образу: 976 MB
* [Посилання на коміт](https://github.com/TsNikolay/-Software-development-methodologies-Labs-3/commit/ddfcdb4622a78d67e4eb1e785eafa95c92d9ac77)

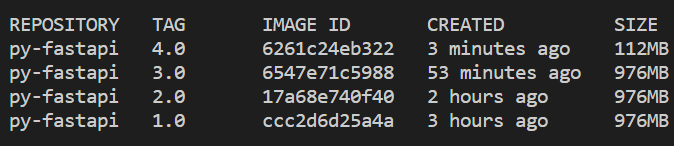
4) За основу проекту візьмемо базове зображення меншого розміру. Наприклад, замість python:3.11.2-bulseye (на основі Debian) ми спробуємо python:3.11.2-alpine (на основі alpine).



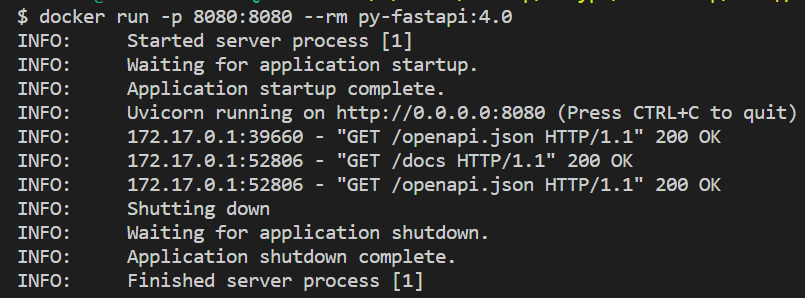
Команда збірки нового образу:docker build . -t py-fastapi:4.0



Щоб побачити image скористуємося командою docker images



Запускаємо контейнер за допомогою команди docker run -p 8080:8080 --rm py-fastapi:4.0



**Результат:**

* Час збірки образу: 218.8с (такий час маємо через те, що завантаження нового базового образу з Docker Hub зайняло деякий час. Виконавши ту ж команду ще раз я отримав результат 1.7с)
* Розмір образу: 112 MB
* [Посилання на коміт](https://github.com/TsNikolay/-Software-development-methodologies-Labs-3/commit/e606488feacfe624e9e7913d41fcf8895991e4f3)

5) Додамо в проект залежність numpy і додамо кінцеву точку в spaceship/routers/api.py, яка генеруватиме 2 випадкові матриці 10x10 і перемножуватиме їх разом. Кінцева точка поверне словник наступної структури:

{

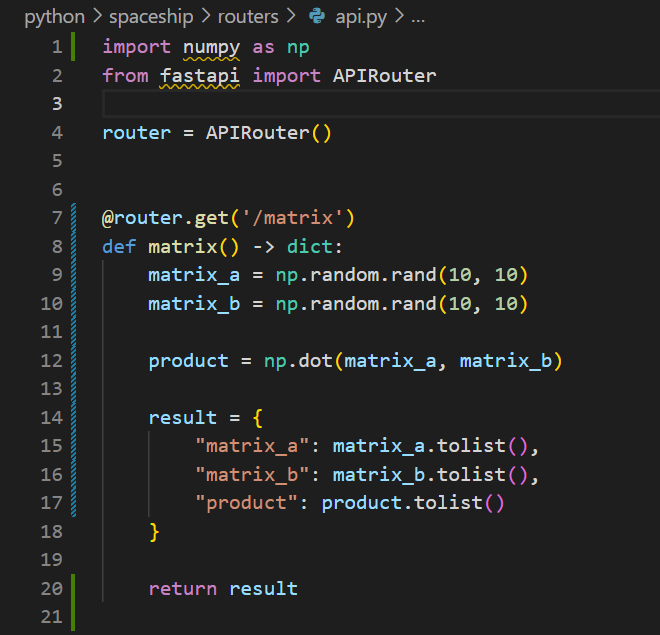
“matrix\_a”: [[...], ...],

“matrix\_b”: [[...], ...],

“product”: [[...], ...],

}

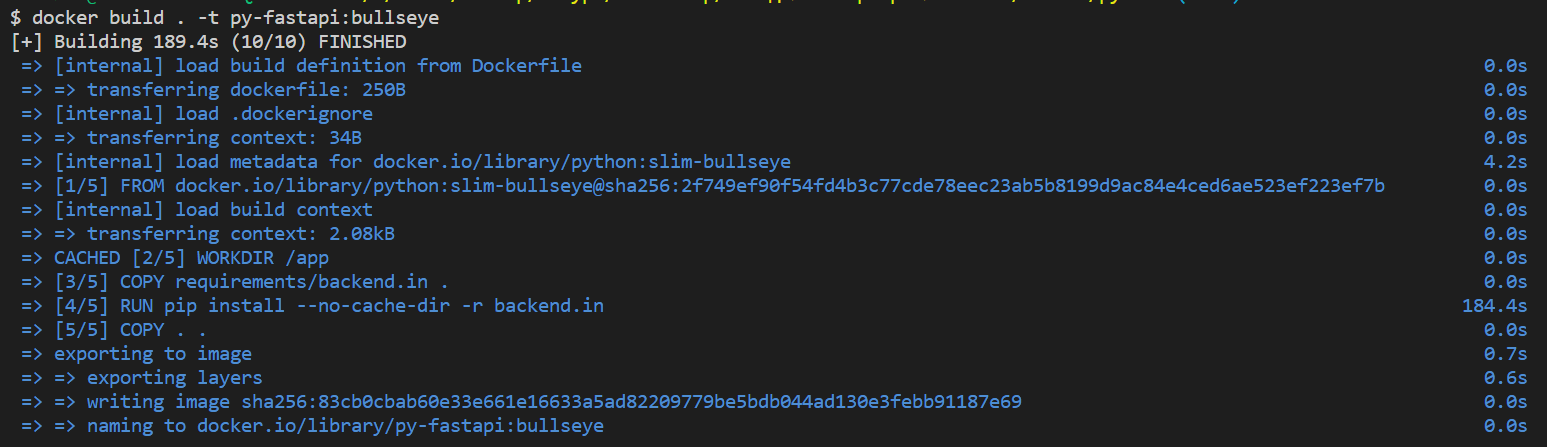
api.py



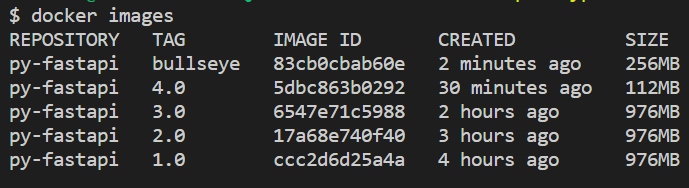
**Збірка python:slim-bullseye:**



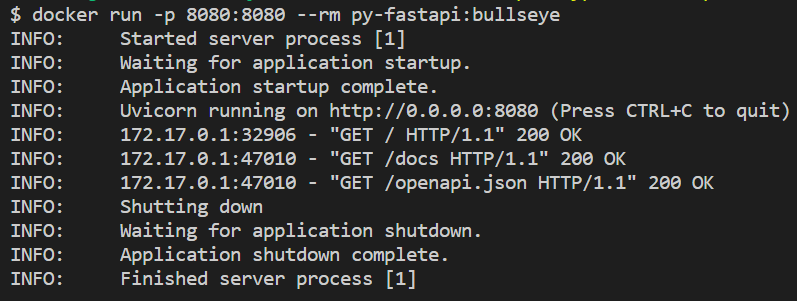
Команда збірки нового образу:docker build . -t py-fastapi:bullseye



Щоб побачити image скористуємося командою docker images

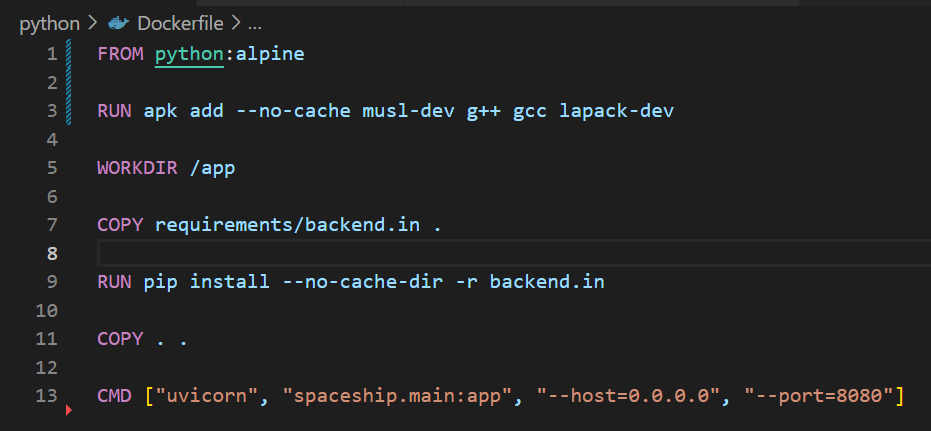


Запускаємо контейнер за допомогою команди docker run -p 8080:8080 --rm py-fastapi:bullseye

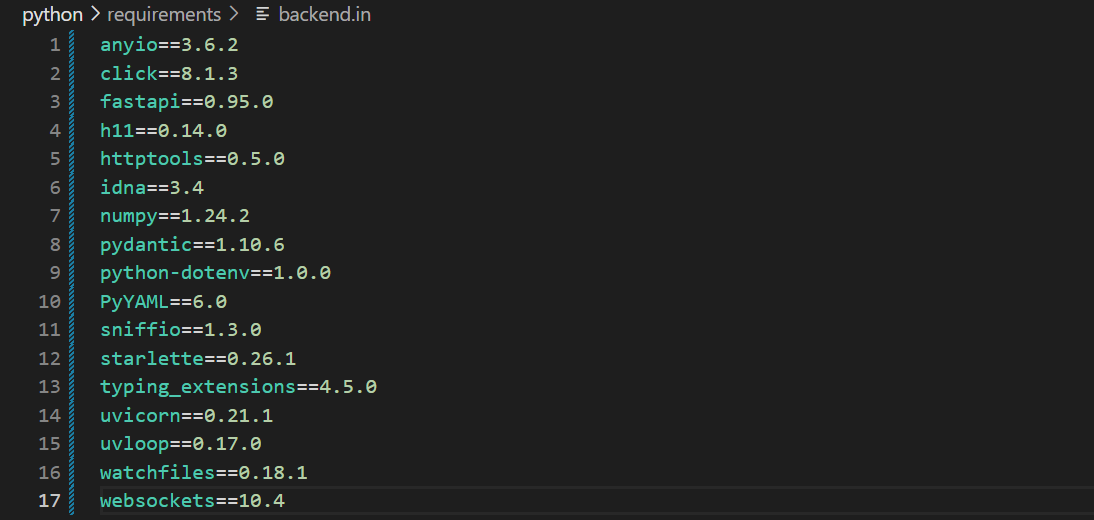


**Збірка python:alpine:**

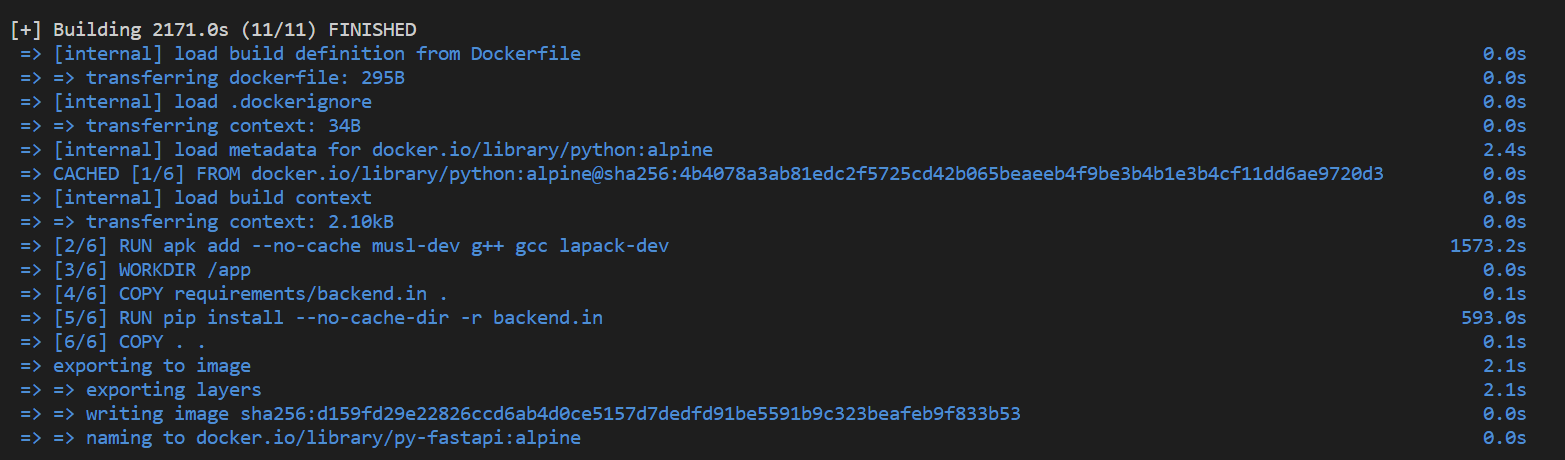
**При білді образу** під час встановлення пакетів pip виникла проблема зі збіркою бібліотеки NumPy, що спричинило помилку встановлення. Для вирішення цієї проблеми я спробував додати інструкцію в Dockerfile, яка встановлює всі необхідні залежності, такі як musl-dev, g++, gcc та інші.

****

**Так виглядає файл залежностей:**

****

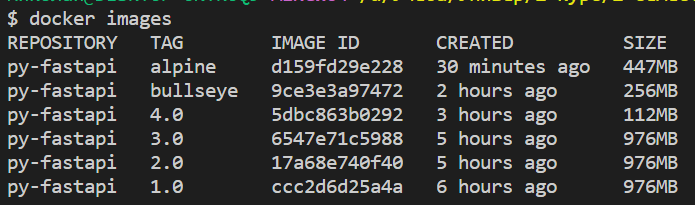
Команда збірки нового образу:docker build . -t py-fastapi:alpine



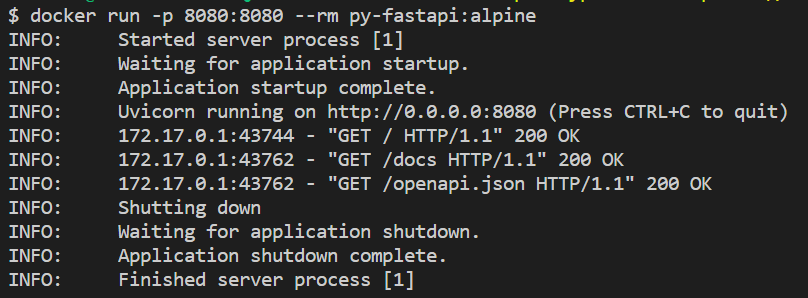
[](https://www.youtube.com/watch?v=yMOVwjoKQZo)

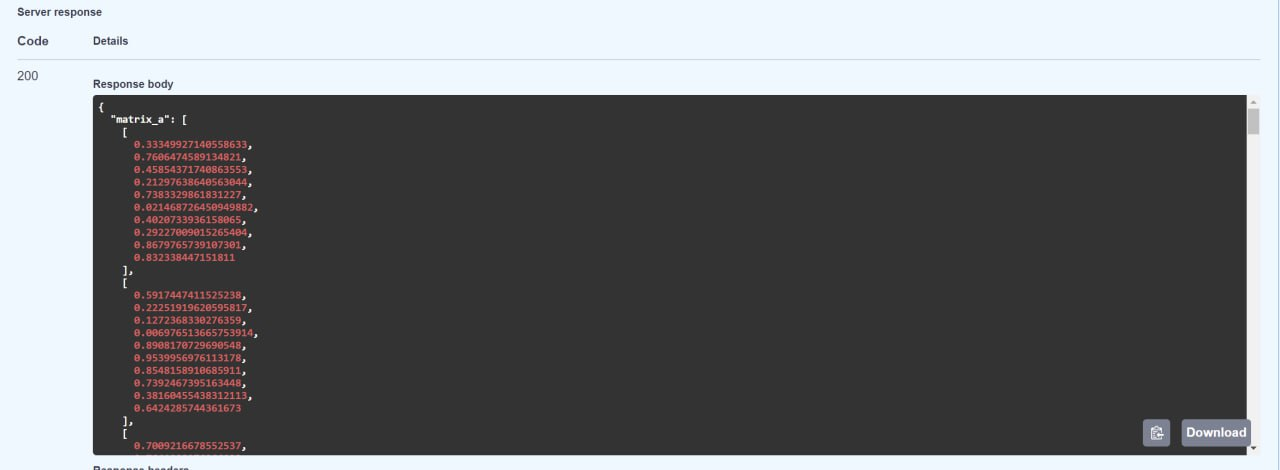
My honest reaction to this building time ->

Щоб побачити image скористуємося командою docker image



Запускаємо контейнер за допомогою команди docker run -p 8080:8080 --rm py-fastapi: alpine





**Результат:**

**python:slim-bullseye:**

Розмір образу: 256 MB

Час збірки образу: 189.4с (при повторному ~= 2.5с)

**python:alpine:**

Розмір образу: 447 МВ

Час збірки образу: 2170 с

**Висновок**

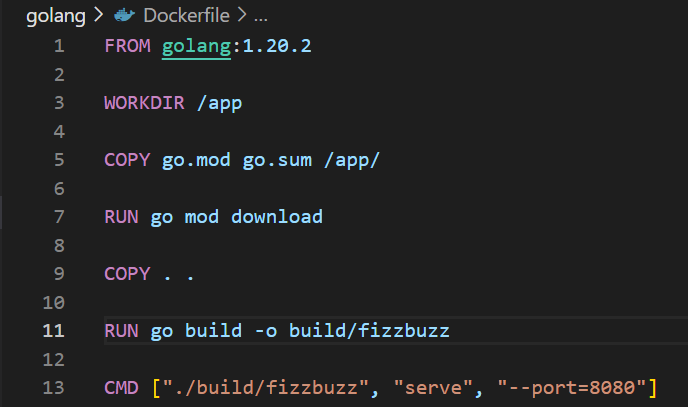
**python:slim-bullseye:** Фактичний розмір образу не відображається реальність, оскільки директорія середовища не була скопійована до цього образу, порівняно з попередніми версіями, що призвело до значного зменшення розміру. Час вийшов мінімальний через те, що всі шари(layers) були взяті з кешу, але при першій збірці всі залежності Python будуть встановлені заново через оновлення файлу backend.in, що, безперечно, займе більше часу.

**python:alpine:** В мене не вийшло зібрати образ при використанні базового образу python:alpine через відсутність необхідних залежностей для побудови numpy. Це призвело до постійних помилок поки я не почав встановлювати необхідні залежностей вручну. І я прийшов до висновку , що це не має сенсу, оскільки розмір та час збірки виявилися набагато більшими порівняно з образом на базі Debian.

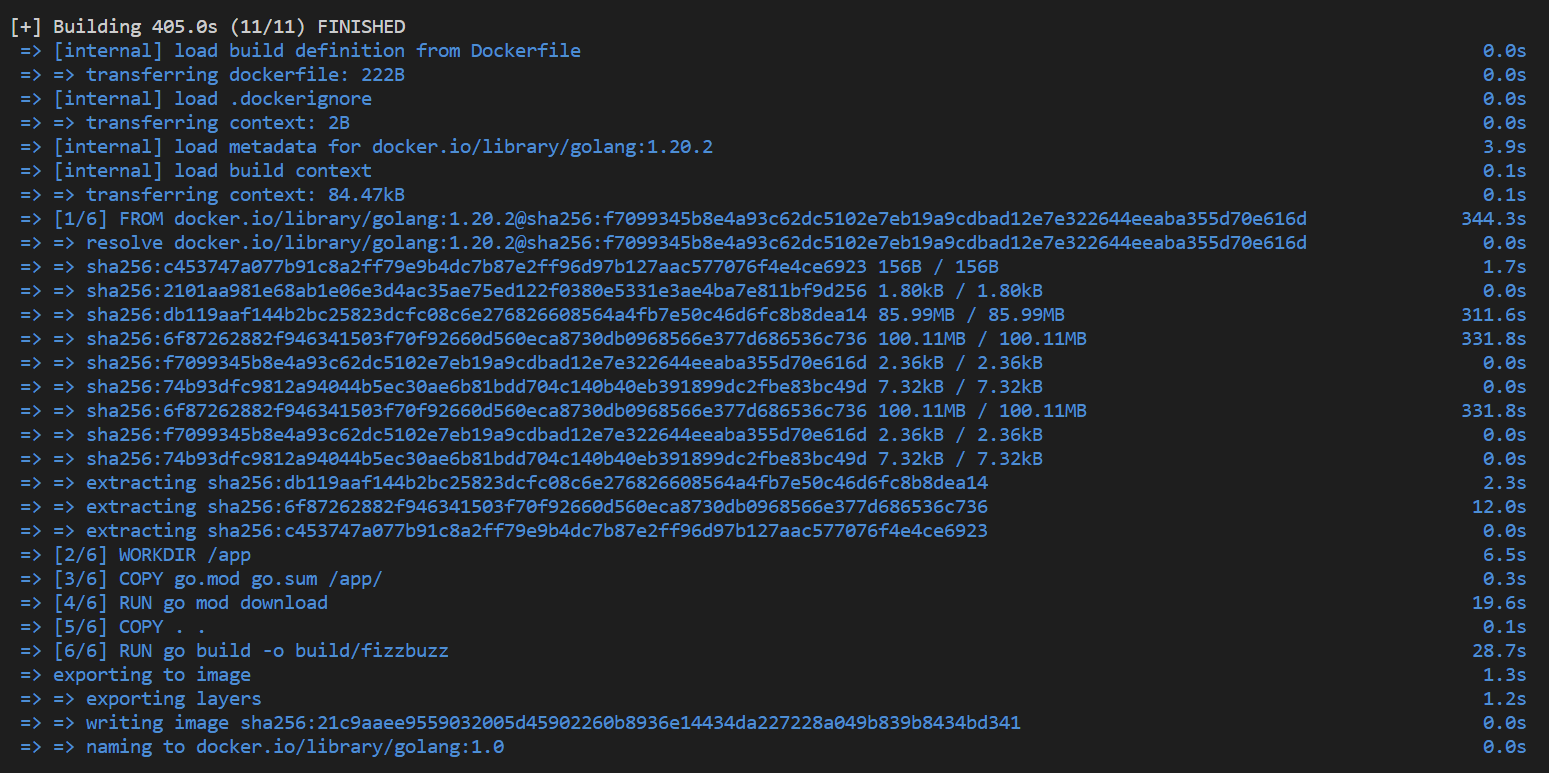
[Посилання на коміт](https://github.com/TsNikolay/-Software-development-methodologies-Labs-3/commit/6d2a15c73f6cff428ee698c10857db1d8ac131c1)

**GOLANG**

1. Створимо образ проекту, додамо залежності до контейнера, створимо виконуваний файл.



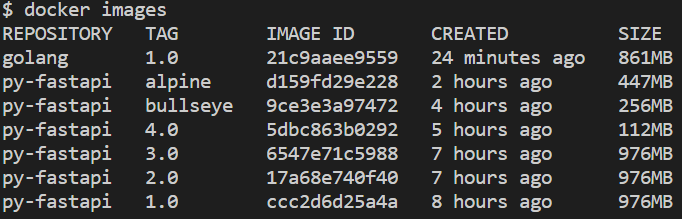
Команда збірки образу:docker build . -t golang:1.0

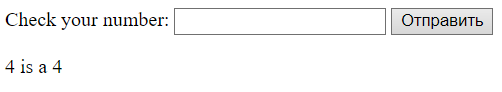


Запускаємо контейнер за допомогою команди docker run -p 8080:8080 --rm golang:1.0



Команда doker images

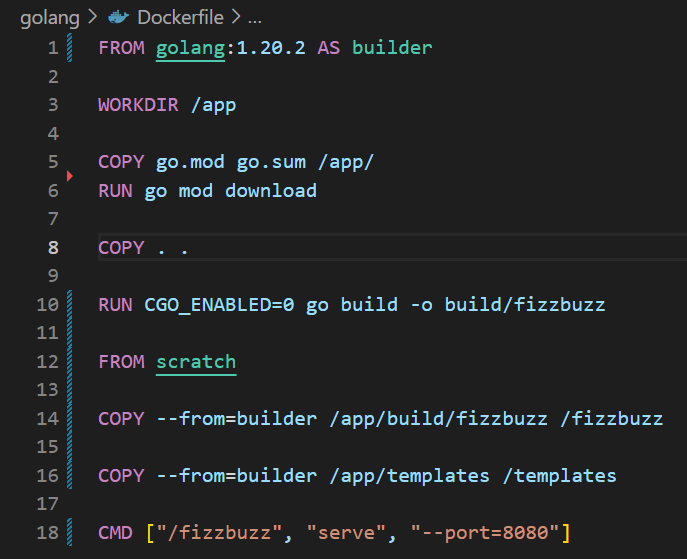


****

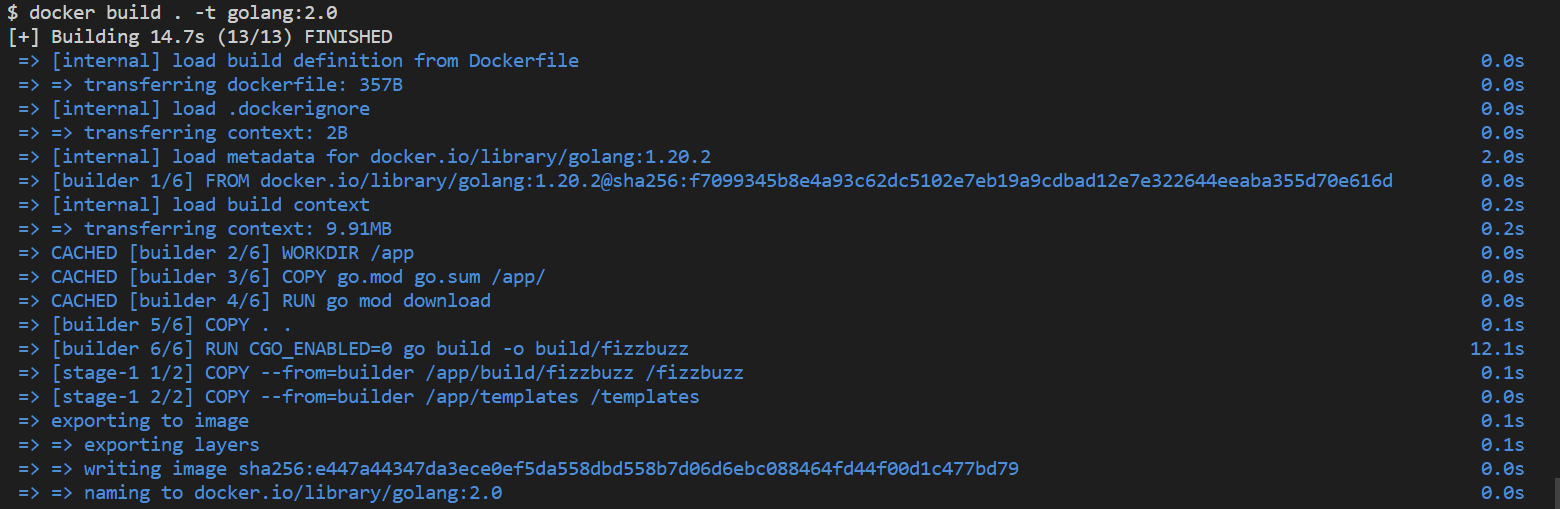
**Результат:**

* Розмір образу: 861 MB
* Час збірки образу: 405с
* Чи усі фали, що там є, потрібні для запуску проекту? Ні. Для запуску необхідний тільки бінарний файл build/fizzbuzz, створений командою go build
* [Посилання на коміт](https://github.com/TsNikolay/-Software-development-methodologies-Labs-3/commit/3ff8be3e20d2314e33f67ac89fc2d574724d55df)

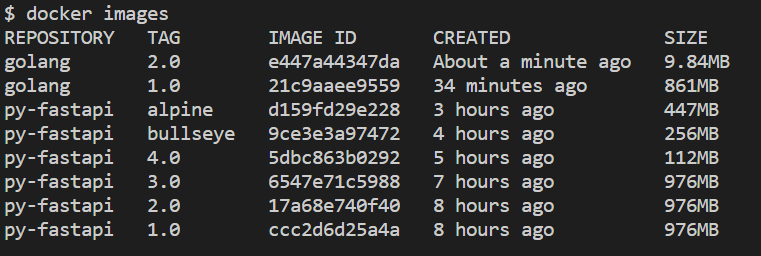
2) Зробимо багатоступінчасте збирання. Для цього на першому етапі збираємо бінарний файл, а на наступному етапі копіюємо файл збірки в новий порожній образ (З нуля).



Команда збірки образу:docker build . -t golang:2.0



Команда docker images



Запускаємо контейнер за допомогою команди docker run -p 8080:8080 --rm golang:2.0



**Результат:**

* Розмір збірки: 9.84 MB
* Час збірки: 14.7 с
* Чи достатньо файлів для запуску нашого проекту? Так, тут пустий образ та бінарний файл . Цього достатньо якщо виконуваний файл скомпільовано статично, бо так всі залежності включені у виконуваний файл. Крім того треба було скопіювати html файл.
* Чи зручно таким образом користуватися? У деяких випадках багатоетапне збирання на основі scratch може виявитися корисним, особливо якщо ви прагнете створити дуже легкий образ Docker, який не містить непотрібних компонентів операційної системи.

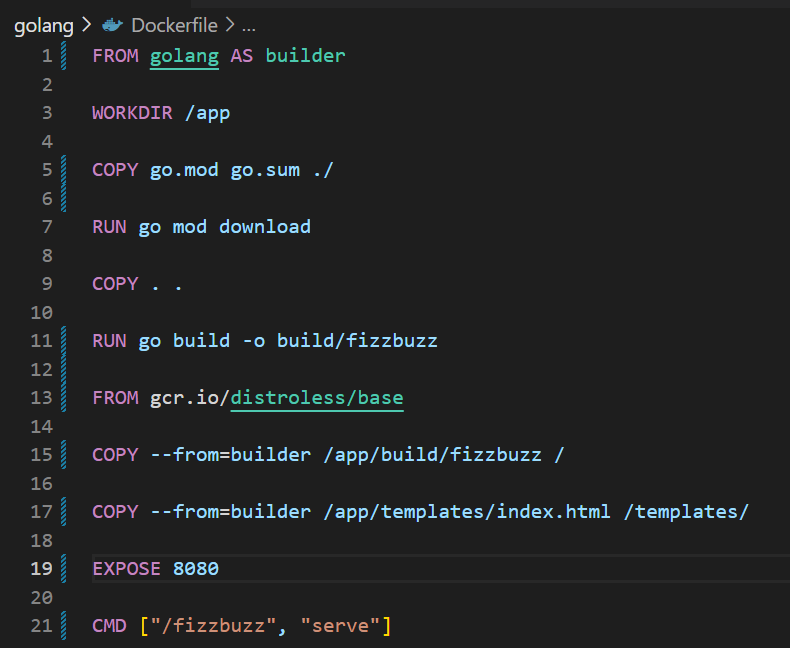
Таке збирання дозволяє зменшити розмір образу Docker, бо дозволяє прибрати файли які не використовуються для запуску програми .

Однак, використання базового тимчасового образу може бути викликом, оскільки він не містить компонентів операційної системи, і ви повинні вручну додавати всі необхідні бібліотеки та інші компоненти, необхідні для запуску програми.

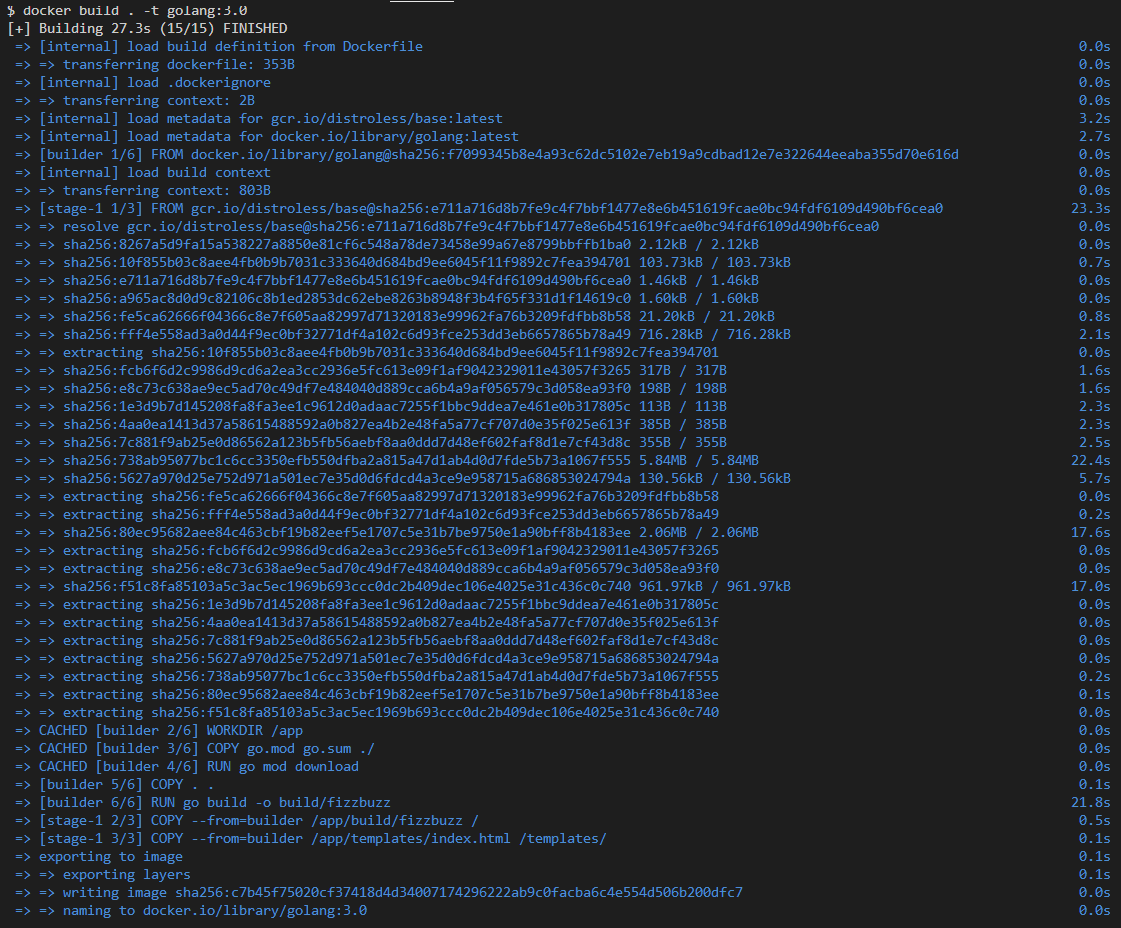
Таким чином, багатоетапне складання на основі scratch може бути зручним для створення легкого та ефективного образу Docker, але не обов'язково підходить для всіх сценаріїв.

[Посилання на коміт](https://github.com/TsNikolay/-Software-development-methodologies-Labs-3/commit/3fe825691a8736803cef7cddd9fb5e40f4a2c4f6)

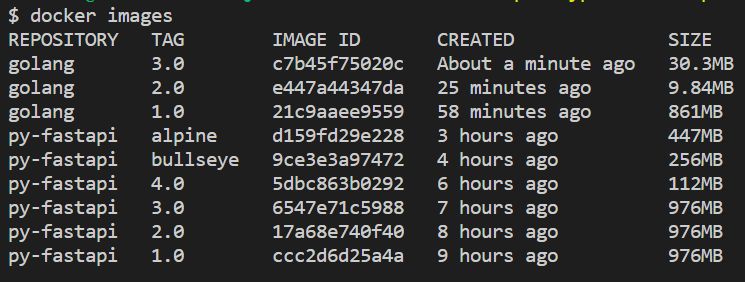
3) Замість скретчу будемо використовувати образи з distroless проекту



Команда збірки образу:docker build . -t golang:3.0



Команда docker images



Запускаємо контейнер за допомогою команди docker run -p 8080:8080 --rm golang:3.0



**Результат:**

* Розмір образу: 30.3 MB
* Час збірки образу: 27.3 c

Scratch - це базовий образ з мінімальним набором компонентів, необхідних для запуску контейнера Docker. У нього немає менеджера пакетів або оболонки, і його розмір декілька мегабайт. Distroless також мінімалістичний, але надає певні функціональні можливості, включаючи необхідні бібліотеки та утиліти.

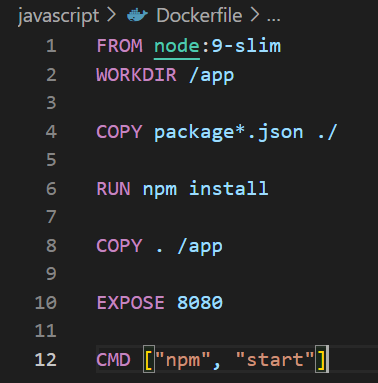
Якщо порівняти використання базових образів Scratch і Distroless, то Scratch дозволяє створювати легкі та оптимізовані образи, що містять тільки необхідні компоненти та залежності. Проте, ручне додавання залежностей може займати багато часу та потребувати додаткових знань, що не є найкращим вибором для розробки.

Distroless-образи призначені для використання з певною мовою програмування та фреймворком, такими як Java, Python або Node.js, і вже містять необхідні залежності та бібліотеки для цих мов та фреймворків. Якщо ви не хочете вручну додавати залежності та бібліотеки і хочете використовувати готовий образ, який містить усі необхідні компоненти для запуску програми, то Distroless-образ може бути найкращим вибором.

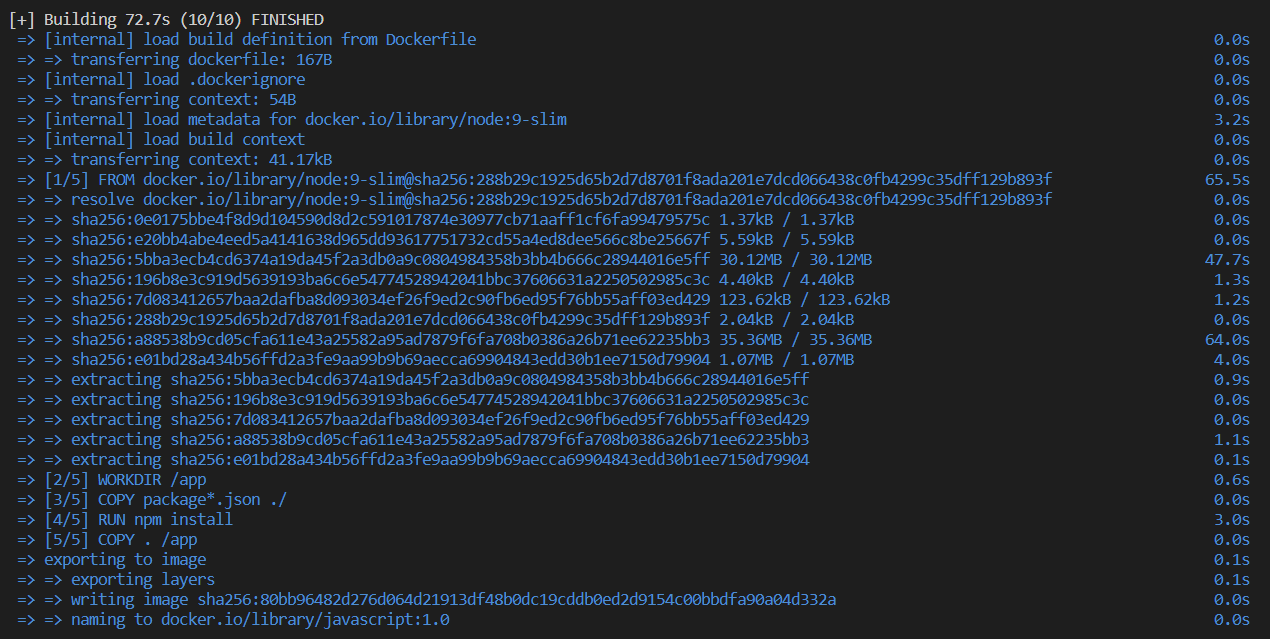
[Посилання на коміт](https://github.com/TsNikolay/-Software-development-methodologies-Labs-3/commit/50c0b4aebb6eb8193ac8ddcac07e42d66cc23bf6)

**JavaScript**

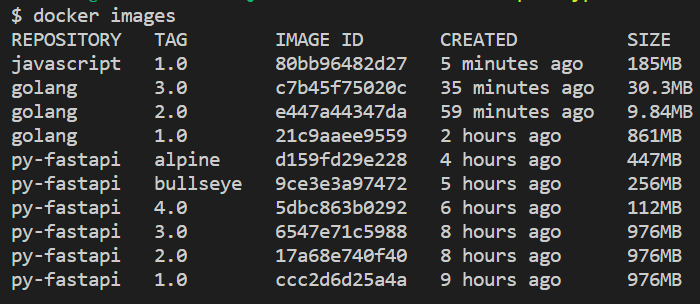
1) Створюємо образ проекту, додаємо залежності в контейнер, збираємо виконуваний файл



Команда збірки образу: docker build . -t js-helloworld:1.0

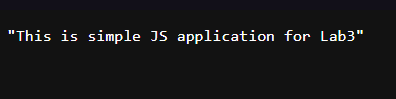


Команда docker images



Запускаємо контейнер за допомогою команди docker run -p 8080:8080 --rm javascript:1.0





**Загальний висновок**

Лабараторна містила три частини на трьох різних мовах: Python, Go, Javascript.

* У частині Python ми експериментували з оптимізацією Dockerfile, порівнювали розміри та час складання різних базових образів та додавали залежності до проекта.
* У частині Golang ми використовували багатоетапні зборки та аналізували вміст отриманих образів.
* У частині Javascript додали свій прсотий проект та використали на ньому докер.

Після тестів можу сказати що ефективність docker образів залежить від вибору базового образу та правильності напиання Dockerfile.

З приводу Dockerfile можу зазначити що є сенс розділяти файли на ті, які будуть змінюватися дуже часто, та ті, які будуть змінюватися дуже рідко, та розміщувати їх у Dockerfile як окремі інструкції. Завдяки цьому ми зменшимо час витрачаємий на збірку, оскільки залежності будуть кешуватися . Крім того, зміни в рідко змінюваних файлах не будуть впливати на процес збірки.

На рахунок базового образу скажу що у випадку коли проект включає виконуваний файл, оптимальним підходом буде багатоетапна збірка образу де спочатку інсталюємо залежності та компілюємо вихідний код в бінарний файл, а потім цей файл копіюється в кінцевий образ та запускається. Для мінімального розміру образу краще всього використовувати Scratch або образи з distroless бо вони дозволяють використовувати лише необхідні файли для виконання програми

Загалом лабораторна робота дала цінний практичний досвід створення образів контейнерів для додатків та оптимізації їх розміру та часу збирання.