Лабораторна робота №2

Дослідження з теми “Контейнеризація”

Python Application

1. Спершу створюємо опис контейнера для застосунку. Додаємо в цей образ код застосунку, залежності, та збираємо наш образ. Необхідно виміряти:

- розмір образу
- час збірки образу

Спочатку заморожую поточні залежності проекту:

```
pip freeze > requirements.txt
```

Створюю НЕідеальний докерфайл:

```
FROM python:3.12-bookworm
WORKDIR /app
COPY . .
RUN pip install -r requirements.txt
CMD ["python", "spaceship/app.py"]
```

Перед оцінюванням ефективності завантажую базовий образ, щоб оцінювання було “чесним”:

```
docker pull python:3.12-bookworm
```

student@nodeserver3:~/devops_labs$ time docker build --no-cache -t python-lab:test-bookworm .

час збірки:

```
real    0m38.290s
user    0m0.328s
sys     0m0.357s
```

student@nodeserver3:~/devops_labs$ docker images python-lab:test-bookworm

розмір образу:

2
Вносимо зміни у spaceship/app.py:

```python
from fastapi import FastAPI
from fastapi.staticfiles import StaticFiles
from starlette.responses import FileResponse

from spaceship.config import Settings
from spaceship.routers import api, health

# some comment
def make_app(settings: Settings) -> FastAPI:
  print("Mariia Khorunzha IM-42")
  app = FastAPI(
      debug=settings.debug,
      title=settings.app_title,
      description=settings.app_description,
      version=settings.app_version,
  )
  app.state.settings = settings

  if settings.debug:
      app.mount('/static', StaticFiles(directory='build'), name='static')
  # and another comment
  app.include_router(api.router, prefix='/api', tags=['api'])
  app.include_router(health.router, prefix='/health', tags=['health'])

  @app.get('/', include_in_schema=False, response_class=FileResponse)
  async def root() -> str:
      return 'build/index.html'
```

```
return app
```

час збірки:

```
real    0m39.955s
user    0m0.264s
sys     0m0.250s
```

розмір образу:

```
IMAGE                          ID             DISK USAGE   CONTENT SIZE   EXTRA
python-lab:test-bookworm2      2f27567a22ab       1.69GB          445MB
```

3
Оптимізуємо докерфайл:

```
FROM python:3.12-bookworm

WORKDIR /app

COPY requirements.txt .

RUN pip install -r requirements.txt

COPY . .

CMD ["python", "spaceship/app.py"]
```

час збірки:

```
real    0m41.193s
user    0m0.268s
sys     0m0.259s
```

розмір образу:

```
IMAGE                  ID             DISK USAGE   CONTENT SIZE   EXTRA
python-lab:optimized   078efd6b5120       1.69GB          445MB
```

Вносимо зміни у spaceship/app.py:

```
from fastapi import FastAPI
from fastapi.staticfiles import StaticFiles
from starlette.responses import FileResponse


from spaceship.config import Settings
from spaceship.routers import api, health


# some comment
def make_app(settings: Settings) -> FastAPI:
 print("Mariia Khorunzha IM-42")
 print("Hello world!")
 app = FastAPI(
    debug=settings.debug,
    title=settings.app_title,
    description=settings.app_description,
    version=settings.app_version,
 )
 app.state.settings = settings


 if settings.debug:
    app.mount('/static', StaticFiles(directory='build'), name='static')
 # and another comment
 app.include_router(api.router, prefix='/api', tags=['api'])
 app.include_router(health.router, prefix='/health', tags=['health'])


 @app.get('/', include_in_schema=False, response_class=FileResponse)
 async def root() -> str:
    return 'build/index.html'
```

```
return app
```

час збірки:

```
real    0m1.136s
user    0m0.134s
sys     0m0.128s
```

розмір образу:

```
IMAGE                  ID             DISK USAGE   CONTENT SIZE   EXTRA
python-lab:optimized   8e18c2f342dd       1.69GB          445MB
```

Отже, по передостанньому скриншоту можна побачити, що відбулося кешування. Ось в чому справа:

- до оптимізації COPY . . копіював і стабільні файли залежностей, і код, який постійно змінюється. RUN pip install … залежав від попереднього кроку. І як результат, як тільки я міняла хоча б один символ у коді, докерфайл бачив, що шар COPY . . Змінився. Через це він анулював кеш для всіх наступних кроків, і йому доводилося заново запускати важний pip install.
- після отпимізації COPY requirements/... копію тільки файл із списком бібліотек, він змінюється рідко. RUN pip install … залежить тільки від файлу вимог. і COPY . . копіює решту коду. Тобто тепер після зміни коду докер перевіряє шари по черзі і бачить, що шар вимог не змінився, тому беремо його з кешу. pip install знову запускати теж не треба, бо вхідні дані для нього із шару 1 не змінилися. і тільки на шарі 3 докерфайл починає дійсно працювати, бо бачить, що код змінився!

4
тепер змінимо за основу проекту менший образ - alpine

змінюємо докерфайл:

```
FROM python:3.12-alpine
WORKDIR /app

COPY requirements.txt .
```

```
RUN pip install -r requirements.txt

COPY . .

CMD ["python", "spaceship/app.py"]
```

Одразу завантажую образ:

```
docker pull python:3.12-alpine
```

час:

```
real    0m36.339s
user    0m0.256s
sys     0m0.251s
```

розмір:

```
IMAGE           ID             DISK USAGE   CONTENT SIZE   EXTRA
python-lab:v5   acf3c62b4877       165MB         46.3MB
```

Отже, Docker працює так: якщо змінюється будь-який шар, то всі наступні шари автоматично анулюються (стають недійсними). Оскільки я змінила самий верхній рядок (FROM), Docker більше не може використовувати старий кеш від bookworm. Він змушений заново запустити pip install, бо ці бібліотеки тепер мають бути встановлені поверх Alpine, а не Debian, тому час збірки знов став більшим, зате розмір образу набагато менший, бо alpine легший.

Якщо ж знову змінити код і виміряти час:

```
from fastapi import FastAPI
from fastapi.staticfiles import StaticFiles
from starlette.responses import FileResponse

from spaceship.config import Settings
from spaceship.routers import api, health

# some comment
def make_app(settings: Settings) -> FastAPI:
  print("Mariia Khorunzha IM-42")
```

```
    app = FastAPI(
        debug=settings.debug,
        title=settings.app_title,
        description=settings.app_description,
        version=settings.app_version,
    )
    app.state.settings = settings

    if settings.debug:
        app.mount('/static', StaticFiles(directory='build'), name='static')
    # and another comment
    app.include_router(api.router, prefix='/api', tags=['api'])
    app.include_router(health.router, prefix='/health', tags=['health'])

    @app.get('/', include_in_schema=False, response_class=FileResponse)
    async def root() -> str:
        return 'build/index.html'

    return app
```

```
real    0m2.186s
user    0m0.157s
sys     0m0.154s
```

```
IMAGE            ID             DISK USAGE   CONTENT SIZE   EXTRA
python-lab:v5    93240a7d7438       165MB         46.3MB
```

Побачимо, що кешування знов спрацювало, і час збірки набагато зменшився.

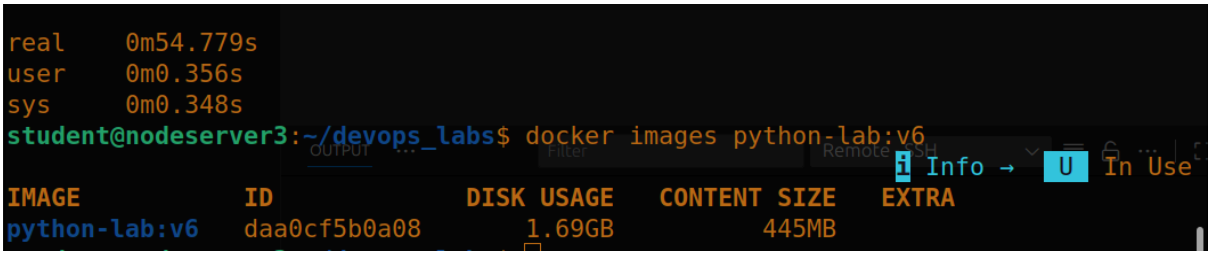
5
Тепер інсталюю numpy і заморожую поточні залежності:

```
pip install numpy
pip freeze > requirements.txt
```

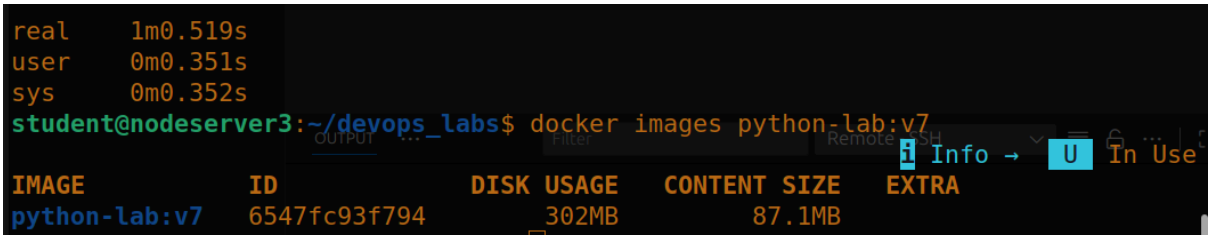
Додаю в spaceship/routers/api.py простий ендпоінт, який за допомогою доданої залежності згенерує 2 випадкові матриці 10x10 та перемножить їх між собою:

```python
from fastapi import APIRouter
import numpy as np

router = APIRouter()


@router.get('')
def hello_world() -> dict:
  return {'msg': 'Hello, World!'}

@router.get("/matrix")
def get_matrix():
  matrix_a = np.random.rand(10, 10)
  matrix_b = np.random.rand(10, 10)

  product = np.matmul(matrix_a, matrix_b)

  return {
     "matrix_a": matrix_a.tolist(),
     "matrix_b": matrix_b.tolist(),
     "product": product.tolist()
  }
```

bookworm



alpine



Образ на базі Alpine у 5 разів менший, ніж на Bookworm. Alpine програє у часі першої збірки, оскільки потребує більше зусиль для підготовки середовища та інсталяції важких бібліотек типу NumPy. Отже, для розробки (де важлива швидкість білду) зручніше Bookworm. Для продакшену (де важливий розмір і безпека) - Alpine.

Musl (Alpine) vs glibc (Debian/Ubuntu/etc)

Після виконання пунктів 1-4 з методички:

Логи:
dnsmasq[1]: reading /etc/resolv.conf
dnsmasq[1]: using nameserver 127.0.0.11#53
dnsmasq[1]: read /etc/hosts - 9 names
dnsmasq[1]: query[AAAA] myservice.internal from 172.18.0.3
dnsmasq[1]: forwarded myservice.internal to 127.0.0.11
dnsmasq[1]: reply myservice.internal is NXDOMAIN
dnsmasq[1]: query[AAAA] myservice.internal.corp from 172.18.0.3
dnsmasq[1]: forwarded myservice.internal.corp to 127.0.0.11
dnsmasq[1]: reply myservice.internal.corp is NODATA-IPv6
dnsmasq[1]: query[A] myservice.internal from 172.18.0.3
dnsmasq[1]: cached myservice.internal is NXDOMAIN
dnsmasq[1]: query[A] myservice.internal.corp from 172.18.0.3
dnsmasq[1]: config myservice.internal.corp is 10.0.0.50
dnsmasq[1]: query[AAAA] myservice.internal from 172.18.0.3
dnsmasq[1]: forwarded myservice.internal to 127.0.0.11
dnsmasq[1]: reply myservice.internal is NXDOMAIN
dnsmasq[1]: query[A] myservice.internal from 172.18.0.3
dnsmasq[1]: cached myservice.internal is NXDOMAIN
dnsmasq[1]: query[AAAA] myservice.internal from 172.18.0.3
dnsmasq[1]: cached myservice.internal is NXDOMAIN
dnsmasq[1]: query[A] myservice.internal from 172.18.0.3
dnsmasq[1]: cached myservice.internal is NXDOMAIN

Синім виділені логи з ubuntu, сірим - з alpine

Аналіз логів DNS показав різницю в роботі системних бібліотек. Bookworm (glibc) робить зайві запити: він не тільки шукає основний домен, а й намагається підставити суфікс .corp. Натомість Alpine (musl) працює інакше: він бачить крапку в назві домену, одразу вважає його повним і просто запитує IP.

Наскільки я зрозуміла, бібліотека glibc має складний механізм резолвінгу, який підтримує параметри search. І якщо коротке ім'я не знайдено, вона перебирає всі варіанти з dns-search. В Alpine використовується musl libc, спрощена для швидкості та легкості. Її логіка така: якщо в імені немає жодної крапки, вона може спробувати суфікс. Але якщо в імені є крапка (як у моєму випадку myservice.internal), вона вважає це вже "майже повним ім'ям", і не додає до нього суфікс .corp.

На практиці це може призвести до:

1. Багато сервісів усередині компаній звертаються один до одного як database.internal. На Ubuntu це працює, а після переходу на Alpine (Docker) зв'язок розривається.
2. Програми на Python/Node.js почнуть видавати помилки підключення до бази даних або API, хоча в конфігах все вказано "правильно".

Отже, при використанні Alpine в Docker-образах завжди треба вказувати повні доменні імена (наприклад, myservice.internal.corp замість myservice.internal), щоб не залежати від особливостей реалізації DNS-резолвера в системній бібліотеці.

Golang Application та Multi-stage builds

Докерфайл:

```
FROM golang:1.22-bookworm
WORKDIR /app
COPY . .
RUN go mod download
RUN go build -o main .
CMD ["./main"]
```

Час та розмір збірки:

```
real    2m24.455s
user    0m0.926s
sys     0m0.981s
student@nodeserver3:~/devops_lab2$ docker images go-heavy
                                                  i Info →   U  In Use
IMAGE             ID             DISK USAGE   CONTENT SIZE   EXTRA
go-heavy:latest   18f1f0d51f4f       1.34GB          324MB
```

Файли всередині:

```
student@nodeserver3:~/devops_lab2$ docker run --rm -it go-heavy sh
# ls -lh
total 11M
-rw-rw-r-- 1 root root 1.1K May  2 19:42 README.rst
drwxrwxr-x 2 root root 4.0K May  2 19:42 cmd
-rw-rw-r-- 1 root root  105 May  2 19:45 dockerfile
-rw-rw-r-- 1 root root  181 May  2 19:42 go.mod
-rw-rw-r-- 1 root root  74K May  2 19:42 go.sum
drwxrwxr-x 2 root root 4.0K May  2 19:42 lib
-rwxr-xr-x 1 root root  11M May  2 19:49 main
-rw-rw-r-- 1 root root  119 May  2 19:42 main.go
drwxrwxr-x 2 root root 4.0K May  2 19:42 templates
```

```
# ls /usr/local/go
CONTRIBUTING.md  README.md    api            doc     misc  test
LICENSE          SECURITY.md  bin            go.env  pkg
PATENTS          VERSION      codereview.cfg  lib     src
```

Образ виходить немалий, бо там лежить весь компілятор Go, бібліотеки Debian, кеш збірки та вихідний код. Вони для запуску не потрібні, потрібен лише один файл main.

РОбимо багатоетапну збірку, змінюємо докерфайл:

```
FROM golang:1.22-bookworm AS builder
WORKDIR /app
COPY . .
RUN go build -o main .

FROM scratch
COPY --from=builder /app/main /main
ENTRYPOINT ["/main"]
```

Час та розмір збірки:

```
real    0m20.314s
user    0m0.213s
sys     0m0.191s
student@nodeserver3:~/devops_lab2$ docker images go-scratch
                                                  i Info →   U  In Use
IMAGE                ID             DISK USAGE   CONTENT SIZE   EXTRA
go-scratch:latest    d3a5e1d908b4       16.8MB         6.01MB
```

Файли всередині:

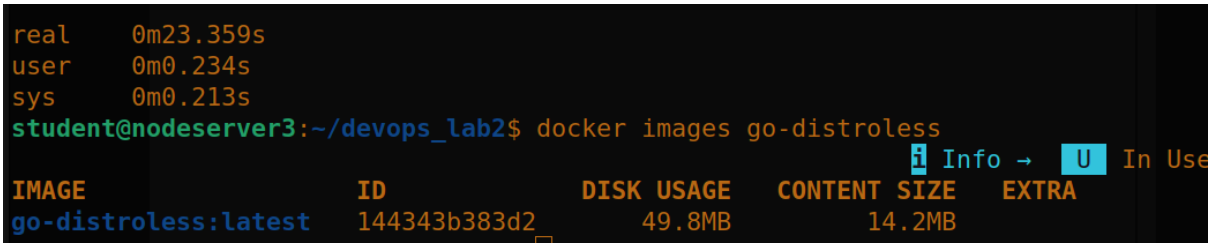
```
.dockerenv
dev/
dev/console
dev/pts/
dev/shm/
etc/
etc/hostname
etc/hosts
etc/mtab
etc/resolv.conf
main
proc/
sys/
```

Для простого бінарника Go файлів достатньо, але для запуску мого проекту - ні. І таким користуватися не дуже зручно. У scratch немає навіть команди ls, sh чи cat, типу нема bash. Всередині виконувати будь-які дії неможливо - там порожнеча, крім мого файлу.

Використаємо образ з distroless, перепишу докерфайл:

```
FROM golang:1.22-bookworm AS builder
WORKDIR /app
COPY . .
RUN go build -o main .

FROM gcr.io/distroless/base-debian12
COPY --from=builder /app/main /main
ENTRYPOINT ["/main"]
```

Час та розмір збірки:



Файли всередині:

student@nodeserver3:~/devops_lab2$ tar -tf distroless_files.tar
.dockerenv
bin/
boot/
dev/
dev/console
dev/pts/
dev/shm/
etc/
etc/debian_version
etc/default/
etc/dpkg/
etc/dpkg/origins/
etc/dpkg/origins/debian
etc/ethertypes
etc/group
etc/host.conf
etc/hostname
etc/hosts
etc/issue
etc/issue.net
etc/ld.so.conf.d/
etc/ld.so.conf.d/x86_64-linux-gnu.conf
etc/mime.types
etc/mtab
etc/nsswitch.conf
etc/os-release
etc/passwd
etc/profile.d/
etc/protocols
etc/resolv.conf
etc/rpc
etc/services

etc/skel/
etc/ssl/
etc/ssl/certs/
etc/ssl/certs/ca-certificates.crt
etc/update-motd.d/
etc/update-motd.d/10-uname
home/
home/nonroot/
lib/
lib/x86_64-linux-gnu/
lib/x86_64-linux-gnu/ld-linux-x86-64.so.2
lib/x86_64-linux-gnu/libBrokenLocale.so.1
lib/x86_64-linux-gnu/libanl.so.1
lib/x86_64-linux-gnu/libc.so.6
lib/x86_64-linux-gnu/libc_malloc_debug.so.0
lib/x86_64-linux-gnu/libdl.so.2
lib/x86_64-linux-gnu/libm.so.6
lib/x86_64-linux-gnu/libmemusage.so
lib/x86_64-linux-gnu/libmvec.so.1
lib/x86_64-linux-gnu/libnsl.so.1
lib/x86_64-linux-gnu/libnss_compat.so.2
lib/x86_64-linux-gnu/libnss_dns.so.2
lib/x86_64-linux-gnu/libnss_files.so.2
lib/x86_64-linux-gnu/libnss_hesiod.so.2
lib/x86_64-linux-gnu/libpcprofile.so
lib/x86_64-linux-gnu/libpthread.so.0
lib/x86_64-linux-gnu/libresolv.so.2
lib/x86_64-linux-gnu/librt.so.1
lib/x86_64-linux-gnu/libthread_db.so.1
lib/x86_64-linux-gnu/libutil.so.1
lib64/
lib64/ld-linux-x86-64.so.2
main
proc/
root/
run/
sbin/
sys/
tmp/
usr/
usr/bin/
usr/games/

usr/include/
usr/lib/
…
і ще куча файлів далі

Для запуску готової програми main не потрібен вихідний код (.go файли), не потрібен компілятор і не потрібен git. Тут проблема в тому, що зараз образ містить забагато зайвого, і займає місце на сервері.

Отже, багатоетапна збірка (особливо з нуля) дозволяє зменшити розмір образу в десятки разів, видаляючи все зайве: компілятори, вихідний код та системні утиліти. Також Образи scratch та distroless безпечніші, бо в них немає оболонки sh, і типу навіть якщо хтось отримає доступ до контейнера, вони не зможуть запустити жодну команду. Повторна збірка (кешування) займає лічені секунди, це добре швидкої розробки. Ну і виходіть, що distroless - кращий із запропонованих варіантів, бо він містить необхідні для Go сертифікати SSL та часові пояси, але залишається легким (хоча кнш треба дивитись по ситуації, кому що важливіше).

Хорунжа Марія ІМ-42