

Python

1. При спробі запуску просту було помічено одруківку в рідмі файлі (\$./venv/bin/acticate). Для фіксування залежностей було використано команду **pip freeze**, вивід якої було перенаправлено у файл requirements.txt (**pip freeze > requirements.txt**). Далі я написав **Dockerfile**, на основі якого буде створено образ (docker image).

Ось власне і сам [Dockerfile](#). Для збірки образу на основі **Dockerfile** було використано команду **docker build -t python_mtsd:latest**.

Можна побачити, що час першої збірки образу сягає 34.0 секунди

```
(.venv) ~/Programming/Methodologies_Lab3/Python
sudo docker build -t python_mtsd:latest .

[+] Building 34.0s (11/11) FINISHED
```

Такий час можна пояснити таким чином :

- базовий образ [python:3.9-slim](#) для версії Python 3.9 було використано не з кешу Docker, а з DockerHub (локально немає образу);

```
[1/5] FROM docker.io/library/python:3.9-slim@sha256:0b4b0801ae9ae61bb57bc738b1efbe4e16b927fd581774c8edfed90f0e0f01ad
=> resolve docker.io/library/python:3.9-slim@sha256:0b4b0801ae9ae61bb57bc738b1efbe4e16b927fd581774c8edfed90f0e0f01ad
=> sha256:8a1e25ce7c4f75e372e9884f8f7b1bedcfe4a7a7d452eb4b0a1c7477c9a90345 29.12MB / 29.12MB
=> sha256:1103112ebfc46e01c0f35f3586e5a39c6a9ffa32c1a362d4d5f20e3783c6fdd7 3.51MB / 3.51MB
=> sha256:6e52db3290c070e9fa594bc1940b70208e44df68c33cd7dc6f124dd764a902f2 11.89MB / 11.89MB
=> sha256:0b4b0801ae9ae61bb57bc738b1efbe4e16b927fd581774c8edfed90f0e0f01ad 1.86kB / 1.86kB
=> sha256:5fa679d492e5cb06ce42ddbfb02609291f80f3288b799f6bc2b752a8b51268fc 1.37kB / 1.37kB
=> sha256:04efcd1709fbaa9e0322704432ca717c602cc8bf3d8b1e7c27c1b58d1ff825a0 6.92kB / 6.92kB
=> sha256:937bce5dbc70ad48eca027ccde2bc65d0490b7f84657ab4e68a3710405f4168c 243B / 243B
=> sha256:05e63546fee1cc38a926d111acae4390f54a48d631272f0472191b8bcec74792 3.13MB / 3.13MB
=> extracting sha256:8a1e25ce7c4f75e372e9884f8f7b1bedcfe4a7a7d452eb4b0a1c7477c9a90345
=> extracting sha256:1103112ebfc46e01c0f35f3586e5a39c6a9ffa32c1a362d4d5f20e3783c6fdd7
=> extracting sha256:6e52db3290c070e9fa594bc1940b70208e44df68c33cd7dc6f124dd764a902f2
=> extracting sha256:937bce5dbc70ad48eca027ccde2bc65d0490b7f84657ab4e68a3710405f4168c
=> extracting sha256:05e63546fee1cc38a926d111acae4390f54a48d631272f0472191b8bcec74792
[internal] load build context
=> transferring context: 59.24MB
```

- такі дії, як встановлення робочої директорії, копіювання файлу із залежностями та власне встановлення залежностей, не були закешовані.

```
[2/5] WORKDIR /app
[3/5] COPY requirements.txt .
[4/5] RUN pip install --no-cache-dir -r requirements.txt
[5/5] COPY . /app
exporting to image
```

Розмір образу становить 227 MB

```
(.venv) ~/Programming/Methodologies_Lab3/Python git:[main]
sudo docker images
[sudo] password for daniorerio:
REPOSITORY          TAG          IMAGE ID          CREATED          SIZE
python_mtsd         latest       a47b5ba7b182     58 minutes ago  227MB
```

- Внесено зміни в [spaceship/routers/api.py](#) для виводу персональної інфо студента. Зберемо наново образ, але закешуємо базовий образ. Бачимо, що час збірки зменшився до 3.1 секунди, це було досягнуто завдяки кешуванню. Тепер кешування відбувається для: базового образу [python:3.9-slim](#) замість завантаження з DockerHub; **WORKDIR /app**, бо не змінюється робочий каталог; **COPY requirements.txt .**, бо вміст файлу не змінювався; **RUN pip install --no-cache-dir -r requirements.txt**;

```
(.venv) ~/Programming/Methodologies_Lab3/Python git:[main]
sudo docker build -t python_mtsd:2.0 .
[sudo] password for daniorerio:
[+] Building 3.1s (11/11) FINISHED
=> [internal] load build definition from Dockerfile
=> => transferring dockerfile: 231B
=> [internal] load metadata for docker.io/library/python:3.9-slim
=> [auth] library/python:pull token for registry-1.docker.io
=> [internal] load .dockerignore
=> => transferring context: 2B
=> [internal] load build context
=> => transferring context: 266.01kB
=> [1/5] FROM docker.io/library/python:3.9-slim@sha256:0b4b0801ae9ae61bb57bc738b1efbe4e16b927fd581774c8edfed90f0e0f01ad
=> CACHED [2/5] WORKDIR /app
=> CACHED [3/5] COPY requirements.txt .
=> CACHED [4/5] RUN pip install --no-cache-dir -r requirements.txt
=> [5/5] COPY . .
=> exporting to image
=> => exporting layers
=> => writing image sha256:08dbe483346927e9ac76f3b8ba4352b103fcc70aefaf8ddfa2920aadb1fa90ab
=> => naming to docker.io/library/python_mtsd:2.0
```

Також видно, що розмір образу не змінився, так і залишився 227 MB

```
(.venv) ~/Programming/Methodologies_Lab3/Python git:[main]
sudo docker images
REPOSITORY          TAG          IMAGE ID          CREATED          SIZE
python_mtsd         2.0          08dbe4833469     8 seconds ago   227MB
```

- Ще раз внесемо зміни до [spaceship/routers/api.py](#) (додамо ще три додаткових поля). Зробимо оптимізований **Dockerfile** на [неоптимізований](#). Тепер будемо копіювати всі файли в образ перед

встановленням залежностей. Це означає, що будь-які зміни у файлах проєкту призведуть до повного перезапуску кроку встановлення залежностей, що призведе до збільшення часу збірки.

Зробімо перший запуск образу будь-якого без кешування (і без кешування базового образу: беремо з DockerHub). Бачимо достатньо суттєве збільшення часу (на 6.9 секунд).

```
(.venv) ~/Programming/Methodologies_Lab3/Python git:[main]
sudo docker build -t python_mtsd:unopt .

[+] Building 40.9s (11/11) FINISHED
```

Тепер спробуємо подивитися на час з кешуванням. Базовий образ є локально, його не треба брати з DockerHub. Однак після внесених змін (які не впливають на залежності) у код можна побачити, що команда **RUN** не кешується, а вона виконується 13.1 секунди! Цього недоліку не було в оптимізованій версії **Dockerfile**. Перевищення на обличчя: аж на 12.7 секунд

```
(.venv) ~/Programming/Methodologies_Lab3/Python git:[main]
sudo docker build -t python_mtsd:unopt1.4 .

[+] Building 15.8s (10/10) FINISHED
⇒ [internal] load build definition from Dockerfile
⇒ ⇒ transferring dockerfile: 247B
⇒ [internal] load metadata for docker.io/library/python:3.9-slim
⇒ [internal] load .dockerignore
⇒ ⇒ transferring context: 2B
⇒ [1/5] FROM docker.io/library/python:3.9-slim@sha256:0b4b0801ae9ae61bb57bc738b1efbe4e16b927fd581774c8edfed90f0e0f01ad
⇒ [internal] load build context
⇒ ⇒ transferring context: 265.51kB
⇒ CACHED [2/5] WORKDIR /app
⇒ [3/5] COPY . /app
⇒ [4/5] COPY requirements.txt requirements.txt
⇒ [5/5] RUN pip install --no-cache-dir -r requirements.txt
⇒ exporting to image
⇒ ⇒ exporting layers
⇒ ⇒ writing image sha256:e7cf45780f585ee6b128b3253da29d9a2625081b947473156fd416acf22bbb6f
⇒ ⇒ naming to docker.io/library/python_mtsd:unopt1.4
```

Розмір образів не змінився

```
(.venv) ~/Programming/Methodologies_Lab3/Python git:[main]
sudo docker images
```

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
python_mtsd	unopt1.4	e7cf45780f58	19 minutes ago	227MB
python_mtsd	unopt	78b3be4546f0	44 minutes ago	227MB

4. На оптимізованому [Dockerfile](#) змінимо базовий образ з `python:3.9-slim` на [python:3.9-alpine](#). Можна одразу побачити, що саме встановлення базового образу з DockerHub швидше. Якщо порівнювати з першим пунктом, то там час повного встановлення без хешування становив 34 сек, а ми досягли 22.6 секунд, що дійсно зменшило час на 11.4 секунди.

```
(.venv) ~/Programming/Methodologies_Lab3/Python git:[main]
sudo docker build -t python_mtsd:alpine .

[+] Building 22.6s (11/11) FINISHED
=> [internal] load build definition from Dockerfile
=> => transferring dockerfile: 233B
=> [internal] load metadata for docker.io/library/python:3.9-alpine
=> [auth] library/python:pull token for registry-1.docker.io
=> [internal] load .dockerignore
=> => transferring context: 2B
=> [internal] load build context
=> => transferring context: 265.70kB
=> [1/5] FROM docker.io/library/python:3.9-alpine@sha256:99161d2323b4130fed2d849dc8ba35274d1e1f35da170435627b21d305dad954
=> => resolve docker.io/library/python:3.9-alpine@sha256:99161d2323b4130fed2d849dc8ba35274d1e1f35da170435627b21d305dad954
=> => sha256:99161d2323b4130fed2d849dc8ba35274d1e1f35da170435627b21d305dad954 1.65kB / 1.65kB
=> => sha256:756e9fe7f6ad5a18d302006b445d3c9e8e1d868938c2562415a30c240e4bdd56 1.37kB / 1.37kB
=> => sha256:8f4429cefc37edb511b1140db530b9d613b71ef9c1f79ffda963694c0926c107 6.25kB / 6.25kB
=> => sha256:3cdf40b8bda8e4ca4be0f5fa7f1d128907271efcbc72cbfc7c8b0f939ec25ea 619.60kB / 619.60kB
=> => sha256:a05d724777b115e78126981b762c012b90a5556294acc51e4b3f642a8f79891d 11.61MB / 11.61MB
=> => sha256:2d50379da84735a24b4d6ed1f2d370bf1f94bf5a9ec8a4f1da85e97c30101cb7 241B / 241B
=> => extracting sha256:c3cdf40b8bda8e4ca4be0f5fa7f1d128907271efcbc72cbfc7c8b0f939ec25ea
=> => sha256:89a1f66d04aae2a80b0ab75c14c5af3476e886fa103e8a3708008f4ed636a88e 2.85MB / 2.85MB
=> => extracting sha256:a05d724777b115e78126981b762c012b90a5556294acc51e4b3f642a8f79891d
=> => extracting sha256:2d50379da84735a24b4d6ed1f2d370bf1f94bf5a9ec8a4f1da85e97c30101cb7
=> => extracting sha256:89a1f66d04aae2a80b0ab75c14c5af3476e886fa103e8a3708008f4ed636a88e
=> [2/5] WORKDIR /app
=> [3/5] COPY requirements.txt .
=> [4/5] RUN pip install --no-cache-dir -r requirements.txt
=> [5/5] COPY . .
=> exporting to image
=> => exporting layers
=> => writing image sha256:31e636fd3a8a21afce59fc485626fced5508dc41956d748475316f62ccb98f00
=> => naming to docker.io/library/python_mtsd:alpine
```

Щодо розміру образу, то він також був зменшений до 152 MB.

```
(.venv) ~/Programming/Methodologies_Lab3/Python git:[main]
sudo docker images
```

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
python_mtsd	alpine	31e636fd3a8a	8 minutes ago	152MB

5. [Додав numpy](#) як залежність + ендпоінт, який генерує 2 випадкові матриці 10x10 та множить їх.

Спочатку зберемо образ на основі **python:3.9-alpine** (зі встановленням базового образу з DockerHub). Зібрався образ за 30 секунд, а розмір має 302 MB.

```
(.venv) ~/Programming/Methodologies_Lab3/Python git:[main]
sudo docker build --pull -t python_mtsd:alpinematrix .

[+] Building 30.0s (10/10) FINISHED
```

python_mtsd	alpinematrix	20920e99911a	About a minute ago	302MB
-------------	--------------	--------------	--------------------	-------

Потім зберемо образ на основі **python:3.9-slim** (зі встановленням базового образу з DockerHub). Зібрався образ за 38.9 секунд, має розмір 377 MB.

```
(.venv) ~/Programming/Methodologies_Lab3/Python git:[main]
sudo docker build --pull -t python_mtsd:slimmatrix .

[+] Building 38.9s (10/10) FINISHED
```

```
(.venv) ~/Programming/Methodologies_Lab3/Python git:[main]
sudo docker images
```

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
python_mtsd	slimmatrix	426c6fab8794	24 seconds ago	377MB

Тож можна побачити, що образ на основі **python:3.9-alpine** швидше білдиться та менше важить, ніж образ на основі **python:3.9-slim**. Також усі додавання в код не призводили до суттєвого збільшення часу/розміру. Тільки при додаванні ще однієї залежності збільшився час/розмір. Ще порядок команд у Dockerfile важливий, якщо на початок додавати те, що змінюється частіше, то можна втратити багато переваг по часу і розміру та й в кешуванні.

Golang

1. Після аналізу та збірки проекту було написано [Dockerfile](#). Процес збірки образу (із завантаженням базового образу [golang:1.17-alpine](#)) зайняв 72.1 секунди (30.3 сек було витрачено на завантаження базового образу). Розмір образу - 947 MB

```
~/Programming/Methodologies_Lab3/Golang git:[main]
sudo docker build --pull -t goland_mtsd:latest .

[+] Building 72.1s (11/11) FINISHED
```

```
~/Programming/Methodologies_Lab3/GoLang git:[main]
sudo docker images
```

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
goland_mtsd	latest	c5697ad8f2df	About a minute ago	947MB

Для того, щоб проаналізувати вміст контейнеру треба потрапити в оболонку **sh** завдяки команді **docker run --rm -it golang_mtsd /bin/sh** та зробити перегляд робочої директорії через команду **ls**. В середині контейнеру було помічено непотрібні файли для запуску проєкту, такі як **Dockerfile** та **README.rst**, але це виправляється внесенням цих файлів у **.dockerignore** файл.

```
~/Programming/Methodologies_Lab3/Golang git:[main]
sudo docker run --rm -it goLand_mtsd /bin/sh
[sudo] password for daniorerio:
/app # ls
Dockerfile  README.rst  build      cmd        go.mod     go.sum     lib        main.go     templates
/app #
```

- Описав [Dockerfile](#), який відповідає вимогам багатоетапної збірки. Основна складність була у відсутності досвіду [Multi-stage builds](#) - документація допомогла. На першому етапі (**builder**) збирається програма fizzbuzz на основі golang:1.17-alpine, а на наступному етапі (**scratch**) створюється порожній образ, до якого копіюється лише зібраний двійковий файл fizzbuzz та файли шаблонів з директорії templates.

Повна збірка без кешування виконувалася 88 секунд

```
~/Programming/Methodologies_Lab3/Golang git:[main]
sudo docker build -t goland_mtsd:scratch -f DockerfileScratch .

[+] Building 88.0s (14/14) FINISHED
```

А з кешуванням 5.6 секунд

```
sudo docker build -t golang_mtsd:scratch -f DockerfileScratch .

[+] Building 5.6s (14/14) FINISHED
⇒ [internal] load build definition from DockerfileScratch
```

Щодо розміру, то він досяг значного зменшення 6.37MB (хоча до цього етапу було 947 MB).

```
~/Programming/Methodologies_Lab3/Golang git:[main]
sudo docker image ls
REPOSITORY          TAG          IMAGE ID          CREATED          SIZE
goland_mtsd         scratch     a62bc1e798a0     12 seconds ago  6.37MB
```

Щодо файлів для запуску, то цей образ містить тільки необхідні компоненти - зібраний двійковий файл `fizzbuzz` та файли шаблонів з директорії `templates`. Основна зручність - це легкість образу, якщо програма не вимагає додаткових середовищ виконання або взаємодії всередині контейнера. Однак базовий образ **scratch** немає навіть основних утиліт та середовища виконання, як, наприклад, `shell (/bin/sh)`. Тому виконання якихось дій всередині контейнера не є можливим. Тому можемо просто запустити контейнер на 8080 порту.

```
~/Programming/Methodologies_Lab3/Golang git:[main]
sudo docker run -p 8080:8080 --rm goland_mtsd:scratch
[sudo] password for daniorerio:
Listening on http://0.0.0.0:8080
```

- Замінив використання **scratch** на [distroless y Dockerfile](#). Було витрачено 72.3 секунди на збірку образу без кешування

```
~/Programming/Methodologies_Lab3/Golang git:[main]
sudo docker build -t goland_mtsd:distroless1.0 -f DockerfileDistroless .
[+] Building 72.3s (16/16) FINISHED
```

А з кешуванням 1.2 секунди.

```
~/Programming/Methodologies_Lab3/Golang git:[main]
sudo docker build -t goland_mtsd:distroless2.0 -f DockerfileDistroless .
[+] Building 1.2s (16/16) FINISHED
```

Щодо розміру, то він дійсно збільшився до 8.94 MB

```
~/Programming/Methodologies_Lab3/Golang git:[main]
sudo docker image ls
REPOSITORY          TAG          IMAGE ID          CREATED          SIZE
goland_mtsd         distroless1.0  b2831549ef34     4 minutes ago   8.94MB
```

Отже, завдяки багатоетапній збірці отриманий образ не містить ніяких зайвих компонентів, а тільки необхідні файли для застосунку (файл **fizzbuzz** та файли шаблонів **templates**). Ба більше, базовий образ gcr.io/distroless/static-debian11 не містить динамічні бібліотеки, оболонки чи пакетні менеджери. Через це все отриманий образ також є маленьким всього 8.94 MB. Хоча через обмежений функціонал виконання дій всередині контейнеру поки не є можливим, треба дод. налаштування. Можемо просто запустити контейнер на 8080 порту.

```
~/Programming/Methodologies_Lab3/Golang git:[main]
sudo docker run -p 8080:8080 --rm b2831549ef34
[sudo] password for daniorerio:
Listening on http://0.0.0.0:8080
```

Мова на вибір (JavaScript)

Обрав фреймворк [fastify](#) для написання базового веб-застосунку на отримання чи створення (але ніде не зберігається) співаків, наприклад, на адресу <http://0.0.0.0:8080/singers/2>. Було додано [Dockerfile](#) на основі `node:20-alpine` та додано деякі файли в [.dockerignore](#). Щодо часу першої збірки, то вона зайняла 18.9 секунд, а розмір 146 MB

```
~/Programming/Methodologies_Lab3/JavaScript git:[main]
sudo docker build -t fastify:1.0 .

[+] Building 18.9s (10/10) FINISHED
```

```
~/Programming/Methodologies_Lab3/JavaScript git:[main]
sudo docker image ls
```

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
fastify	1.0	75ce6547b0a7	6 minutes ago	146MB

Далі заради тестування було обрано достатньо великий базовий образ `node:20-bookworm`. Час на всю збірку без кешування зайняв аж цілих 112.6 секунд, а розмір 1.11GB!

```
~/Programming/Methodologies_Lab3/JavaScript git:[main]
sudo docker build -t fastify:booworm .

[+] Building 112.6s (11/11) FINISHED
```



```
~/Programming/Methodologies_Lab3/JavaScript git:[main]
sudo docker image ls
```

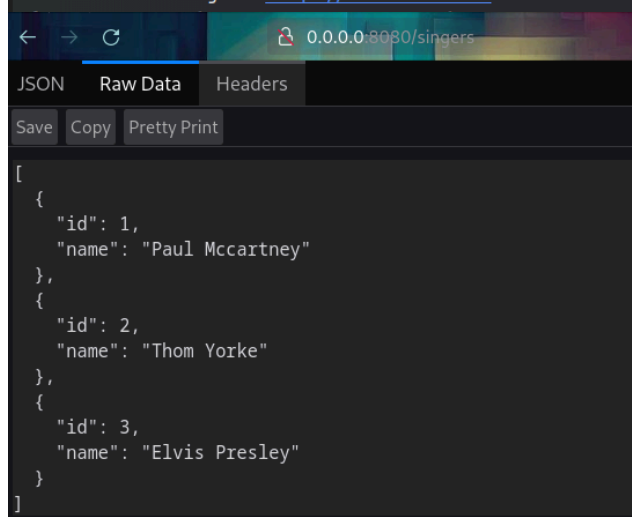
REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
fastify	booworm	e0924811ea38	7 seconds ago	1.11GB

Далі можна проаналізувати вміст контейнеру - треба потрапити в оболонку **sh** завдяки команді `sudo docker run --rm -it fastify:booworm /bin/sh` та зробити перегляд робочої директорії через команду `ls`. Можна побачити, що в контейнері немає деяких файлів: їх було занесено в [.dockerignore](#).

```
~/Programming/Methodologies_Lab3/JavaScript git:[main]
sudo docker run --rm -it fastify:booworm /bin/sh
# pwd
/app
# ls
index.js  node_modules  package-lock.json  package.json
#
```

Також можна запустити контейнер на 8080 порту, подивитися, чи працює отримання всіх співаків.

```
~/Programming/Methodologies_Lab3 git:[main]
sudo docker run -p 8080:8080 --rm fastify:booworm
[sudo] password for daniorerio:
(node:1) [FSTDEP011] DeprecationWarning: Variadic listen method is deprecated. Please use ".listen(optionsObject)" :
(Use `node --trace-deprecation ...` to show where the warning was created)
{"level":30,"time":1712430922164,"pid":1,"hostname":"ca976ae67c90","msg":"Server listening at http://0.0.0.0:8080"}
Server listening at http://0.0.0.0:8080
```



```
[
  {
    "id": 1,
    "name": "Paul Mccartney"
  },
  {
    "id": 2,
    "name": "Thom Yorke"
  },
  {
    "id": 3,
    "name": "Elvis Presley"
  }
]
```

Також виконаємо [двоетапну збірку](#). На першому етапі на базі **node:20-alpine** збілдемо застосунок, а на другому на основі

gcr.io/distroless/static-debian11 скопіюємо файли з попереднього етапу в робочу директорию. Час збірки сягає 16.6 секунд, а розмір 11.5MB

```
~/Programming/Methodologies_Lab3/JavaScript git:[main]
sudo docker build -t fastify:msb -f DockerfileMSB ./

[+] Building 16.6s (14/14) FINISHED
```

```
~/Programming/Methodologies_Lab3/JavaScript git:[main]
sudo docker image ls
```

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
fastify	msb	9751f3c8755d	12 seconds ago	11.5MB

Але gcr.io/distroless/static-debian11 не має середовище NodeJs, тому [викорисаємо node:20-slim](https://nodejs.org/en/download/package-manager/#debian-11). Можна помітити, що час збільшився, порівняно з першим еспериментом, до 32.6 секунд, а розмір до 210 MB

```
~/Programming/Methodologies_Lab3/JavaScript git:[main]
sudo docker build -t fastify:slim -f DockerfileMSB .

[+] Building 32.6s (15/15) FINISHED
```

```
~/Programming/Methodologies_Lab3/JavaScript git:[main]
sudo docker image ls
```

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
fastify	slim	2cab47aa718d	10 seconds ago	210MB

Тепер можемо запустити контейнер на 8080 порту і отримати співака за 2 індексом.

```
~/Programming/Methodologies_Lab3/JavaScript git:[main]
sudo docker run -p 8080:8080 --rm fastify:slim
(node:1) [FSTDEP011] DeprecationWarning: Variadic listen method is deprecated. Please use ".listen(optionsObject)"
(Use `node --trace-deprecation ...` to show where the warning was created)
{"level":30,"time":1712434664396,"pid":1,"hostname":"90bc6f4c3e70","msg":"Server listening at http://0.0.0.0:8080"}
Server listening at http://0.0.0.0:8080
```

JSON

Raw Data

Headers

Save

Copy

Pretty Print

```
{
  "id": 2,
  "name": "Thom Yorke"
}
```

Отже, якщо порівнювати час та розмір з минулими пунктами, то можна помітити, що образи на Js білдилися швидше та важили менше, бо менше залежностей. Однак якщо брати базовий образ `node:20-bookworm`, то вихідний образ буде дуже великим $> 1\text{GB}$. Тому краще використовувати `node:20-alpine`. Щодо багатоетапної збірки, то вона не була такою ефективною, як на прикладі Golang, бо на Js я не робив виконувальний файл + мені потрібне було середовище NodeJs, ось тому і не міг використовувати образи типу `static-debian11`. Хоча, якщо і використовувати “distroless”, то для NodeJs підійде gcr.io/distroless/nodejs20-debian12. Також і в останньому експерименті на двоетапну збірку з використанням ще й `node:20-slim` було перевищення по часу і розміру, порівняно з першим експериментом.

Висновки (уся робота)

На основі 3 блоків роботи можна виділити рекомендації для створення контейнерів:

1. краще використовувати менші базові образи, такі як `alpine`. На всіх 3 прикладах образи збиралися швидше та мали менший розмір;
2. для великих і складних застосунків краще використовувати багатоетапну збірку для зменшення розміру кінцевого образу. Можна розділити збірку, залежності та запуск на різні етапи, використовуючи різні базові образи для кожного етапу;
3. якщо не потрібні динамічні бібліотеки, оболонки чи пакетні менеджери, то краще використовувати образи “Distroless”, які будуть мати тільки необхідні компоненти для застосунку. Хоча тут треба ще дивитися на середовище, яке використовується, наприклад, для Java, NodeJs це будуть різні образи;
4. краще вносити деякі файли у **.dockerignore**, щоб непотрібні файли не знаходилися в контейнері. Це дозволить зменшити розмір контейнера;
5. порядок виконання важливий, оскільки може забезпечити ефективне кешування та швидше створення образів. Спочатку додаємо у образ те, що змінюватиметься найрідше (системні залежності, залежності проекту), а ті файли, які змінюватимуться частіше (наприклад, наш код) — додаємо

останніми. Наприклад, спочатку встановлюємо робочу директорію; копіюємо файли та залежності; потім власне встановлення залежностей; збірка програми; копіювання решти файлів; на кінцевому етапі слід встановити команду, яка буде виконуватися при запуску контейнера. Це може бути CMD або ENTRYPOINT, які запускають ваш застосунок або сервіс.