Звіт лабораторної роботи №3

** Можливі непропорційні співвідношення зміни часу збірки образу до зміни його розміру пов'язані із нестабільним станом мережевого з'єднання.*

Проєкт на Python

Пункт 1.

Для того, щоб закріпити версії залежностей, я, попередньо встановивши пакети за допомогою `pip install`, використовував команду `pip freeze`.

Після написання Dockerfile я зібрав початковий образ застосунку:

```
[root@fedora Python]# docker build -t mtrpz-py:v1 .
[+] Building 78.7s (9/9) FINISHED
```

Створення першої версії образу зайняло **78,7** секунд. Це пояснюється тим, що базовий образ ***python:3.10-bullseye*** спочатку потрібно було завантажити з Docker Hub (це зайняло **63,9** секунд або ж **81%** від загального часу збірки). Другим найбільш вагомим чинником, що вплинув на час збірки було встановлення залежностей проєкту (**12,1** секунди або **15%** відповідно).

Розмір початкового образу склав аж **964** MB!

```
[root@fedora Python]# docker images
REPOSITORY   TAG       IMAGE ID       CREATED          SIZE
mtrpz-py     v1        2ff31e5b2e1c   20 minutes ago   964MB
```

Як відтворити

1	Версія коду
	Dockerfile
	Команди:
	`[root@... Python]# pip install -r requirements/backend.in`
	`[root@... Python]# pip freeze > requirements/backend.txt`
	`[root@... Python]# docker build -t mtrpz-py:v1 .`

Пункт 2.

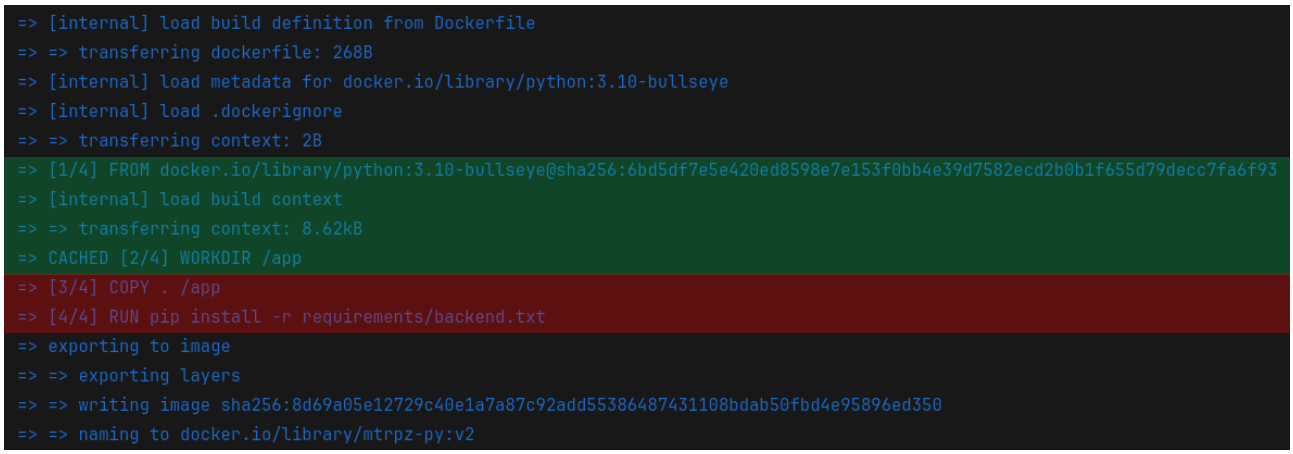
Після змін, внесених до файлу build/index.html, я зібрав новий образ та отримав наступні результати:

- час збірки образу – **13,7** секунд (у **5,75** рази швидше порівняно з ***mtrpz-py:v1***);

- розмір образу – **964** MB (зміни у коді були незначними (<< **0,5** MB), тому розмір залишився практично незмінним).

Досягнути настільки суттєвого зменшення часу збірки допомогло кешування базового образу ***python:3.10-bullseye***, тобто цього разу не потрібно було витрачати час на його завантаження. Також закешувалося виконання інструкції ***`WORKDIR /app`***, бо робочий каталог залишився таким самим. На скриншоті закешовані дії позначено зеленим.

Проте через зміни у коді проєкту, решта кроків виконувалася повторно (на скриншоті це позначено червоним).

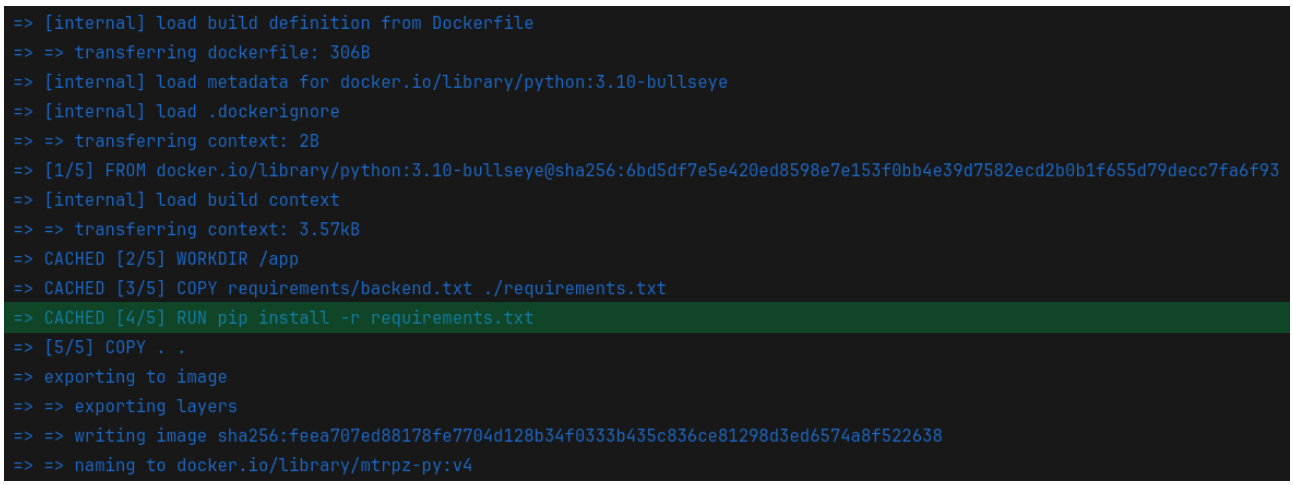


Як відтворити

1	Версія коду Dockerfile Команда: `[root@... Python]# docker build -t mtrpz-py:v2 .`

Пункт 3.

Спочатку я переписав Dockerfile відповідно до умов завдання. Вдосконалена версія Dockerfile повинна забезпечувати встановлення усіх залежностей перед копіюванням файлів проєкту, тобто тепер зміни у коді не мають провокувати повторне встановлення залежностей замість того, щоб брати їх з кешу. Аби перевірити це припущення, створив образ ***mtrpz-py:v3*** з метою закешувати кроки для наступного процесу збірки. Потім змінив файл spaceship/app.py та зібрав новий образ ***mtrpz-py:v4***. Як і передбачалося, цього разу, незважаючи на модифікацію кодової бази, були використані попередньо закешовані залежності (на скриншоті це позначено зеленим).



Результати експерименту:

- час збірки образу – **1,5** секунди (у **9,13** рази швидше порівняно з ***mtrpz-py:v2*** та у **52,47** рази – порівняно з ***mtrpz-py:v1*** (!));
- розмір образу – **964** MB.

Як відтворити

1	Версія коду Dockerfile Команда: `[root@... Python]# docker build -t mtrpz-py:v3 .`
2	Версія коду Dockerfile Команда: `[root@... Python]# docker build -t mtrpz-py:v4 .`

Пункт 4.

В останній версії Dockerfile змінив базовий образ ***python:3.10-bullseye*** на ***python:3.10-alpine***, після чого створив образ ***mtrpz-py:v5***.

Його розмір склав **107** MB (це у **9** разів менше, ніж ***mtrpz-py:v1-4***, у яких базовим образом був ***python:3.10-bullseye***).

Також заміна базового образу зумовила значно менший час збірки: **19** секунд. Це на **59,7** секунд швидше, ніж збірка для ***mtrpz-py:v1***, для якого також ще не було виконано кешування. Різниця очевидна, навіть попри можливі похибки, пов’язані зі станом мережі.

Як відтворити

1	Версія коду Dockerfile Команда:

	`[root@... Python]# docker build -t mtrpz-py:v5 .`

Пункт 5.

Додавши у проєкт залежність **numpy** та реалізувавши необхідну функціональність, я створив новий Dockerfile та зібрав (без попереднього кешування) два образи, порівняння яких наведено у таблиці нижче:

Образ	***mtrpz-py:debian***	***mtrpz-py:alpine***
Дистрибутив-основа	Debian	Alpine
Базовий образ	***python:3.10-bullseye***	***python:3.10-alpine***
Час збірки образу, *с*	**88,2**	**30,3**
Розмір образу, *MB*	**1005**	**197**

Дане порівняння слугує доказом того, що Docker-образи на основі Alpine займають значно менше дискового простору, ніж їхні Debian-аналоги та, відповідно, мають швидший процес збірки.

Окрім цього слід зауважити, що після додавання нової залежності, час збірки та розмір для обох образів дещо збільшилися.

Як відтворити

1	Команди:
	`[root@... Python]# docker rmi mtrpz-py:v{1,2,3,4,5}`
	`[root@... Python]# docker builder prune -a`
	** Надалі я використовуватиму такий самий алгоритм для створення образів без кешування.*

2	Версія коду debian.Dockerfile Команда: `[root@... Python]# docker build -f debian.Dockerfile -t mtrpz-py:debian .`
3	Версія коду alpine.Dockerfile Команда: `[root@... Python]# docker build -f alpine.Dockerfile -t mtrpz-py:alpine .`

Проєкт на Go

Пункт 1.

Проаналізувавши кодову базу проєкту, написав Dockerfile для збірки його образу. Потім на основі ***golang:1.17-alpine*** створив ***mtrpz-go:alpine***. Час, витрачений на збірку образу, склав **62,3** секунди (з яких **22,7** секунд або **36%** від загального часу збірки тривало завантаження базового образу, ще **30,5** секунд або **49%** відповідно зайняло завантаження залежностей проєкту), а його розмір – **936** MB.

Для того щоб переглянути вміст контейнера (лише каталог із файлами проєкту), я використовував утиліту ***tree***:

```
[root@fedora Go]# docker run --rm mtrpz-go:alpine-tree tree -a
.
├── .gitignore
├── README.rst
├── alpine.Dockerfile
├── build
│   └── fizzbuzz
├── cmd
│   ├── query.go
│   ├── root.go
│   └── serve.go
├── go.mod
├── go.sum
├── lib
│   └── fizzbuzz.go
├── main.go
└── templates
    └── index.html

4 directories, 12 files
```

Аналізуючи наведений скриншот, нескладно зауважити, що під час виконання інструкції ***`COPY . .`*** до контейнера потрапили деякі зайві файли, що ніяк не впливають на роботу застосунку (на скриншоті позначені червоним). Проте цю проблему нескладно виправити, перерахувавши усе непотрібне у контейнері у спеціальному конфігураційному файлі .dockerignore.

Як відтворити

1	Версія коду alpine.Dockerfile Команда: `[root@... Go]# docker build -f alpine.Dockerfile -t mtrpz-go:alpine .`
2	Версія коду alpine.Dockerfile Команда: `[root@... Go]# docker build -f alpine.Dockerfile -t mtrpz-py:alpine-tree .`

Пункт 2.

Під час написання Dockerfile для багатоетапної збірки у мене виникли деякі труднощі, пов'язані з відсутністю подібного досвіду. Проте цьому зарадили офіційна документація та численні приклади з Інтернету.

Отриманий образ ***mtrpz-go:scratch*** займає лише **8,95** MB на диску (це у **105** разів менше (!) ніж ***mtrpz-go:alpine***). Цей факт є беззаперечним доказом вражаючої ефективності такого способу контейнеризації. Проте він має і свої недоліки. Оскільки образ має лише мінімальний набір файлів, необхідних для його запуску, це, у свою чергу, виключає можливість виконувати якісь дії всередині контейнера.

```
[root@fedora Go]# docker ps
CONTAINER ID   IMAGE              COMMAND
679a6e13bf34   mtrpz-go:scratch   "./fizzbuzz serve"
[root@fedora Go]# docker export 679a6e13bf34 | tar -t
.dockerenv
dev/
dev/console
dev/pts/
dev/shm/
etc/
etc/hostname
etc/hosts
etc/mtab
etc/resolv.conf
fizzbuzz
proc/
sys/
templates/
templates/index.html
```

На скриншоті вище можна побачити повний перелік файлів, що знаходяться всередині запущеного з ***mtrpz-go:scratch*** контейнера. Як і було зазначено, у ньому є лише файли і каталоги для коректної роботи Docker, виконуваний файл проєкту і компонент для відображення HTML-сторінки (одного двійкового файлу недостатньо для запуску проєкту). Останні два позначені на скриншоті зеленим кольором.

Зручність використання подібного образу напряму залежить від задачі, яку він має вирішувати. Якщо мені, приміром, не потрібна ніяка додаткова функціональність (командний рядок, пакетний менеджер тощо), то я б, не вагаючись, обрав саме цей варіант, зважаючи на його компактність. Але якщо мені окрім робочого застосунку треба ще можливість його досліджувати, то краще використовував би образ на основі якоїсь ОС.

Як відтворити

1	Версія коду scratch.Dockerfile Команда: `[root@... Go]# docker build -f scratch.Dockerfile -t mtrpz-go:scratch .`

Пункт 3.

Новостворений ***mtrpz-go:distroless***, для якого я використав ***gcr.io/distroless/static-debian11***, виявився лише на **28%** (розмір ***mtrpz-go:distroless*** – **11,5** MB) більшим, ніж ***mtrpz-go:scratch*** на базі порожнього образу ***scratch***, тобто все ще дуже легкий, порівняно з ***mtrpz-go:alpine***. Це пояснюється відсутністю більшості системних компонентів (тому, як і в попередньому експерименті, виконання якихось дій всередині контейнера недоступне), проте деякі з них за необхідності можна додати та налаштувати (не сильно вникав у це, проте на безмежних просторах всесвітньої павутини бачив гайди для налаштування bash).

Оскільки повний перелік файлів контейнера, запущеного з цього образу, складається майже з 2-х тисяч елементів, я виніс його в окремий файл distroless-files.txt. Список проєктних файлів залишився таким же, як і для ***mtrpz-go:scratch***.

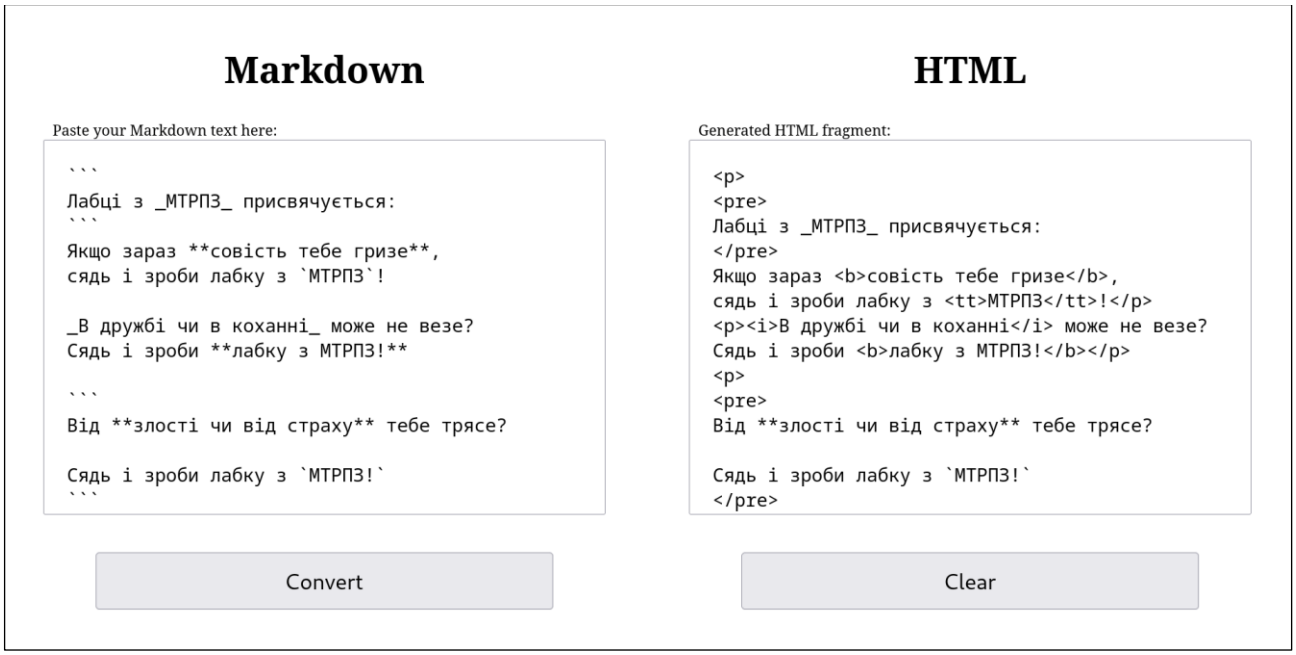
Як відтворити

1	Версія коду distroless.Dockerfile Команда: `[root@... Go]# docker build -f distroless.Dockerfile -t mtrpz-go:distroless .`

Проєкт на JavaScript

Вступ

На основі фреймворку Fastify і минулої лабораторної роботи реалізував простий вебзастосунок, що конвертує спрощену Markdown розмітку у фрагмент HTML (деталі у README лабораторної роботи 2 та JS проєкту лабораторної роботи 3).



Далі створив початкову, частково оптимізовану (містить деякі оптимізації, досліджені у перших двох пунктах лабораторної роботи) версію Dockerfile, що збирає образ на базі ***node:20.12.2-alpine*** (як було з'ясовано раніше, базові образи на основі дистрибутиву Alpine мають найменший розмір).

Початковий образ

- Час збірки (без попереднього кешування) – **15,3** секунди
- Розмір – **148** MB
- Вміст робочого каталогу:

```
[root@fedora JavaScript]# docker run --rm mtrpz-js:v1-tree tree -a
.
├── .eslintignore
├── .eslintrc.json
├── .gitignore
├── .idea
│   ├── .gitignore
│   ├── JavaScript.iml
│   ├── inspectionProfiles
│   │   └── Project_Default.xml
│   ├── jsLibraryMappings.xml
│   ├── modules.xml
│   ├── vcs.xml
│   └── workspace.xml
├── Dockerfile
├── README.md
├── lib
│   ├── converter.js
│   └── server.js
├── node_modules
```

** Зважаючи на значну кількість елементів всередині **node_modules**, я пропустив вміст цього каталогу. До того ж, протягом дослідження він не повинен суттєво змінитися.*

```
├── package-lock.json
├── package.json
└── public
    ├── index.css
    ├── index.html
    └── index.js

505 directories, 2744 files
```

У ході подальших експериментів я планую вдосконалити процес збірки, а потім порівняти початковий і кінцевий образи у плані часу збірки, розміру та вмісту.

Як відтворити

1	Версія коду Dockerfile Команда: `[root@... JavaScript]# docker build -t mtrpz-js:v1 .`
2	Версія коду Dockerfile Команда: `[root@... JavaScript]# docker build -t mtrpz-js:v1-tree .`

Очищення робочого каталогу

Шаблони усіх непотрібних для роботи застосунку файлів та каталогів вніс до .dockerignore. Хоча це майже не вплинуло на час збірки та підсумковий розмір образу, робочий каталог всередині контейнера тепер виглядає значно охайніше:

```
[root@fedora JavaScript]# docker run --rm mtrpz-js:v2-tree tree -a
.
├── lib
│   ├── converter.js
│   └── server.js
├── node_modules
```

...

```
├── package-lock.json
├── package.json
└── public
    ├── index.css
    ├── index.html
    └── index.js

503 directories, 2732 files
```

Як відтворити

1	Версія коду Dockerfile

	Команда: `[root@... JavaScript]# docker build -t mtrpz-js:v2-tree .`
2	Версія коду Dockerfile Команда: `[root@... JavaScript]# docker build -t mtrpz-js:v2 .`

Багатоетапна збірка

Ознайомившись із багатоетапною збіркою для мови програмування Go, вирішив застосувати цей прийом і для проєкту на Node.js. Я дотримувався такого плану: на першому етапі встановити залежності на ***node:20.12.2-alpine***, а на другому – скопіювати усі необхідні файли до нового образу ***gcr.io/distroless/nodejs20-debian11***. Проте мене не влаштовував отриманий результат: час збірки сягнув **18,3** секунди (це на **3** секунди або на **20%** довше ніж збирався початковий образ), розмір – **184** MB (на **36** MB або на **24%** більше відповідно). Ці обставини змусили мене повернутися до попередньої версії Dockerfile (без двоетапної збірки).

Як відтворити

1	Версія коду Dockerfile Команда: `[root@... JavaScript]# docker build -t mtrpz-js:v3 .`

MythBusters moment

Після проведених досліджень я ще деякий час блукав мережею у пошуках нових способів оптимізації процесу збірки для Node.js проєкту і знайшов одну цікаву статтю на цю тему. Гадаю було би непогано пересвідчитися у дієвості запропонованого там методу та відобразити це у звіті лабораторної роботи.

Я змінив Dockerfile відповідно до рекомендацій зі статті (тепер у якості базового образу використовується остання стабільна версія дистрибутиву Alpine (`***FROM alpine***`), на який потім окремо встановлюється поточна LTS-версія Node.js та пакети npm (`***RUN apk add --no-cache nodejs npm***`)). Далі, використовуючи цей Dockerfile, зібрав образ ***mtrpz-js:v4***. Розмір даного образу був на **62,3** MB або **42%** легшим, ніж початковий образ, а час збірки зменшився на **0,6** секунд або **4%** відповідно.

Отже, цей спосіб можна вважати робочим. Думаю розмір зменшився через те, що на ***node:20.12.2-alpine***, окрім Node.js та npm, можуть бути встановлені якісь додаткові інструменти для розробки Node.js-застосунків. Оскільки для

коректної роботи мого проєкту не потрібно нічого, окрім середовища виконання і пакетного менеджера, я збиратиму фінальний образ, користуючись розглянутим методом.

Як відтворити

1	Версія коду Dockerfile Команда: `[root@... JavaScript]# docker build -t mtrpz-js:v4 .`

Підсумок

Отже, пора зібрати фінальну версію образу проєкту та порівняти її з початковою.

Остаточний образ

- Час збірки (без попереднього кешування) – **14,4** с
- Розмір – **85,7** MB
- Вміст робочого каталогу – такий же як для ***mtrpz-js:v2*** (з розділу про очищення робочого каталогу)

Як відтворити

1	Версія коду Dockerfile Команда: `[root@... JavaScript]# docker build -t mtrpz-js .`

У результаті дослідження, проведеного в цьому пункті лабораторної роботи, мені вдалося скоротити час збірки на **6%**, зменшити його розмір на **42%**, а також зробити так, щоб у робочому каталозі був лише мінімально необхідний для запуску проєкту набір файлів і каталогів.

Висновки

На основі отриманих навичок створення контейнерів для різних мов програмування, а також їх подальшої оптимізації, я сформулював такі основні рекомендації упаковки застосунків у контейнери:

- правильний порядок інструкцій у Dockerfile має важливе значення. Кожна інструкція утворює окремий шар, який у випадку відсутності змін під час виконання цієї інструкції буде братися з кешу, що, у свою чергу, може суттєво пришвидшити процес збірки. Проте, якщо якийсь із шарів все ж зазнає змін, то всі наступні інструкції також будуть виконуватися повторно.

Тому спочатку слід розміщувати те, що змінюватиметься рідше (наприклад, встановлення залежностей), а далі те, що змінюватиметься частіше (наприклад, копіювання кодової бази).

- вибір базового образу має ґрунтуватися на вимогах до контейнеризованого проєкту. Якщо найбільш пріоритетними є розмір та швидкість збірки, можна обрати базовий образ на основі дистрибутиву Alpine. Якщо ж важливу роль відіграють надійність, стабільність та розширені системні можливості, слід віддати перевагу образам на основі Debian.
- багатоетапна збірка може допомогти відчутно полегшити кінцевий образ. Її ідея полягає у тому, щоб збирати та запускати проєкт на різних базових образах. Особливо ефективно цей метод працює для компільованих мов програмування, наприклад, Go, адже тоді готовий застосунок можна запустити з мінімальним набором системних компонентів (або взагалі без них). Для таких випадків існують відповідні базові образи (наприклад, ***scratch*** та ***distroless***).
- усі непотрібні для роботи застосунку файли мають бути перераховані у .dockerignore аби не потрапити всередину контейнера.