

# Raspodijeljeni sustavi (RS)

**Nositelj:** doc. dr. sc. Nikola Tanković

**Asistent:** Luka Blašković, mag. inf.

**Ustanova:** Sveučilište Jurja Dobrile u Puli, Fakultet informatike u Puli



Fakultet informatike u Puli

## (7) Kontejnerizacija i Load balancing

#7

RS

Naučili smo kako definirati asinkrone mikroservise kroz razvojne okvire poput aiohttp i FastAPI. Jednom kad imamo robusne mikroservise, sljedeći korak je njihovo raspoređivanje i upravljanje resursima, bilo na lokalnom ili u proizvodnjkom okruženju. Kontejnerizacija predstavlja tehnologiju koja omogućuje doslovno pakiranje aplikacija i svih njenih ovisnosti u jednu samostalnu i lako-prenosivu cjelinu, tzv. kontejner (*eng. Container*). Kontejneri osiguravaju konzistentnost i predvidljivost ponašanja aplikacija u različitim okruženjima, smanjujući mogućnost grešaka. S druge strane, uravnotežavanje opterećenja (*eng. Load balancing*) osigurava ravnomjernu raspodjelu zahtjeva između više mikroservisa odnosno instanci kontejnera. Kombinacija ovih dvaju tehnologija omogućuje skaliranje i optimizaciju modernih softverskih rješenja u dinamičnim okruženjima, kao što je to računarstvo u oblaku (*eng. Cloud computing*).

**Posljednje ažurirano: 22.1.2025.**

- to be added: zadaci za vježbu

## Sadržaj

- [Raspodijeljeni sustavi \(RS\)](#)
- [\(7\) Kontejnerizacija i Load balancing](#)
  - [Sadržaj](#)
- [1. Kontejnerizacija mikroservisa \(Docker\)](#)
  - [1.1 Instalacija Docker platforme](#)
  - [1.2 Dockerfile](#)
    - [1.2.1 Osnovne naredbe u Dockerfileu](#)
  - [1.3 Kontejnerizacija osnovnog Python programa](#)
  - [1.4 Kontejnerizacija aiohttp mikroservisa](#)
    - [1.4.1 Mapiranje portova](#)

- [1.5 Tablica osnovnih Dockerfile naredbi](#)
- [1.6 Tablica osnovnih Docker naredbi](#)
- [1.7 Kontejnerizacija FastAPI mikroservisa](#)
  - [1.7.1 Implementacija mikroservisa](#)
  - [1.7.2 Kontejnerizacija mikroservisa](#)
- [1.8 Zadaci za vježbu: Kontejnerizacija mikroservisa](#)
- [2. Docker Compose](#)
  - [2.1 Kako spakirati više mikroservisa u jednu cjelinu](#)
    - [2.1.1 Sintaksa docker-compose.yml datoteke](#)
  - [2.2 Interna komunikacija mikroservisa](#)
  - [2.3 Varijable okruženja u Dockeru](#)
  - [2.4 Zadaci za vježbu: Docker Compose](#)
- [3 Load balancing \(nginx\)](#)
  - [3.1 Horizontalno skaliranje koristeći samo Docker Compose](#)

# 1. Kontejnerizacija mikroservisa (Docker)

**Docker** je popularna platforma otvorenog koda koja se koristi za automatizaciju razvoja i isporuke koristeći tehnologiju kontejnerizacije (*eng. Containerization*). U računarstvu, kontejnerizacija predstavlja vrstu virtualizacije na razini operacijskog sustava koja omogućuje pokretanje i izvršavanje aplikacija u izoliranim okruženjima zvanim **kontejneri** (*eng. Container*).



**Kontejner** (*eng. Container*) je standardizirana, samostalna i izolirana softverska jedinica koja sadrži sve potrebne datoteke, biblioteke, konfiguracije i druge ovisnosti potrebne za pokretanje aplikacije. Kontejneri služe za brzo pakiranje i distribuciju aplikacija u različitim okruženjima, primjerice na razvojnom računalu, testnom poslužitelju ili proizvodnjkom sustavu, ili različitim operacijskim sustavima.

U usporedbi s virtualnim mašinama (*eng. Virtual Machine (VM)*), kontejneri su znatno memorijski efikasniji, brže se pokreću te su portabilni. Međutim, kontejneri pokrenuti na našim računalima (ili u Cloud okruženju) direktno ovise o operacijskom sustavu domaćina te dijele resurse s njim, što ne predstavlja potpuni izolacijski sloj kao kod virtualnih mašina koje imaju vlastiti operacijski sustav, programe, aplikacije itd.

Ipak, upravo ovo dijeljenje kernela operacijskog sustava domaćina omogućuje brže pokretanje i manju potrošnju resursa, što je čini idealnom tehnologijom za razvoj i isporuku mikroservisa.

## 1.1 Instalacija Docker platforme

Kako bi nastavili, potrebno je prvo instalirati Docker platformu koja dolazi s grafičkim korisničkim sučeljem (**Docker Desktop**) za sve operacijske sisteme.

- [Docker Desktop za Windows](#)
- [Docker Desktop za macOS](#)
- [Docker Desktop za Linux](#)

Ako ste na Windows OS-u, Docker Desktop zahtjeva instalaciju **WSL-2** (Windows Subsystem for Linux) koji se može instalirati preko PowerShell naredbe:

```
wsl --install
```

Dodatno, je potrebno omogućiti **virtualizaciju** za Windows računala.

Ovisno o proizvođaču maticne ploče, postupak se razlikuje, ali BIOS-u se obično pristupa pritiskom tipke **F2**, **F10**, **F12** ili **DEL** na samom pokretanju računala (**ovo se ne radi za macOS računala**).

Najbolji način je pretražiti na internetu kako pristupiti BIOS-u za vaš model računala. Nakon toga pratite upute na linku iznad, ovisno o operacijskom sustavu.

## System requirements

### Tip

#### Should I use Hyper-V or WSL?

Docker Desktop's functionality remains consistent on both WSL and Hyper-V, without a preference for either architecture. Hyper-V and WSL have their own advantages and disadvantages, depending on your specific set up and your planned use case.

WSL 2 backend, x86\_64   Hyper-V backend, x86\_64   WSL 2 backend, Arm (Beta)

- WSL version 1.1.3.0 or later.
- Windows 11 64-bit: Home or Pro version 22H2 or higher, or Enterprise or Education version 22H2 or higher.
- Windows 10 64-bit: Minimum required is Home or Pro 22H2 (build 19045) or higher, or Enterprise or Education 22H2 (build 19045) or higher.
- Turn on the WSL 2 feature on Windows. For detailed instructions, refer to the [Microsoft documentation](#).
- The following hardware prerequisites are required to successfully run WSL 2 on Windows 10 or Windows 11:
  - 64-bit processor with [Second Level Address Translation \(SLAT\)](#)
  - 4GB system RAM
  - Enable hardware virtualization in BIOS. For more information, see [Virtualization](#).

For more information on setting up WSL 2 with Docker Desktop, see [WSL](#).

### Note

Docker only supports Docker Desktop on Windows for those versions of Windows that are still within [Microsoft's servicing timeline](#). Docker Desktop is not supported on server versions of Windows, such as Windows Server 2019 or Windows Server 2022. For more information on how to run containers on Windows Server, see [Microsoft's official documentation](#).

### Important

To run Windows containers, you need Windows 10 or Windows 11 Professional or Enterprise edition. Windows Home or Education editions only allow you to run Linux containers.

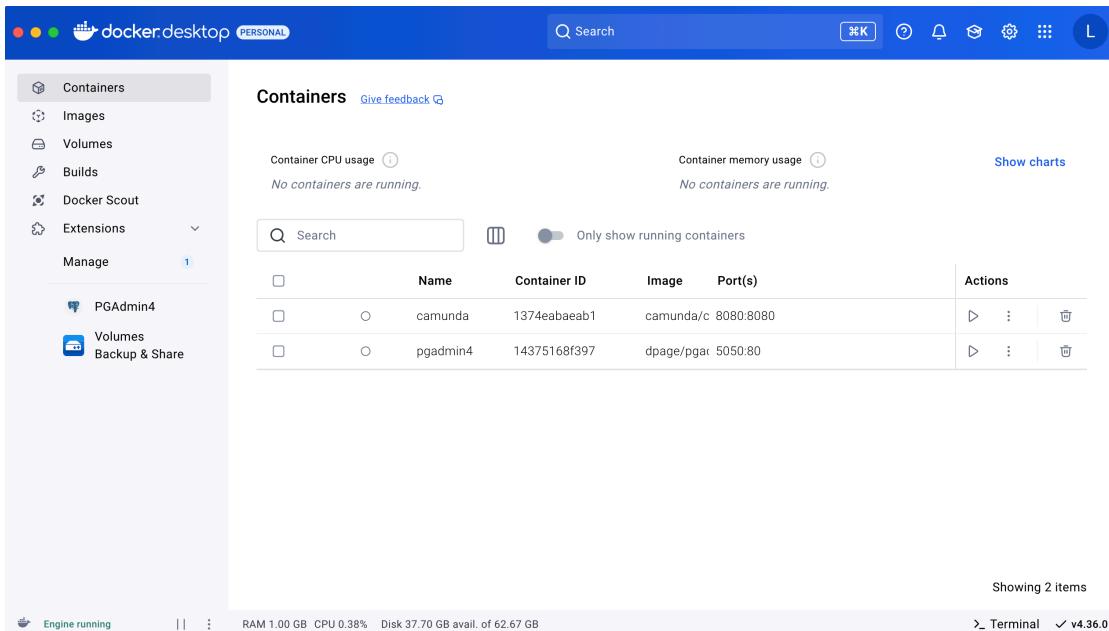
Na Windowsu je moguće koristiti **WSL** (Windows Subsystem for Linux) ili **Hyper-V** platformu za virtualizaciju, detaljne upute: <https://docs.docker.com/desktop/setup/install/windows-install/>

Docker je moguće koristiti i na **Linux** (dostupno za: Ubuntu, Debian, RHEL, Fedora) i **macOS** (dostupno za: Apple silicon, Intel chip) operacijskim sustavima bez dodatnih postavki. [Na Linuxu možete instalirati Docker i bez grafičkog sučelja preko terminala](#), međutim za početnike je preporuka instalirati grafičko sučelje - **Docker Desktop**.

Nakon što ste uspješno instalirati **Docker Desktop**, provjerite je li uspješno instaliran preko naredbe:

```
docker --version
```

Pokrenite Docker Desktop aplikaciju i prijavite se s vašim Docker računom. Ako nemate Docker račun, možete ga besplatno kreirati na [Docker Hub-u](#).



## Grafičko sučelje Docker Desktop aplikacije

Grafičko sučelje Docker Desktop aplikacije sastoji se od nekoliko osnovnih elemenata:

1. **Container** - prikaz svih pokrenutih kontejnera (eng. *Docker container*). Docker Container je svaka instanca izgrađenog Docker predloška (*image*) koja se pokreće u izoliranom okruženju
2. **Images** - prikaz svih preuzetih Docker predložaka (eng. *Docker image*). Docker Image je nepromjenjivi predložak za definiranje i pokretanje kontejnera.
3. **Volumes** - prikaz svih Docker "volumena" (eng. *Docker volumes*). Docker Volume koristi se za trajno pohranjenje podataka, obzirom da se podaci unutar kontejnera brišu prilikom gašenja kontejnera.
4. **Builds** - prikaz svih provedenih Docker "buildova" (eng. *Docker build*). Ovdje su pohranjeni svi Docker buildovi koji su se izvršavali na vašem računalu.
5. **Docker Scout** - napredna analiza pohranjenih docker predložaka, u svrhu pronalaska potencijalnih ranjivosti (eng. *vulnerabilities*).
6. **Extensions** - dodatne ekstenzije za Docker Desktop aplikaciju. Za sada nam nisu potrebne.

U pravilu, za sada će nam najzanimljiviji biti `Container` i `Images` tabovi.

U nastavku ove skripte, za Docker Images neće se koristiti termin Docker "slika" već **predložak**.

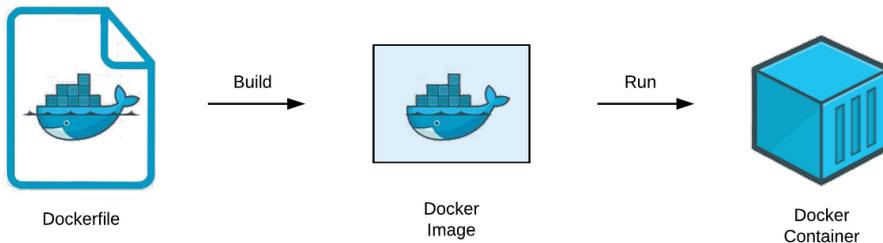
## 1.2 Dockerfile

**Dockerfile** je tekstualna datoteka koju koristimo za definiranje predložaka kontejnera. **Predložak** (Docker image) je ništa drugo nego **tekstualna datoteka koja se sastoji od niza naredbi** koje se izvršavaju prilikom izgradnje kontejnera.

`Dockerfile` može biti vrlo jednostavan, ali i vrlo složen, ovisno o mikroservisu kojeg definiramo, ali i o ovisnostima koje ima (npr. baza podataka, vanjski servisi, itd.).

U kontekstu ovog kolegija, mi ćemo naučiti kako definirati Dockerfileove za naše mikroservise, koje smo definirali u Pythonu, konkretno koristeći `FastAPI` i `aiohttp`, ali to može biti bilo koji drugi jezik ili tehnologija, ili oblik softverskog rješenja (ne mora biti API).

Upravo je to i **glavni cilj Docker platforme** - omogućiti jednostavno pakiranje i distribuciju bilo koje aplikacije, neovisno o njenim karakteristikama, ovisnostima ili tehnologijama koje koristi.



**Dockerfile** definira **predložak** kontejnera, a **kontejner** je instanca tog predloška koja se pokreće u izoliranom okruženju

Dockerfile definiramo **doslovnim nazivom datoteke**: `Dockerfile` (bez ekstenzije), a on se nalazi u korijenskom direktoriju mikroservisa.

Sintaksa *Dockerfile* naredbe:

```
# komentar
INSTRUCTION argument
```

- `INSTRUCTION` - naredba koja se izvršava prilikom izgradnje Docker predložka
- `argument` - argument naredbe

### 1.2.1 Osnovne naredbe u Dockerfileu

#### FROM

- Svrha: definira bazni predložak (tzv. **base image**) na kojem se gradi naš predložak
- Svaki Docker predložak mora početi s `FROM` naredbom, dakle to je prva naredba u Dockerfileu

```
FROM <image>:<tag>
```

Uobičajeno je koristiti službene verzije predložaka koje su dostupne na [Docker Hubu](#), konkretno za Python ih ima jako puno, npr. `python:3`:

```
# koristi Python 3 kao bazni predložak
FROM python:3
```

#### WORKDIR

- postavlja radni direktorij **unutar datotečnog sustava (eng. File system)** kontejnera
- sve naredbe nakon `WORKDIR` naredbe izvršavaju odnose se na taj direktorij, odnosno svi relativni putevi (*eng. path*) odnose se na taj direktorij

```
WORKDIR <path>
```

Primjer: postavljanje radnog direktorija na `/app` znači da će se sve naredbe koje slijede izvršavati unutar `/app` direktorija kontejnera.

```
# postavlja radni direktorij na /app
WORKDIR /app
```

**COPY**

- kopira datoteke i/ili direktorije **s računala domaćina** (eng. host) **u datotečni sustav kontejnera**
- naredba prima dva argumenta: `<src>` putanju do datoteke/direktorija na računalu domaćina i `<dest>` putanju do datoteke/direktorija unutar kontejnera
- ako želimo kopirati sve datoteke/direktorije iz trenutnog direktorija, možemo koristiti točku `.` kao `<src>`

```
# kopira datoteku app.py iz trenutnog direktorija (<src>) u destinacijski direktorij
kontejnera (<dest>
COPY <src> <dest>
```

Primjer: kopiranje ukupnog sadržaja iz trenutnog direktorija u `/app` direktorij kontejnera:

```
# kopira sve datoteke i direktorije iz trenutnog direktorija u /app direktorij kontejnera
COPY . /app
```

**CMD**

- definira **bilo koju naredbu** koja će se izvršiti **prilikom pokretanja kontejnera**
- može se koristiti **samo jednom** u Dockerfileu
- tipično se koristi za pokretanje aplikacije prilikom pokretanja kontejnera.
- naredba se **ne pokreće prilikom stvaranja predloška, već prilikom pokretanja kontejnera**

```
# pokreće aplikaciju prilikom pokretanja kontejnera
CMD ["executable", "arg1", "arg2"]
```

Primjer: pokretanje Python aplikacije `app.py` prilikom pokretanja kontejnera:

```
# pokreće Python aplikaciju prilikom pokretanja kontejnera
CMD ["python", "app.py"]
```

**RUN**

- izvršava naredbu **prilikom izgradnje Docker predloška**

- uobičajeno se koristi za instalaciju ovisnosti, konfiguraciju okruženja i sl.
- rezultati izvršene naredbe se pohranjuju u predložak, odnosno postaju dostupni prilikom pokretanja kontejnera
- u usporedbi s naredbom `CMD`, `RUN` se izvršava prilikom izgradnje predloška, dok se `CMD` izvršava prilikom pokretanja kontejnera

```
RUN <command>
```

**Primjer:** instalacija Python paketa `requests` prilikom izgradnje predloška:

```
# instalira Python paket requests prilikom izgradnje predloška
RUN pip install requests
```

#### EXPOSE

- služi za dokumentaciju porta na kojem će kontejner slušati u mreži.
- **neće otvoriti port na hostu**, već samo **dokumentira** koji port koristi kontejner

```
EXPOSE <port>
```

**Primjer:** dokumentiranje porta `8080`

```
# dokumentira port 8080 na kojem će kontejner slušati
EXPOSE 8080
```

Dakle, osnovne naredbe su `FROM`, `WORKDIR`, `COPY`, `CMD`, `RUN` i `EXPOSE`. Krenut ćemo s jednostavnim primjerima koji koriste samo ove naredbe.

## 1.3 Kontejnerizacija osnovnog Python programa

[Docker Hub](#) je servis koji omogućuje preuzimanje gotovih predložaka (**bazni predlošci**), ali i dijeljenje vlastitih. Na njemu možete pronaći veliki broj gotovih Docker predložaka koje možemo koristiti kao bazne (u svrhu definicije vlastitog predloška) ili kao gotove servise (npr. baze podataka, AI modele, mikroservise, desktop aplikacije ili bilo što drugo).

Međutim, mi ćemo koristiti osnovni Python 3 Dockerfile koji možemo jednostavno izgraditi kloniranjem `python:3` predloška.

Zamislimo da radimo na jednostavnom Python programu koji ispisuje "Hello, World!" poruku. Naš Python program `app.py` izgleda ovako:

```
# app.py
if __name__ == '__main__':
    print("Hello, World!")
```

Program pokrećemo jednostavno naredbom `python app.py` u terminalu.

Kako bi razumjeli kako Docker radi, prvo ćemo običnim tekstom napisati "niz naredbi" koji ćemo potom preslikati u odgovarajuće Docker naredbe.

1. Prvo kloniramo postojeći Python 3 predložak koji će biti predložak za naš kontejner.
2. Zatim definiramo radni direktorij unutar kontejnera gdje će se naša aplikacija pokrenuti, uobičajeno je to `/app`.
3. Kopiramo datoteku `app.py` s našeg računala u radni direktorij kontejnera.
4. Definiramo naredbu koja će se izvršiti prilikom pokretanja kontejnera, u našem slučaju to je `python app.py`.

Kreirajte novu datoteku `Dockerfile` u korijenskom direktoriju vašeg Python programa i unesite sljedeće naredbe koje preslikavaju tekst iznad:

```
# 1. Prvo kloniramo postojeći Python 3 predložak koji će biti predložak za naš kontejner.
FROM python:3

# 2. zatim definiramo radni direktorij unutar kontejnera gdje će se naša aplikacija
# pokrenuti, uobičajeno je to `/app`.

WORKDIR /app

# 3. Kopiramo datoteku `app.py` s našeg računala u radni direktorij kontejnera.

COPY app.py /app

# 4. Definiramo naredbu koja će se izvršiti prilikom pokretanja kontejnera, u našem
# slučaju to je `python app.py`.

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

Brisanjem komentara, `Dockerfile` svodimo na sljedeće:

```
FROM python:3
WORKDIR /app
COPY app.py /app
CMD ["python", "app.py"]
```

Struktura direktorija bi trebala izgledati ovako:

```
.
├── Dockerfile
└── app.py
```

`Dockerfile` dodajemo u korijenski direktorij našeg Python programa

Kako bismo **izgradili predložak** (eng. *build*) iz definiranog Dockerfilea, koristimo naredbu `docker build -t <ime>:<verzija> .`:

- opcionalnom zastavicom `-t` možemo odrediti ime i verziju našeg predložka
- točka `.` označava trenutni direktorij gdje se nalazi Dockerfile (pazite da se u terminalu nalazite u direktoriju gdje se nalazi Dockerfile)

```
cd /putanja/do/direktorija/sa/Dockerfileom
docker build -t hello-world:1.0 .
```

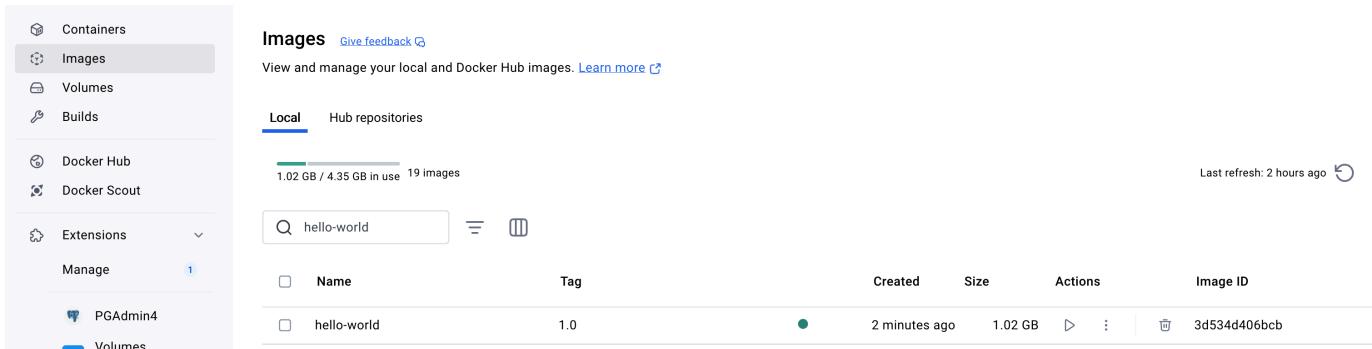
Čitaj: "izgradi Docker predložak s imenom `hello-world` i verzijom `1.0` na temelju Dockerfilea iz trenutnog direktorija"

Ako dobijete grešku prilikom izgradnje: `ERROR: Cannot connect to the Docker daemon at unix:///Users/lukablaskovic/.docker/run/docker.sock. Is the docker daemon running?`, to znači da Docker daemon nije pokrenut. Pokrenite Docker Desktop aplikaciju i pokušajte ponovno.

Izgradnja Docker predložka potrajat će neko vrijeme budući da je prvi korak preuzimanje i priprema baznog predložka `python:3`.

- nakon što se izgradi predložak, možete ga vidjeti u Docker Desktop aplikaciji pod tabom `Images`.

**Jednom kad je predložak izgrađen**, otvorite **Docker Desktop** i provjerite je li vaš predložak uspješno izgrađen u tabu `Images`.



Vidimo da je predložak `hello-world:1.0` uspješno izgrađen i ima oko 1GB, to je zato što je bazni predložak `python:3` dosta velik!

Kontejner možemo pokrenuti odabirom `Actions -> Run` ili preko terminala naredbom `docker run <ime>:<verzija>`:

```
docker run hello-world:1.0
```

**Napomena:** Naredbu je moguće pokrenuti bilo kojem terminalu, ne samo u terminalu gdje se nalazite u direktoriju s Dockerfileom.

Pokretanjem kontejnera trebali biste vidjeti ispis "Hello, World!" poruke u terminalu, odnosno u Docker Desktop aplikaciji u tabu `Container`.

Name	Container ID	Image	Port(s)	Actions
dreamy_montalcini	2d6efdc50d9b	hello-world:1.0		

Kontejner `hello-world:1.0` je uspješno pokrenut i ispisuje "Hello, World!" poruku

Pokretanjem kontejnera na ovaj način, Docker automatski dodjeljuje **naziv** i **ID kontejnera**.

## 1.4 Kontejnerizacija aiohttp mikroservisa

Na ovaj način možemo kontejnerizirati bilo koji Python program koji se sinkrono izvršava. Međutim, kako bismo kontejnerizirali asinkroni mikroservis, poput aiohttp mikroservisa koji smo imali priliku razvijati na prethodnim vježbama, **potrebit će malo drugačije pristupi izradi Dockerfilea**.

U ovom primjeru, kontejnerizirat ćemo jednostavan aiohttp mikroservis koji sadrži dva endpointa: `GET /proizvodi` i `POST /proizvodi`.

Napraviti ćemo novi direktorij `aiohttp-microservice` u kojem ćemo kreirati novi Python program `app.py` koji sadrži aiohttp mikroservis:

```
mkdir aiohttp-microservice
cd aiohttp-microservice
```

Budući da koristimo aiohttp, potrebno je instalirati ovaj paket u virtualno okruženje:

- navodimo verziju Pythona (3.11)

```
conda create -n aiohttp-microservice python=3.11
conda activate aiohttp-microservice
```

Instalirajte aiohttp paket:

```
pip install aiohttp
```

Mikroservis ćemo definirati u datoteci `app.py`:

- `GET /proizvodi` - vraća listu proizvoda
- `POST /proizvodi` - dodaje novi proizvod
- podaci su pohranjeni u listi `proizvodi`
- poslužitelj sluša na portu `8080`

```
import asyncio
from aiohttp import web

proizvodi = [
    {"id": 1, "naziv": "Laptop", "cijena": 1500},
    {"id": 2, "naziv": "Miš", "cijena": 20},
    {"id": 3, "naziv": "Tipkovnica", "cijena": 50},
    {"id": 4, "naziv": "Monitor", "cijena": 300},
    {"id": 5, "naziv": "Slušalice", "cijena": 100},
]

app = web.Application()

async def get_proizvodi(request):
    return web.json_response(proizvodi)
```

```

async def add_proizvod(request):
    data = await request.json()

    if data["naziv"] in [proizvod["naziv"] for proizvod in proizvodi]:
        return web.json_response({"error": "Proizvod već postoji!"}, status=400)

    proizvod = {
        "id": proizvodi[-1]["id"] + 1,
        "naziv": data['naziv'],
        "cijena": data['cijena']
    }
    proizvodi.append(proizvod)
    return web.json_response(proizvod)

app.router.add_routes([
    web.get('/proizvodi', get_proizvodi),
    web.post('/proizvodi', add_proizvod)
])

web.run_app(app, host='localhost', port=8080)

```

Napravite novu datoteku `Dockerfile` u korijenskom direktoriju `aiohttp-microservice`.

**Sada je naš program je složeniji**, imamo asinkroni mikroservis koji sluša na portu `8080`, stoga je potrebno definirati nekoliko dodatnih naredbi u Dockerfileu. Osim toga, imamo i ovisnost o `aiohttp` paketu, stoga je potrebno instalirati ovaj paket prilikom izgradnje predloška.

Moguće je iskoristiti naredbu `RUN` za instalaciju paketa, primjerice:

```
RUN pip install aiohttp
```

Međutim to nije uobičajeno raditi, obzirom da **stvarni mikroservisi imaju često puno više od jedne ovisnosti**. Uz to, na ovaj način ne navodimo direktno o kojoj se verziji biblioteke radi, što može dovesti do problema u budućnosti prilikom ažuriranja međuovisnosti paketa.

Bolja opcija je izlistati **sve ovisnosti** koje koristi naš mikroservis te ih instalirati jednom `RUN` naredbom.

**Ovisnosti je uobičajeno definirati u posebnoj datoteci:** `requirements.txt`

To možemo napraviti naredbom `pip freeze` koja će nam u terminal izlistati **sve pakete** koje koristi trenutno aktivno virtualno okruženje i **njihove verzije**:

```
aiohappyeyeballs==2.4.4
aiohttp==3.11.11
aiosignal==1.3.2
attrs==24.3.0
frozenlist==1.5.0
idna==3.10
multidict==6.1.0
propcache==0.2.1
setuptools==75.1.0
wheel==0.44.0
yarl==1.18.3
```

Možemo ih kopirati u ručno izrađenu datoteku `requirements.txt`, ili možemo koristiti naredbu `pip freeze > requirements.txt` koja će ih automatski zapisati u datoteku tog naziva.

Struktura direktorija bi trebala izgledati ovako:

```
.
├── Dockerfile
└── app.py
└── requirements.txt
```

Sada ćemo uzeti prethodni `Dockerfile` i prilagoditi ga za naš `aiohttp` mikroservis:

```
# Dockerfile za osnovni Python program
FROM python:3
WORKDIR /app
COPY app.py /app
CMD [ "python", "app.py" ]
```

Prvo ćemo zamijeniti `python:3` bazni predložak s `python:3.11`, kako bi se poklapao s verzijom Pythona koju koristimo. Osim toga, možemo koristiti neki neku od službenih distribucija Pythona koje su memorijski efikasnije, npr. `python:3.11-slim`:

```
FROM python:3.11-slim
```

2. korak je postavljanje **radnog direktorija u kontejneru** na `/app`:

```
WORKDIR /app
```

3. Kako sad osim `app.py` imamo i `requirements.txt`, potrebno je kopirati oba u radni direktorij kontejnera. Za to smo rekli da koristimo `COPY` naredbu s točkom `.` za `<src>`

```
# kopiraj sve datoteke iz trenutnog direktorija u /app direktorij kontejnera
COPY . /app
```

4. Sada ćemo instalirati sve ovisnosti iz `requirements.txt` datoteke. To ćemo napraviti naredbom `RUN pip install -r requirements.txt`:

- kada ne bismo koristili zastavicu `-r`, `pip` bi pokušao instalirati paket `requirements.txt` iz PyPi repozitorija, što nije ono što želimo

```
# instaliraj sve ovisnosti iz requirements.txt datoteke
RUN pip install -r requirements.txt
```

5. Iako je već u servisu definiran port `8080`, dobra praksa je dokumentirati ga koristeći naredbu `EXPOSE`:

```
# dokumentiraj port 8080
EXPOSE 8080
```

6. Na kraju, definiramo naredbu koja se koristi za pokretanje mikroservisa, u ovom slučaju ista je kao i prije.

```
# pokreće Python aplikaciju prilikom pokretanja kontejnera
CMD ["python", "app.py"]
```

Konačni `Dockerfile` izgleda ovako:

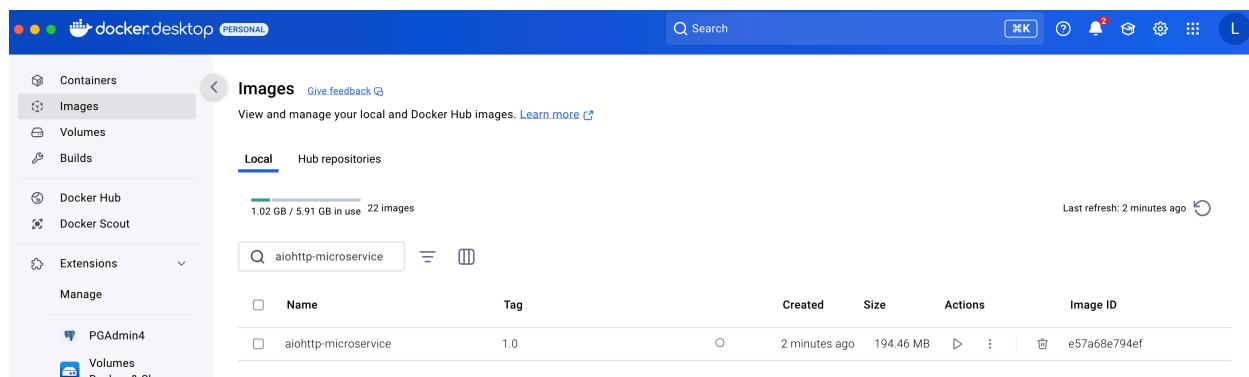
```
FROM python:3.11-slim
WORKDIR /app
COPY . /app
RUN pip install -r requirements.txt
EXPOSE 8080
CMD ["python", "app.py"]
```

Navigirat ćemo u direktorij `aiohttp-microservice` i izgradit ćemo predložak `aiohttp-microservice:1.0`:

```
docker build -t aiohttp-microservice:1.0 .
```

U terminalu možete vidjeti kako se izgrađuje predložak u 4 koraka:

1. Preuzimanje baznog predloška `python:3.11-slim`
2. Postavljanje radnog direktorija na `/app`
3. Kopiranje datoteka iz trenutnog direktorija u kontejnerski `/app`
4. Instalacija ovisnosti iz `requirements.txt`



Otvorite Docker desktop i provjerite je li predložak uspješno izgrađen.

Vidimo da je predložak `aiohttp-microservice:1.0` uspješno izgrađen i zauzima znatno manje memorije (~200MB) obzirom da smo koristili `slim` veziju za bazni predložak.

Kontejner možemo pokrenuti naredbom:

```
docker run aiohttp-microservice:1.0
```

i to radi!

The screenshot shows the Docker Desktop interface. The left sidebar has 'Containers' selected. The main area displays the 'Containers' dashboard with CPU and memory usage statistics. Two containers are listed in the table:

Name	Container ID	Image	Port(s)	Actions
dreamy_montalcini	2d6efdc50d9b	hello-world:1.0		⋮
vigorous_kilby	7c83edb9d790	aiohttp-microservice:1.0		⋮

## 1.4.1 Mapiranje portova

Naredbom `docker ps` možemo vidjeti sve pokrenute kontejnere:

```
docker ps
```

Ispisuje aktivne kontejnere:

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS
PORTS	NAMES			
a604911ac56a	aiohttp-microservice:1.0	"python app.py"	2 seconds ago	Up 2 seconds
8080/tcp	trusting_spence			

Oznake u ispisu:

- `CONTAINER ID` - jedinstveni identifikator kontejnera
- `IMAGE` - ime i verzija predloška
- `COMMAND` - naredba koja se izvršava prilikom pokretanja kontejnera (definirana u `CMD` naredbi)
- `CREATED` - vrijeme kada je kontejner pokrenut
- `STATUS` - status kontejnera (npr. `Up 2 seconds` znači da je kontejner pokrenut prije 2 sekunde)
- `PORTS` - portovi na kojima kontejner sluša
- `NAMES` - naziv kontejnera

Vidimo da je kontejner pokrenut i sluša na portu `8080`. Međutim, ako pokušamo pristupiti `localhost:8080/proizvodi` u web pregledniku ili kroz HTTP klijent pošaljemo zahtjev, dobit ćemo grešku povezivanja, što mislite zašto? 😕

► Spoiler alert! Odgovor na pitanje

U stupcu `PORTS` vidimo označku `8080/tcp`, što znači da je port `8080` otvoren (*eng. exposed*) unutar kontejnera, ali ne prema domaćinu (*eng. host*).

Mapiranje portova možemo obaviti pomoću zastavice `-p` u naredbi `docker run`:

Sintaksa:

```
docker run -p <host_port>:<container_port> <image>:<tag>
```

Nekoliko primjera da bude jasnije:

- ako mikroservis interno radi na portu `8080`, možemo ga mapirati na isti port domaćina (ako je slobodan):

```
docker run -p 8080:8080 aiohttp-microservice:1.0
```

- ako mikroservis interno radi na portu `8080`, a želimo ga mapirati na port `8083` domaćina:

```
docker run -p 8083:8080 aiohttp-microservice:1.0
```

- ako mikroservis interno radi na portu 4000, a želimo ga mapirati na port 3000 domaćina:

```
docker run -p 3000:4000 aiohttp-microservice:1.0
```

Zastavicom `--name` moguće je i dodijeliti ime kontejneru, kako ga Docker ne bi nasumično generirao:

```
docker run --name aiohttp-microservice -p 8080:8080 aiohttp-microservice:1.0
```

Redoslijed zastavica u ovom slučaju nije bitan, ali je dobra praksa prvo navesti zastavice za mapiranje portova, a zatim ime i verziju predloška:

```
docker run -p 8080:8080 --name aiohttp-microservice aiohttp-microservice:1.0
```

Kako je ovaj kontejner već pokrenut, možemo ga zaustaviti naredbom `docker stop <container_id_or_name>`:

```
docker stop a604911ac56a  
# ili  
docker stop aiohttp-microservice
```

Pokrenut ćemo kontejner s mapiranim portom i provjeriti stanje naredbom `docker ps`:

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS
PORTE	NAMES			
702711364e85	aiohttp-microservice:1.0	"python app.py"	4 seconds ago	Up 4 seconds
0.0.0.0:8080->8080/tcp	aiohttp-microservice			

- `0.0.0.0:8080->8080/tcp` port je uspješno mapiran na port 8080 domaćina!

Praktično je koristiti Docker desktop sučelje budući da ono pamti kontejnere koje smo pokrenuli ili ugasili, **odnosno pamti parametre koje smo pritom koristili**. Tako možemo jednostavno ponovno pokrenuti kontejner klikom na `Actions -> Start` ili `Actions -> Restart`, na kontejneru gdje smo **već definirali mapiranje portova** u prvom pokretanju.



Pokretanje kontejnera s mapiranim portom iz Docker Desktop sučelja (tab `Containers`)

Ipak, stvari niti sada neće raditi! 😊

Ako otvorimo implementaciju mikroservisa, vidjet ćemo sljedeću naredbu za pokretanje:

```
web.run_app(app, host='localhost', port=8080)
```

- "slušaj na `localhost` hostu". `localhost` je ustvari *loopback* adresa mrežnog sučelja na računalu, a najčešće se asocira s IPv4 adresom `127.0.0.1`.
- port je `8080` i to je u redu.

**Problem:** mikroservis se pakira u kontejner, a kontejner je izolirano okruženje, odnosno **ne koristi mrežne postavke domaćina**. Prema tome, `localhost` u kontejneru se odnosi na sam kontejner, a ne na domaćinu!

Kada definiramo `localhost` kao host, mikroservis će prihvati samo zahtjeve koji dolaze iz samog kontejnera, a ne izvana.

Kako bismo definirali da mikroservis sluša na svim mrežnim sučeljima, **uključujući i domaćinu**, koristimo adresu `0.0.0.0`.

U produkcijskim okruženjima, ovo može biti sigurnosni rizik budući da mikroservis sluša na svim mrežnim sučeljima, ali za potrebe razvoja i testiranja, to je sasvim u redu.

Prema tome, izmijenit ćemo kod u mikroservisu:

```
web.run_app(app, host='0.0.0.0', port=8080) # zamijenili smo 'localhost' s '0.0.0.0'
```

Kontejner možemo izbrisati direktno u Docker Desktop aplikaciji ili naredbom `docker rm <container_id_or_name>`:

```
docker rm aiohttp-microservice
```

Nakon što izmjenimo kod mikroservisa, moramo **ponovno izraditi predložak** budući da je izmijenjen programski kod, a **Docker predložak je nepromjenjiv** - nije ga moguće izmjeniti nakon što je izgrađen.

Izgradimo ponovo predložak:

```
docker build -t aiohttp-microservice:1.0 .
```

Nakon što je predložak izgrađen, pokrenimo kontejner s mapiranim portom:

```
docker run -p 8080:8080 --name aiohttp-microservice aiohttp-microservice:1.0
```

---

Sada možemo poslati zahtjev na Docker kontejner s našeg računala koristeći `localhost:8080/proizvodi` u web pregledniku ili kroz HTTP klijent.

HTTP RS7 / proizvodi

Save Share

GET http://localhost:8080/proizvodi

Send

Overview Params Authorization Headers (7) Body Scripts Tests Settings Cookies

Body none form-data x-www-form-urlencoded raw binary GraphQL

This request does not have a body

Body Cookies Headers (4) Test Results

200 OK 62 ms 400 B Save Response

{ } JSON ▾ Preview Visualize

```
1 [  
2 {  
3     "id": 1,  
4     "naziv": "Laptop",  
5     "cijena": 1500  
6 },  
7 {  
8     "id": 2,  
9     "naziv": "Miš",  
10    "cijena": 20  
11 },  
12 {  
13     "id": 3,  
14     "naziv": "Tipkovnica",  
15     "cijena": 50  
16 },  
17 {  
18     "id": 4,  
19     "naziv": "Monitor",  
20     "cijena": 300  
21 },  
22 {  
23     "id": 5,  
24     "naziv": "Slušalice",  
25     "cijena": 100  
26 }
```

Poslali smo `GET /proizvodi` zahtjev na `localhost:8080` preko Postmana. Vidimo da kontejnerizirani mikroservis uspješno vraća listu proizvoda.

Detaljne mrežne postavke aktivnog Docker kontejnera možete provjeriti naredbom: `docker inspect <container_id_or_name>`:

```
docker inspect aiohttp-microservice
```

Osim toga, Docker Desktop pruža praktično sučelje za pregled drugih detalja aktivnog kontejnera, kao što su:

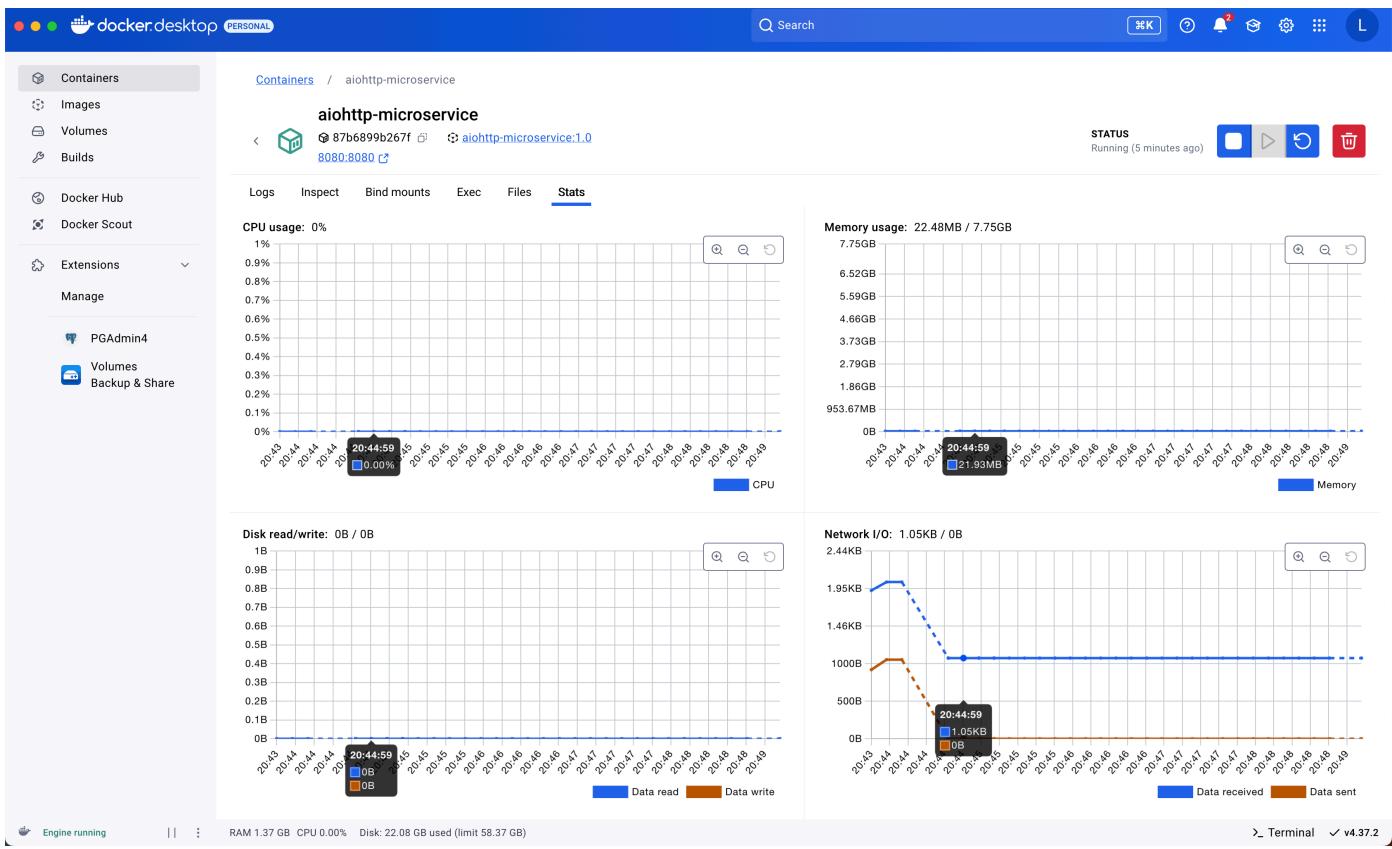
- logovi (terminal)
- detalji mrežnih postavki i druge informacije o kontejneru
- interni datotečni sustav kontejnera
- statistike o korištenju resursa

The screenshot shows the Docker Desktop interface. On the left, a sidebar includes 'Containers', 'Images', 'Volumes', 'Builds', 'Docker Hub', 'Docker Scout', 'Extensions' (with 'Manage'), 'PGAdmin4', and 'Volumes Backup & Share'. The main area displays the 'aiohttp-microservice' container, which is running. It shows logs from 2025-01-21 20:44:25, indicating it's running on http://0.0.0.0:8080. There are status buttons for Stop, Start, Restart, and Delete.

Pregled logova aktivnog kontejnera iz Docker Desktop sučelja

The screenshot shows the file system of the 'aiohttp-microservice' container. The 'Files' tab is selected. The root directory contains several subfolders and files, including '.dockerenv', 'app' (which is expanded to show 'app.py', 'Dockerfile', 'requirements.txt', and 'bin -> usr/bin'), 'boot', 'dev', 'etc', 'home', 'lib -> usr/lib', 'media', 'mnt', 'opt', 'proc', 'root', 'run', 'sbin -> usr/sbin', 'srv', 'sys', and 'tmp'. The 'app' folder was selected. The bottom of the screen shows resource usage: RAM 1.38 GB, CPU 0.00%, Disk: 22.08 GB used (limit 58.37 GB), and a terminal window labeled '>\_ Terminal v4.37.2'.

Pregled internog datotečnog sustava aktivnog kontejnera iz Docker Desktop sučelja (uočite da je `app.py` datoteka unutar datoteke `/app` koju smo definirali naredbom `WORKDIR`)



Pregled statistika o korištenju resursa aktivnog kontejnera iz Docker Desktop sučelja

Iz statistika je moguće pratiti korištenje resursa kao što su **CPU, memorija, mreža i disk**.

Uočite da je kod graf-a **Network I/O** prikazan promet podataka u i iz kontejnera, a *spike* koji vidimo odnosi se na HTTP zahtjev koji smo poslali mikroservisu kroz Postman malo ranije.

## 1.5 Tablica osnovnih Dockerfile naredbi

U nastavku je tablica osnovnih Dockerfile naredbi s primjerima i sintaksom, koje smo naučili u ovom poglavlju za definiranje **Docker predložaka**:

Naredba	Sintaksa	Objašnjenje	Primjer
<b>FROM</b>	<code>FROM &lt;image&gt;: &lt;tag&gt;</code>	Definira bazni predložak koji će se koristiti za definiciju vlastitog	<code>FROM ubuntu:20.04</code>
<b>WORKDIR</b>	<code>WORKDIR &lt;path&gt;</code>	Postavlja radni direktorij unutar kontejnera	<code>WORKDIR /app</code>
<b>COPY</b>	<code>COPY &lt;src&gt; &lt;dest&gt;</code>	Kopira datoteke ili direktorije s domaćina u datotečni sustav kontejnera.	<code>COPY . /app</code>
<b>CMD</b>	<code>CMD [ "executable", "arg1" ]</code>	Definira bilo koju naredbu koja će se izvršiti prilikom pokretanja kontejnera	<code>CMD [ "python", "app.py" ]</code>
<b>RUN</b>	<code>RUN &lt;command&gt;</code>	Izvršava bilo koju naredbu koja se poziva za vrijeme izgradnje Docker predloška	<code>RUN apt-get update &amp;&amp; apt-get install -y python3</code>
<b>EXPOSE</b>	<code>EXPOSE &lt;port&gt;</code>	Deklarira portove koje će kontejner koristiti.	<code>EXPOSE 8080</code>

## 1.6 Tablica osnovnih Docker naredbi

U nastavku je tablica osnovnih Docker naredbi s primjerima i sintaksom, koje smo naučili u ovom poglavlju za **izgradnju predložaka i upravljanje kontejnerima**.

Naredba	Sintaksa	Objašnjenje	Primjer
<b>build</b>	<code>docker build -t &lt;image_name&gt;:&lt;tag&gt; &lt;path&gt;</code>	Kreira Docker image iz Dockerfile-a i dodjeljuje mu ime i tag (opcionalno).	<code>docker build -t myapp:1.0 .</code>
<b>run</b>	<code>docker run -p &lt;host_port&gt;:&lt;container_port&gt; --name &lt;container_name&gt; &lt;image&gt;</code>	Pokreće kontejner iz Docker image-a, mapira portove i daje ime kontejneru.	<code>docker run -p 8080:80 --name mycontainer myapp</code>
<b>docker ps</b>	<code>docker ps</code>	Prikazuje listu trenutno aktivnih kontejnera.	<code>docker ps</code>
<b>docker inspect</b>	<code>docker inspect &lt;container_id_or_name&gt;</code>	Prikazuje detaljne informacije o određenom kontejneru ili image-u.	<code>docker inspect mycontainer</code>
<b>docker rm</b>	<code>docker rm &lt;container_id_or_name&gt;</code>	Briše zaustavljeni kontejner.	<code>docker rm mycontainer</code>
<b>docker stop</b>	<code>docker stop &lt;container_id_or_name&gt;</code>	Zaustavlja aktivni kontejner.	<code>docker stop mycontainer</code>

# 1.7 Kontejnerizacija FastAPI mikroservisa

Pokazat ćemo kako kontejnerizirati i nešto složenije mikroservise, poput `FastAPI` mikroservisa sa svim njegovim ovisnostima. Kod `aiohttp`-a proces je bio jednostavniji jer nam je jedina ovisnost bila `aiohttp` paket, dok su drugi uključeni standardnu biblioteku Pythona (npr. `asyncio`).

`FastAPI` mikroservis je složeniji jer koristi više ovisnosti, poput `uvicorn` poslužitelja, `pydantic` za validaciju podataka, `SQLAlchemy` ako radite s relacijskom bazom podataka, itd. Osim toga, dobro razvijeni `FastAPI` poslužitelj gotovo uvijek sadrži strukturirani kod s više datoteka, što znači da je potrebno kopirati više datoteka u kontejner.

## 1.7.1 Implementacija mikroservisa

Definirat ćemo `FastAPI` mikroservis koji vraća podatke o vremenu preko otvorenog API-ja **Državnog hidrometeorološkog zavoda** (DHMZ).

DHMZ nudi besplatan API za pristup meteorološkim podacima koji su pohranjeni u XML formatu, jedini uvjet korištenja je obavezno navođenje DHMZ-a kao izvora korištenih podataka. Odlučili smo koristiti DHMZ API i napraviti moderni FastAPI mikroservis budući da DHMZ API vraća podatke u XML formatu, što je pomalo nečitljivo i danas se sve rjeđe koristi.

Podaci su javno dostupni na sljedećoj poveznici: [https://meteo.hr/proizvodi.php?section=podaci&param=xml\\_korisnici](https://meteo.hr/proizvodi.php?section=podaci&param=xml_korisnici)

Uzet ćemo podatke o `Prognozi` za `Hrvatska/Zagreb sutra`, koji su dostupni na:  
[https://prognoza.hr/prognoza\\_sutra.xml](https://prognoza.hr/prognoza_sutra.xml)

Struktura XML-a slična je JSON strukturi, ali se umjesto `{}` koriste `<>` zagrade za definiranje početnog i završnog elementa, nalik HTML-u.

XML sadrži `metadata` podatke koji pokazuju datum i vrijeme kada su podaci izrađeni:

```
<metadata>
<datatime>210125</datatime>
<creationtime>Tue Jan 21 00:00:00 2025</creationtime>
</metadata>
```

Te podatke unutar `section` elementa, gdje je svako mjerjenje definirano elementom `station`:

```
<section name="All">
<param name="datum" value="220125"/>

<station name="sredisnja" lon="16.03" lat="45.82">
<param name="vrijeme" value="4"/>
<param name="Tmn" value="-1"/>
<param name="Tmx" value="4"/>
<param name="wind" value="6"/>
</station>

<station name="istocna" lon="18.63" lat="45.53">
```

```

<param name="vrijeme" value="6"/>
<param name="Tmn" value="-1"/>
<param name="Tmx" value="3"/>
<param name="wind" value="0"/>
</station>

<station name="gorska" lon="15.37" lat="44.55">
<param name="vrijeme" value="6"/>
<param name="Tmn" value="0"/>
<param name="Tmx" value="5"/>
<param name="wind" value="6"/>
</station>

<station name="unutrasnjost Dalmacije" lon="16.2" lat="44.03">xml
<param name="vrijeme" value="6"/>
<param name="Tmn" value="4"/>
<param name="Tmx" value="10"/>
<param name="wind" value="0"/>
</station>

</section>

```

Prvi korak je izrada direktorija i virtualnog okruženja:

```

mkdir weather-fastapi
cd weather-fastapi

conda create -n weather-fastapi python=3.11
conda activate weather-fastapi

```

Instalirat ćemo `FastAPI` s opcijom `[standard]`:

- pazite na navodne znakove

```
pip install "fastapi[standard]"
```

U datoteku `main.py` dodajemo osnovni kod za pokretanje:

```

# main.py
from fastapi import FastAPI
app = FastAPI()

@app.get("/")
def read_root():
    return {"message": "Hello, weather API!"}

```

Ako pogledate XML podatke, uočite da svaki `station` element ima sljedeće atribute:

- `name` - ime mjernog mjesto
- `lon` - geografska dužina

- `lat` - geografska širina
- `vrijeme` - prognoza vremena (npr. 4 - oblačno, 6 - sunčano)
- `Tmn` - minimalna temperatura
- `Tmx` - maksimalna temperatura
- `wind` - stupanjska jačina vjetra

Recimo da nas zanimaju samo podaci o nazivu mjesta, b i **maksimalnoj temperaturi, prognozi i jačini vjetra**.

Definirat ćemo Pydantic model `vrijeme` koji predstavlja te podatke:

```
# models.py

from pydantic import BaseModel

class Vrijeme(BaseModel):
    mjesto: str
    temperatura_min: int
    temperatura_max: int
    vjetar: int
```

Definirat ćemo endpoint `GET /vrijeme` koji će vraćati podatke o vremenu:

Povratna vrijednost endpointa je lista `Vrijeme` objekata:

```
# main.py
from models import Vrijeme

@app.get("/vrijeme", response_model = list[Vrijeme])
async def get_vrijeme():
    pass
```

Potrebno je slati HTTP zahtjev na `https://prognoza.hr/prognoza_sutra.xml` i parsirati XML podatke u `Vrijeme` objekte.

Za slanje zahtjeva možemo koristiti sinkronu biblioteku `requests` ili još bolje, ono što smo već naučili - asinkronu biblioteku `aiohttp`.

Instalirajmo `aiohttp` paket:

```
pip install aiohttp
```

Moramo otvoriti `clientSession` gdje ćemo slati `GET` zahtjev na URL `https://prognoza.hr/prognoza_sutra.xml`:

```
# main.py
from fastapi import FastAPI, HTTPException
from models import Vrijeme
import aiohttp
```

```

app = FastAPI()

@app.get("/vrijeme", response_model = list[Vrijeme])
async def get_vrijeme():
    url = "https://prognoza.hr/prognoza_sutra.xml"

    async with aiohttp.ClientSession() as session:
        response = await session.get(url)
        if response.status != 200: # u slučaju greške
            raise HTTPException(status_code=response.status, detail="Greška u dohvaćanju XML
podataka s DHMZ API-ja")
        xml_data = await response.text()

```

Možemo omotati kod u `try-except` blok kako bismo uhvatili eventualne greške prilikom slanja zahtjeva:

```

# main.py
from fastapi import status
try:
    async with aiohttp.ClientSession() as session:
        response = await session.get(url)
        if response.status != 200: # u slučaju greške
            raise HTTPException(status_code=response.status, detail="Greška u dohvaćanju XML
podataka s DHMZ API-ja")
        xml_data = await response.text()
except Exception as e: # Uhvati sve greške ako dođe do problema u slanju zahtjeva
    raise HTTPException(status_code=status.HTTP_500_INTERNAL_SERVER_ERROR, detail="Greška
u slanju HTML zahtjeva na DHMZ API")

```

Ako isprintamo `xml_data`, trebali bi dobiti XML podatke u terminalu.

Za samo parsiranje XML-a u Python objekte, možemo koristiti modul iz paketa `xml` - `xml.etree.ElementTree`.

```

# main.py
import xml.etree.ElementTree as ET

```

Pronaći ćemo sve oznake `station`, iterirati ih, te za svaku izvući podatke o `name`, `Tmn`, `Tmx` i `wind`:

```

# main.py
from fastapi import status
try:
    async with aiohttp.ClientSession() as session:
        response = await session.get(url)
        if response.status != 200: # u slučaju greške
            raise HTTPException(status_code=response.status, detail="Greška u dohvaćanju XML
podataka s DHMZ API-ja")
        xml_data = await response.text()
except Exception as e: # Uhvati sve greške ako dođe do problema u slanju zahtjeva

```

```

        raise HTTPException(status_code=status.HTTP_500_INTERNAL_SERVER_ERROR, detail="Greška
u slanju HTML zahtjeva na DHMZ API")

root = ET.fromstring(xml_data)
stations = root.findall("./station")
weather_list = []

for station in stations: # iteriraj kroz sve station elemente i izvuci podatke
    mjesto = station.attrib.get("name")
    temperatura_min = int(station.find("./param[@name='Tmn']").attrib.get("value"))
    temperatura_max = int(station.find("./param[@name='Tmx']").attrib.get("value"))
    vjetar = int(station.find("./param[@name='wind']").attrib.get("value"))

```

nakon toga ćemo u listu dodati `vrijeme` objekte koje definiramo dohvaćenim podacima

```

# main.py
@app.get("/vrijeme", response_model = list[Vrijeme])
async def get_vrijeme():
    """
    Dohvaća podatke o vremenu sa DHMZ API-ja, ali u JSON-u!

    Podaci dostupni na https://prognoza.hr/prognoza_sutra.xml
    """

    url = "https://prognoza.hr/prognoza_sutra.xml"

    try:
        async with aiohttp.ClientSession() as session:
            response = await session.get(url)
            if response.status != 200: # u slučaju greške
                raise HTTPException(status_code=response.status, detail="Greška u dohvaćanju XML
podataka s DHMZ API-ja")
            xml_data = await response.text()
    except Exception as e: # Uhvati sve greške ako dođe do problema u slanju zahtjeva
        raise HTTPException(status_code=status.HTTP_500_INTERNAL_SERVER_ERROR, detail="Greška
u slanju HTML zahtjeva na DHMZ API")

    root = ET.fromstring(xml_data)
    stations = root.findall("./station")
    weather_list = []

    for station in stations:
        mjesto = station.attrib.get("name")
        temperatura_min = int(station.find("./param[@name='Tmn']").attrib.get("value"))
        temperatura_max = int(station.find("./param[@name='Tmx']").attrib.get("value"))
        vjetar = int(station.find("./param[@name='wind']").attrib.get("value"))
        weather_list.append(Vrijeme(
            mjesto=mjesto,
            temperatura_min=temperatura_min,
            temperatura_max=temperatura_max,
            vjetar=vjetar

```

```
)  
return weather_list
```

Otvorite dokumentaciju mikroservisa na `http://localhost:8000/docs` i provjerite radi li sve kako treba, trebali bi vidjeti dokumentiranu rutu `/vrijeme` koja vraća podatke o vremenu u JSON formatu.

**default**

**GET /vrijeme Get Vrijeme**

Dohvaća podatke o vremenu sa DHMZ API-ja, ali u JSON-u!  
Podaci dostupni na [https://prognoza.hr/prognoza\\_sutra.xml](https://prognoza.hr/prognoza_sutra.xml)

**Parameters**

No parameters

**Responses**

**Curl**

```
curl -X 'GET' \
'http://127.0.0.1:8000/vrijeme' \
-H 'accept: application/json'
```

**Request URL**

```
http://127.0.0.1:8000/vrijeme
```

**Server response**

Code	Details
200	<b>Response body</b> [ { "mjesto": "sredinjia", "temperatura_min": -1, "temperatura_max": 4, "vjetar": 6 }, { "mjesto": "istocna", "temperatura_min": -1, "temperatura_max": 3, "vjetar": 0 }, { "mjesto": "zapadna", "temperatura_min": -1, "temperatura_max": 3, "vjetar": 0 } ]

**Execute** **Cancel** **Clear**

Tu ćemo stati, jer ovo nam je dovoljno složeno za pokazati kako kontejnerizirati mikroservis s više ovisnosti i strukturiranim kodom.

## 1.7.2 Kontejnerizacija mikroservisa

Prvi korak je izrada `requirements.txt` datoteke gdje ćemo pohraniti sve ovisnosti:

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

Vidimo da `FastAPI` ima puno više ovisnosti od `aiohttp` mikroservisa:

```
aiohappyeyeballs==2.4.4
aiohttp==3.11.11
aiosignal==1.3.2
annotated-types==0.7.0
anyio==4.8.0
attrs==24.3.0
certifi==2024.12.14
click==8.1.8
dnspython==2.7.0
email_validator==2.2.0
fastapi==0.115.6
fastapi-cli==0.0.7
frozenlist==1.5.0
h11==0.14.0
httpcore==1.0.7
httptools==0.6.4
httpx==0.28.1
idna==3.10
Jinja2==3.1.5
markdown-it-py==3.0.0
MarkupSafe==3.0.2
mdurl==0.1.2
multidict==6.1.0
propcache==0.2.1
pydantic==2.10.5
pydantic_core==2.27.2
Pygments==2.19.1
python-dotenv==1.0.1
python-multipart==0.0.20
PyYAML==6.0.2
rich==13.9.4
rich-toolkit==0.13.2
shellingham==1.5.4
sniffio==1.3.1
starlette==0.41.3
typer==0.15.1
typing_extensions==4.12.2
uvicorn==0.34.0
uvloop==0.21.0
watchfiles==1.0.4
websockets==14.2
yarl==1.18.3
```

Kreirajmo `Dockerfile` u direktoriju mikroservisa, struktura direktorija treba izgledati ovako:

```
weather-fastapi/
├── main.py
├── models.py
└── requirements.txt
└── Dockerfile
```

Prvo ćemo uzeti prethodni `Dockerfile` za `aiohttp` mikroservisa, a zatim ga malo prilagoditi:

```
FROM python:3.11-slim
WORKDIR /app
COPY . /app
RUN pip install -r requirements.txt
EXPOSE 8080
CMD ["python", "app.py"]
```

- `FROM python:3.11-slim` - OK
- `WORKDIR /app` - OK
- `COPY . /app` - OK
- `RUN pip install -r requirements.txt` - OK

FastAPI u pravilu radi na portu `8000`, a za pokretanje koristi `uvicorn` poslužitelj. Moramo izmijeniti `EXPOSE` i `CMD` naredbe i ručno pokrenuti poslužitelj i definirati port.

```
EXPOSE 8000
```

Naredba za pokretanje je: `uvicorn main:app`, međutim ako bismo dodali zastavice u `CMD` naredbu, moramo ih odvojiti zarezom, a ne razmakom:

Sintaksa:

```
CMD[naredba, argument1, argument2, ...]
```

odnosno:

```
CMD["neka_naredba", "--argument1", "--argument2", ...]
```

U našem slučaju, definirat ćemo `host` na `0.0.0.0` kao i kod `aiohttp` mikroservisa, a port postaviti na `8000`:

```
CMD ["uvicorn", "main:app", "--host", "0.0.0.0", "--port", "8000"]
```

Konačni `Dockerfile` izgleda ovako:

```
FROM python:3.11-slim
WORKDIR /app
COPY . /app
RUN pip install -r requirements.txt
EXPOSE 8000
CMD ["uvicorn", "main:app", "--host", "0.0.0.0", "--port", "8000"]
```

Izradite predložak naredbom `docker build`

- pazite da se nalazite u točnom direktoriju!

```
docker build -t weather-fastapi:1.0 .
```

Pokrenut ćemo kontejner s mapiranim portom:

```
docker run -p 8000:8000 --name weather-fastapi weather-fastapi:1.0
```

```
lukablaskovic — docker run -p 8000:8000 --name weather-fastapi weather-fastapi:1.0 — docker — docker run...
+ ~ docker run -p 8000:8000 --name weather-fastapi weather-fastapi:1.0
INFO: Started server process [1]
INFO: Waiting for application startup.
INFO: Application startup complete.
INFO: Uvicorn running on http://0.0.0.0:8000 (Press CTRL+C to quit)
```

Pokrenut `FastAPI` mikroservis u globalnom terminalu u obliku Docker kontejnera

To je to! Ako otvorimo web preglednik i posjetimo `localhost:8000/docs`, trebali bismo vidjeti dokumentaciju mikroservisa.

## 1.8 Zadaci za vježbu: Kontejnerizacija mikroservisa

- soon

## 2. Docker Compose

**Docker Compose** je alat koji omogućuje definiranje i pokretanje **više kontejnera kao cjeline** pomoću samo jedne konfiguracijske datoteke.

Prednost ovog alata je što značajno pojednostavljuje *multi-container* aplikacije, jer omogućuje definiranje svih kontejnera, mreže, volumena i drugih resursa unutar jedne datoteke. Bez obzira na to, svaki kontejner je i dalje izolirano okruženje.

Na ovaj način možemo praktično definirati složene raspodijeljene sustave koji se sastoje od više mikroservisa, baza podataka i drugih posrednika.

Datoteka koju koristi Docker Compose za definiranje kontejnera i drugih resursa naziva se `docker-compose.yml`.

*Primjer 1:* Raspodijeljeni sustav za e-trgovinu s tri mikroservisa, frontendom i bazom podataka:

- `frontend` Docker kontejner s frontend aplikacijom (npr. Vue.js)
- `backend` Docker kontejner s backend aplikacijom (npr. FastAPI) koji je posrednik između cjelokupnog sustava
- `paymentAPI` Docker kontejner s mikroservisom za plaćanje
- `accountingAPI` Docker kontejner s mikroservisom za računovodstvo
- `database` Docker kontejner s bazom podataka (npr. PostgreSQL)

*Primjer 2:* Raspodijeljeni sustav za analizu podataka s tri mikroservisa i bazom podataka:

- `dataAPI` Docker kontejner s mikroservisom za dohvaćanje podataka
- `analysisAPI` Docker kontejner s mikroservisom za analizu podataka
- `visualizationAPI` Docker kontejner s mikroservisom za vizualizaciju podataka
- `database` Docker kontejner s bazom podataka (npr. MongoDB)

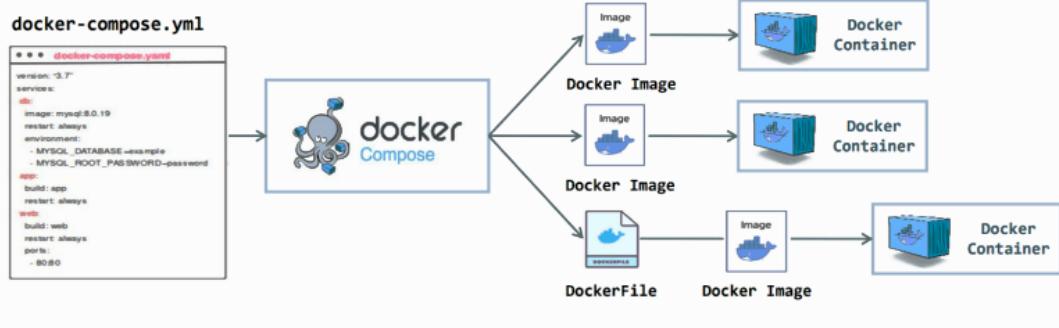
*Primjer 3:* Raspodijeljeni sustav za sustav za pohranu i dijeljenje datoteka koji se sastoji od četiri mikroservisa i baze podataka:

- `fileAPI` Docker kontejner s mikroservisom za pohranu i dijeljenje datoteka
- `encryptionAPI` Docker kontejner s mikroservisom za enkripciju i dekripciju datoteka
- `userAPI` Docker kontejner s mikroservisom za upravljanje korisnicima
- `notificationAPI` Docker kontejner s mikroservisom za obavijesti
- `database` Docker kontejner s bazom podataka (npr. MySQL)

Uočite zajedničke termine u svim ovim primjerima: to su **raspodijeljeni sustav, mikroservisi i docker kontejner**.

U mikroservisnoj arhitekturi, granularnost je ključna. Svaki mikroservis trebao bi obavljati jednu specifičnu funkciju, ili nekoliko srodnih funkcija. Mikroservis u ovom kontekstu može biti bilo koja aplikacija, a mi smo sad vidjeli kako to definirati različite API mikroservise.

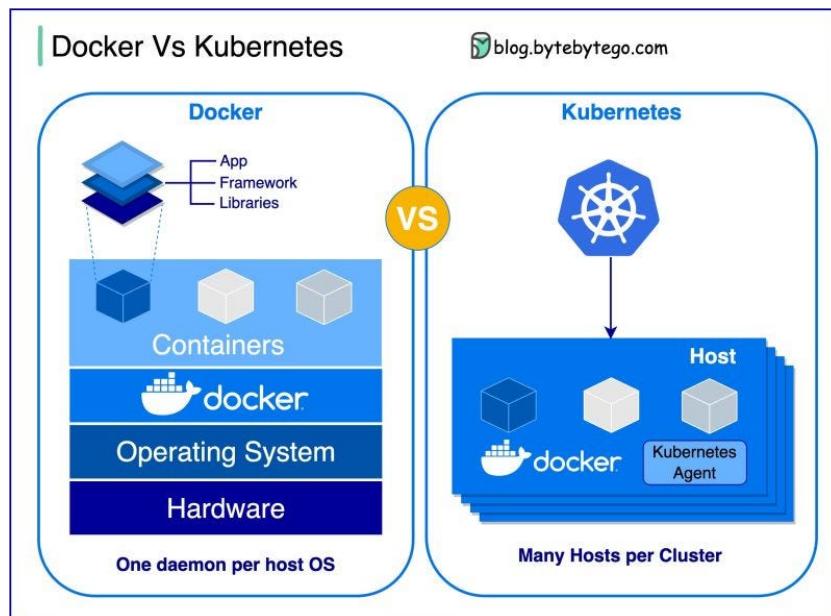
**Raspodijeljeni sustav** je skupina više mikroservisa, a svaki želimo "spakirati" u zaseban Docker kontejner. Kako se sustavi opisani u 3 prethodna primjera sastoje od više mikroservisa, a sustav u cijelosti ne može funkcionirati ako nedostaje barem jedan, praktično je koristiti **Docker Compose** za definiranje i upravljanje svim kontejnerima kao cjelinom 



Ilustracija rada Docker Compose alata

Međutim, **važno je naglasiti sljedeće**: Docker Compose alat nam omogućuje pokretanje više kontejnera kao cjeline, međutim ta cjelina se izvodi na **jednom računalu**. Dakle, ako se jedno računalo pokvari, cijeli sustav će prestati raditi, bez obzira što je on na aplikacijskog razini raspodijeljen na više mikroservisa.

Postoje sofisticirana programska rješenja koja omogućuju **orkestraciju raspodijeljenog sustava** na više računala, kao što su **Kubernetes** i **Docker Swarm**. Ova složena rješenja omogućuju automatsko upravljanje kontejnerima, skaliranje, nadzor i druge napredne značajke. Međutim, to je tema sama za sebe i izlazi iz okvira ovog kolegija.



Ilustracija usporedbe Docker i Kubernetes alata

## 2.1 Kako spakirati više mikroservisa u jednu cjelinu

Docker Compose dolazi već instaliran s najnovijom verzijom Docker Desktop aplikacije, a dostupan je na svim operacijskim sustavima.

Možete provjeriti verziju Docker Compose alata naredbom:

```
docker compose version
```

Na linux sustavima je potencijalno potrebno naknadno instalirati Docker Compose alat, izvorni kod možete pronaći na sljedećoj poveznici: <https://github.com/docker/compose/releases>

Docker Compose koristi `docker-compose.yml` datoteku za definiranje kontejnera i drugih resursa koji će se pokrenuti kao cjelina.

Zašto ne bismo kombinirali `aiohttp` i `FastAPI` mikroservise koje smo ranije definirali u jedan "raspodijeljeni sustav" pomoću Docker Compose alata?

Napravit ćemo novi direktorij `compose-example` i unutar njega kreirati `docker-compose.yml` datoteku:

```
mkdir compose-example
cd compose-example
touch docker-compose.yml
```

Struktura direktorija treba izgledati ovako:

```
compose-example/
└── docker-compose.yml
```

Kako bi stvari imale više smisla, možemo malo redizajnirati `aiohttp` mikroservis na način da vraća podatke o regijama, umjesto o proizvodima.

Kopirat ćemo `aiohttp` mikroservis u novi direktorij `aiohttp-regije` koji se nalazi unutar `compose-example` direktorija:

Struktura direktorija `compose-example` treba izgledati ovako:

```
compose-example/
├── aiohttp-regije/
│   ├── app.py
│   └── Dockerfile
└── docker-compose.yml
```

U `aiohttp` mikroservisu, malo ćemo izmjeniti definiciju ruta i podatke koje vraća:

```
# compose-example/aiohttp-regije/app.py

import asyncio
from aiohttp import web
```

```

app = web.Application()

dummy_podaci_regije = [
    {"kljuc": "sredisnja", "naziv": "Središnja Hrvatska", "gradovi": ["Zagreb", "Karlovac", "Sisak"]},
    {"kljuc": "istocna", "naziv": "Istočna Hrvatska", "gradovi": ["Osijek", "Slavonski Brod", "Vinkovci", "Vukovar"]},
    {"kljuc": "gorska", "naziv": "Gorska Hrvatska", "gradovi": ["Delnice", "Čabar", "Vrbovsko"]},
    {"kljuc": "unutrasnjost Dalmacije", "naziv": "Unutrašnjost Dalmacije", "gradovi": ["Knin", "Sinj", "Imotski"]},
    {"kljuc": "sjeverni Jadran", "naziv": "Sjeverni Jadran", "gradovi": ["Rijeka", "Pula", "Opatija", "Rovinj"]},
    {"kljuc": "srednji Jadran", "naziv": "Srednji Jadran", "gradovi": ["Split", "Zadar", "Šibenik"]},
    {"kljuc": "južni Jadran", "naziv": "Južni Jadran", "gradovi": ["Dubrovnik", "Metković", "Ploče"]}
]

async def get_regije(request):
    return web.json_response(dummy_podaci_regije)

async def get_regija(request):
    kljuc = request.match_info['kljuc']
    for regija in dummy_podaci_regije:
        if regija['kljuc'] == kljuc:
            return web.json_response(regija)
    return web.json_response({"error": "Regija nije pronađena"}, status=404)

app.router.add_get("/regije", get_regije)
app.router.add_get("/regije/{kljuc}", get_regija)

web.run_app(app, host='0.0.0.0', port=4000) # promijenili smo port na 4000, čisto tako

```

Naravno, moramo generirati i `requirements.txt` datoteku:

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

Definirajmo `Dockerfile` za `aiohttp-regije` mikroservis:

```

# compose-example/aiohttp-regije/Dockerfile

FROM python:3.11-slim
WORKDIR /app
COPY . /app
RUN pip install -r requirements.txt
EXPOSE 4000
CMD ["python", "app.py"]

```

Riješili smo `aiohttp-regije` mikroservis, struktura direktorija `compose-example` treba izgledati ovako:

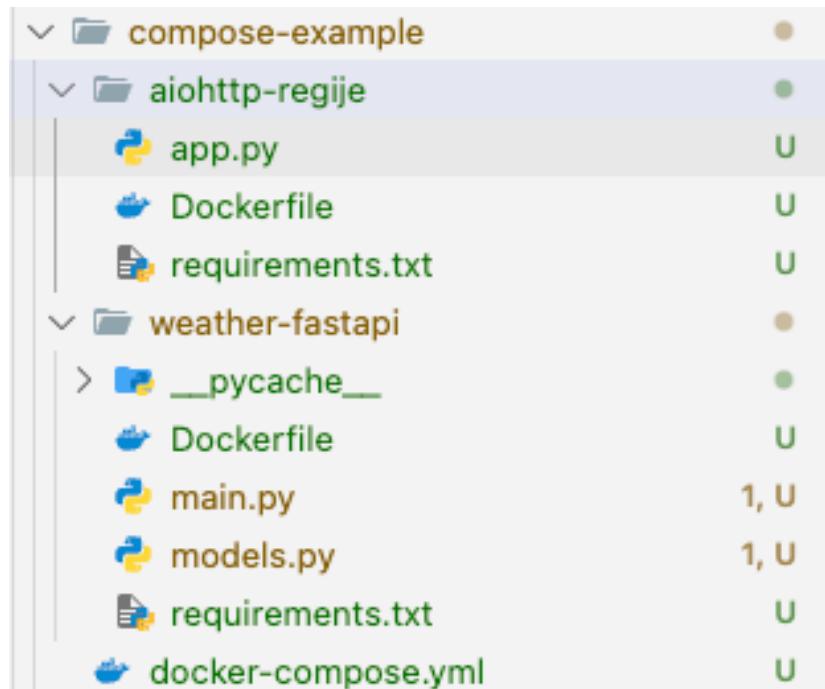
```
compose-example/
├── aiohttp-regije/
│   ├── app.py
│   ├── requirements.txt
│   └── Dockerfile
└── docker-compose.yml
```

FastAPI mikroservis nećemo mijenjati, već ga jednostavno kopiramo u `compose-example` direktorij:

```
compose-example/
├── aiohttp-regije/
│   ├── app.py
│   ├── requirements.txt
│   └── Dockerfile
├── weather-fastapi/
│   ├── main.py
│   ├── models.py
│   ├── requirements.txt
│   └── Dockerfile
└── docker-compose.yml
```

Ako koristite VS Code, preporuka je instalirati Material Icon Theme ekstenziju kako bi direktoriji i datoteke imali ikone:

- [Material Icon Theme](#)



Struktura direktorija `compose-example` u VS Code okruženju, `__pycache__` direktoriji su generirani od strane Python interpretera i možemo ih ignorirati

To je to, struktura je spremna, a sada možemo ova dva mikroservisa pokrenuti kao cjelinu pomoću Docker Compose alata!

## 2.1.1 Sintaksa `docker-compose.yml` datoteke

Otvorite `docker-compose.yml` datoteku u `compose-example` direktoriju.

Na početku svake `docker-compose.yml` datoteke obično se nalazi verzija Docker Compose alata, mi ćemo koristiti verziju `3.8`:

`docker-compose.yml` datoteka:

```
version: '3.8'
```

Mikroservise ćemo definirati unutar ključa `services`:

```
version: '3.8'

services:
  naziv_servisa:
    image: ime_docker_predloska
    ports:
      - "host_port:container_port"
```

Svaki mikroservis je ustvari kontejner, a **za svaki kontejner** moramo obavezno definirati koji Docker predložak koristi te koji portovi su mapirani:

```
version: '3.8'

services:
  aiohttp-regije: # ime kontejnera
    image: aiohttp-regije:1.0 # ime Docker predloška
    ports: # mapiranje portova
      - "4000:4000" # host_port:container_port
```

Dodat ćemo i FastAPI mikroservis:

```
version: '3.8'

services:
  aiohttp-regije:
    image: aiohttp-regije:1.0
    ports:
      - "4000:4000"

  weather-fastapi:
    image: weather-fastapi:1.0
    ports:
      - "8000:8000"
```

Moramo paziti da postoje dva različita Docker predloška definirana na našem računalu, `aiohttp-regije:1.0` i `weather-fastapi:1.0`, koje smo definirali u prethodnim koracima.

Aktivne Docker predloške možemo provjeriti naredbom:

```
docker images
```

Ako ih nema, izgradite prvo oba predloška:

- pazite da se nalazite u direktoriju gdje se nalazi `Dockerfile` određenog mikroservisa!

```
cd aiohttp-regije  
docker build -t aiohttp-regije:1.0 .
```

```
cd ..  
cd weather-fastapi  
docker build -t weather-fastapi:1.0 .
```

Nakon što smo izgradili oba predloška, možemo pokrenuti oba mikroservisa kao cjelinu pomoću Docker Compose alata. Navigirajte u `compose-example` direktorij i pokrenite sljedeću naredbu:

```
docker compose up
```

Ova naredba pokreće sve mikroservise definirane u `docker-compose.yml` datoteci kao cjelinu. Moguće da će vas Docker tražiti autentifikaciju kako bi pristupio vašim predlošcima, u tom slučaju unesite:

```
docker login
```

Nakon što se uspješno autentificirate, Docker Compose će pokrenuti oba mikroservisa kao cjelinu! 🚀

Name	Container ID	Image	Port(s)	CPU (%)	Last started	Actions
compose-example	-	-	-	0.22%	2 minutes ago	[stop] [restart] [delete]
weather-fastapi-1	af9a9466ed7c	weather-fastapi:1.0	8000:8000	0.22%	2 minutes ago	[stop] [restart] [delete]
aiohttp-regije-1	1f11677d0dde	aiohttp-regije:1.0	4000:4000	0%	2 minutes ago	[stop] [restart] [delete]

```
2025-01-22 00:17:46 weather-fastapi-1 | INFO: Started server process [1]  
2025-01-22 00:17:46 weather-fastapi-1 | INFO: Waiting for application startup.  
2025-01-22 00:17:46 weather-fastapi-1 | INFO: Application startup complete.  
2025-01-22 00:17:46 weather-fastapi-1 | INFO: Uvicorn running on http://0.0.0.0:8000 (Press CTRL+C to quit)
```

Pokrenuti mikroservisi kao cjelina pomoću Docker Compose alata. Prikaz unutar Docker Desktop aplikacije

Vidimo da svaki servis ima svoj vlastiti kontejner i da su mapirani portovi definirani u `docker-compose.yml` datoteci.

Mikroservise možemo zaustaviti naredbom:

```
docker compose down
```

## 2.2 Interna komunikacija mikroservisa

Jedna od ključnih značajki mikroservisne arhitekture je **interni komunikacija** između mikroservisa. Svaki mikroservis trebao bi biti izolirano okruženje, a komunikacija između mikroservisa trebala bi biti sigurna i pouzdana.

U našem primjeru, `aiohttp-regije` mikroservis vraća podatke o regijama, a `weather-fastapi` mikroservis vraća podatke o vremenu, a pristup im možemo preko domaćina i odgovarajućih portova.

Što ako želimo da `weather-fastapi` mikroservis dohvata podatke o regijama iz `aiohttp-regije` mikroservisa?

- u tom slučaju pričamo o internoj komunikaciji između mikroservisa
- dakle, servis A i B komuniciraju između sebe, a ne preko vanjskog korisnika (domaćina)
- ovo je **ključna značajka mikroservisne arhitekture**

Recimo da želimo da `weather-fastapi` mikroservis dohvata podatke o regijama iz `aiohttp-regije` mikroservisa jednom kad domaćin pošalje zahtjev na `/vrijeme` rutu mikroservisa `weather-fastapi`.

Domaćin ↔ weather-fastapi ↔ aiohttp-regije

Premda nije potrebno eksplicitno navoditi, uobičajeno je definirati **bridge network** unutar `docker-compose.yml` datoteke kako bi svi mikroservisi bili povezani na istoj mreži.

Mreže dodajemo pod ključ `networks`:

```
version: '3.8'

services:
  aiohttp-regije:
    image: aiohttp-regije:1.0
    ports:
      - "4000:4000"
    networks:
      - interna_mreza

  weather-fastapi:
    image: weather-fastapi:1.0
    ports:
      - "8000:8000"
    networks:
      - interna_mreza

networks:
```

```
interna_mreza: # proizvoljno ime mreže
  driver: bridge # tip mreže
```

**Docker compose** nam omogućuje da koristimo sam naziv kontejnera kao domenu, odnosno *hostname* prilikom definiranja internih komunikacija.

Dakle, `weather-fastapi` mikroservis može poslati HTTP zahtjev na `aiohttp-regije` mikroservis, putem rute:

```
http://aiohttp-regije:4000/regije
```

S druge strane, `aiohttp-regije` mikroservis može poslati HTTP zahtjev na `weather-fastapi` mikroservis, putem rute:

```
http://weather-fastapi:8000/vrijeme
```

Idemo ovo testirati, nadogradit ćemo mikroservis `weather-fastapi` tako da dohvaća podatke o regijama iz `aiohttp-regije` mikroservisa.

U `weather-fastapi` mikroservisu, dodajemo novu rutu `/vrijeme-regije` koja će dohvaćati podatke o regijama iz `aiohttp-regije` mikroservisa:

```
# compose-example/weather-fastapi/main.py

@app.get("/regije")
async def get_regije():
    async with aiohttp.ClientSession() as session:
        response = await session.get("http://aiohttp-regije:4000/regije") # koristimo naziv
kontejnera kao domenu
        regije = await response.json()
        return regije
```

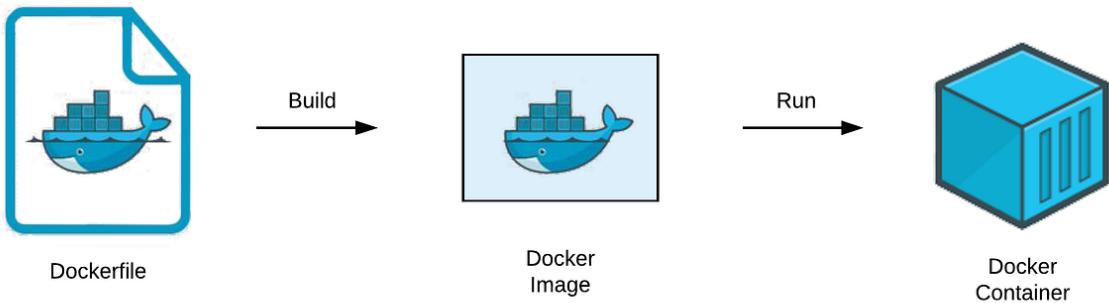
Obzirom da smo izmijenili kod, moramo ponovno izgraditi predložak:

```
cd weather-fastapi
docker build -t weather-fastapi:1.0 .
```

Nakon što izgradimo predložak, možemo ponovno pokrenuti mikroservise kao cjelinu:

```
docker compose up
```

Otvorite dokumentaciju mikroservisa na `http://localhost:8000/docs` i pokušajte pozvati rutu `/regije`. Trebali biste dobiti podatke o regijama koje vraća `aiohttp-regije` mikroservis.



Interna komunikacija između mikroservisa pomoću Docker Compose alata

## 2.3 Varijable okruženja u Dockeru

**Varijable okruženja** (eng. *environment variables*) su ključne za konfiguraciju mikroservisa, jer nam omogućuju da postavimo različite vrijednosti za različite okoline (npr. razvoj, testiranje, produkcija)

Stvari su trivijalne kada definiramo varijable okruženja za svaki mikroservis zasebno. Ako verzioniramo kod, svakako je uobičajena praksa koristiti ih za osjetljive podatke, poput lozinki, privatnih ključeva i drugih tajnih informacija.

Varijable okruženja u Pythonu možemo postaviti pomoću `os` modula ili pomoću `.env` datoteke i `python-dotenv` paketa.

```

import os

os.environ['VARIJABLA'] = 'vrijednost'

```

Ipak, u pravilu ih u kodu želimo samo čitati, ne i postavljati. Varijable okruženja možemo definirati unutar datoteke `.env`:

Vratimo se na primjer s `aiohttp-regije` mikroservisom. Definirat ćemo varijablu okruženja `PORT` unutar `.env` datoteke. Recimo da želimo koristiti različiti PORT ovisno o okolini.

- u lokalnom razvoju koristimo port `4000`
- u kontejneriziranoj okolini koristimo port `5000`

Instalirat ćemo paket `python-dotenv` u okruženju `aiohttp-microservice`:

```

conda activate aiohttp-microservice
pip install python-dotenv

```

Kako smo sad izmijenili ovisnosti, odmah ćemo ažurirati naš `requirements.txt`:

```

pip freeze > requirements.txt

```

Nakon toga, kreiramo `.env` datoteku u `aiohttp-regije` direktoriju:

```
compose-example/
├── aiohttp-regije/
│   ├── app.py
│   ├── requirements.txt
│   ├── Dockerfile
│   └── .env
├── weather-fastapi/
│   ├── main.py
│   ├── models.py
│   ├── requirements.txt
│   └── Dockerfile
└── docker-compose.yml
```

Unutar datoteke `.env` definiramo varijablu okruženja `PORT` i postavljamo vrijednost na `4000`:

```
PORT=4000
```

U `app.py` datoteci, čitamo varijablu okruženja `PORT` i koristimo je za postavljanje poslužitelja:

```
# compose-example/aiohttp-regije/app.py

import os,
from dotenv import load_dotenv

load_dotenv() # učitavamo varijable iz .env datoteke

PORT = os.getenv("PORT") # čitamo varijablu okruženja PORT
```

Sada ju možemo koristi za pokretanje mikroservisa:

```
# compose-example/aiohttp-regije/app.py

web.run_app(app, host='0.0.0.0', port=int(PORT)) # koristimo varijablu okruženja PORT
```

To je to, `Dockerfile` možemo ostaviti nepromijenjen bez obzira na naredbu `EXPOSE 4000` - rekli smo da je to samo informativno i ne utječe na rad kontejnera.

Ipak, moramo ažurirati `docker-compose.yml` datoteku kako bismo izmjenili port u kontejnerskom okruženju:

Možemo definirati varijable okruženja unutar `environment` ključa:

```
version: '3.8'

services:
  aiohttp-regije:
    image: aiohttp-regije:1.0
    ports:
      - 4000:4000 # onda ovo možemo izmijeniti na način da čitamo varijablu okruženja
```

```

environment:
  - PORT=4000 # definiramo varijablu okruženja PORT i postavljamo vrijednost na 4000
networks:
  - interna_mreza

weather-fastapi:
  image: weather-fastapi:1.0
  ports:
    - "8000:8000"
  networks:
    - interna_mreza

```

Sada je potrebno ažurirati ključ `ports` unutar `aiohttp-regije` mikroservisa kako bi čitao varijablu okruženja `PORT`:

```

version: '3.8'

services:
  aiohttp-regije:
    image: aiohttp-regije:1.0
    ports:
      - "${PORT}:${PORT}" # koristimo varijablu okruženja PORT i za host i za kontejner port
    environment:
      - PORT=4000
    networks:
      - interna_mreza

  weather-fastapi:
    image: weather-fastapi:1.0
    ports:
      - "8000:8000"
    networks:
      - interna_mreza

```

Ipak, ako želimo pregaziti vrijednost varijable okruženja unutar `environment`, možemo to učiniti pomoću `.env` datoteke i `env_file` ključa:

```

version: '3.8'

services:
  aiohttp-regije:
    image: aiohttp-regije:1.0
    ports:
      - "${PORT}:${PORT}" # čitamo varijablu okruženja PORT iz .env datoteke
    env_file:
      - .env # učitavamo varijable okruženja iz .env datoteke
    networks:
      - interna_mreza

  weather-fastapi:

```

```
image: weather-fastapi:1.0
ports:
- "8000:8000"
networks:
- interna_mreza
```

**Važno je ovdje uočiti sljedeće:** U ovom kontekstu (datoteke `docker-compose.yml`), `.env` datoteka se nalazi u istom direktoriju kao i `docker-compose.yml` datoteka, **a ne u direktoriju mikroservisa!**

Dakle, možemo ju premjestiti u `compose-example` direktorij:

```
compose-example/
├── aiohttp-regije/
│   ├── app.py
│   ├── requirements.txt
│   └── Dockerfile
├── weather-fastapi/
│   ├── main.py
│   ├── models.py
│   ├── requirements.txt
│   └── Dockerfile
└── .env
└── docker-compose.yml
```

Izgradit ćemo ponovno predložak `aiohttp-regije`:

```
cd aiohttp-regije
docker build -t aiohttp-regije:1.0 .
```

Pokrećemo mikroservise:

```
docker compose up
```

Provjerite radi li kontejner `aiohttp-regije` na ispravnom portu koji ste definirali u `.env` datoteci.

```
docker ps
```

To je to! Dobivamo ispravni port koji smo definirali unutar `.env` datoteke:

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS
PORTS		NAMES		
71a1a86ccd89	weather-fastapi:1.0	"uvicorn main:app --..."	About a minute ago	Up 10 seconds
94d7df51696f	aiohttp-regije:1.0	"python app.py"	About a minute ago	Up 10 seconds

## 2.4 Zadaci za vježbu: Docker Compose

- soon

# 3 Load balancing (`nginx`)

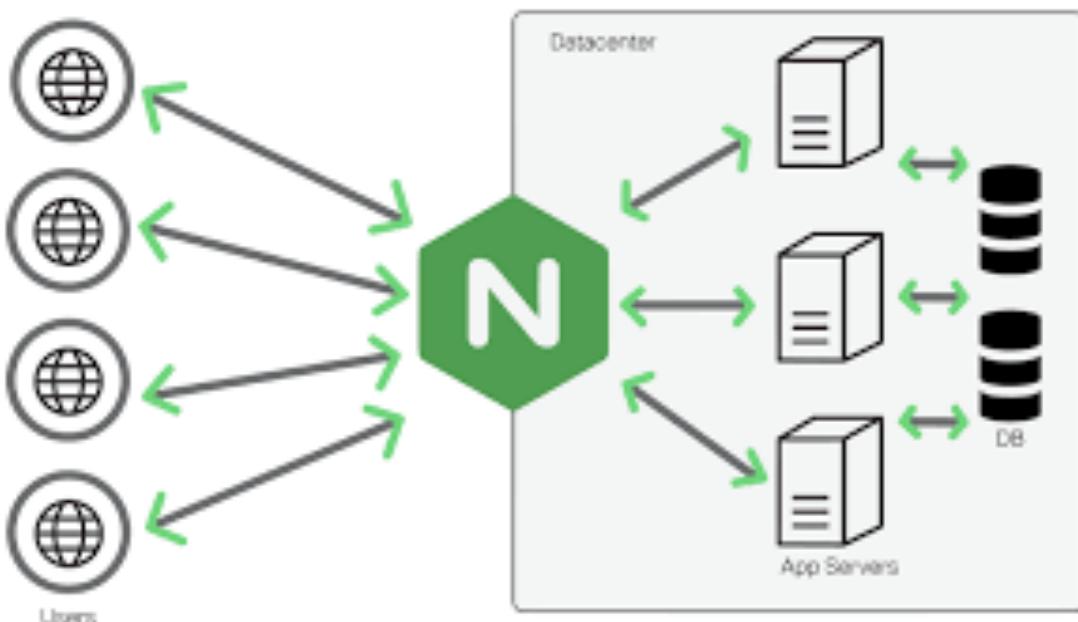
**Load balancing** je tehnika koja se koristi za distribuciju opterećenja između više poslužitelja, računala ili mrežnih uređaja. Ova tehnika omogućuje da se opterećenje ravnomjerno raspodijeli između više poslužitelja, kako bi se osigurala visoka dostupnost i pouzdanost sustava.

Ciljevi load balancinga su sljedeći:

- **Ravnomjerna raspodjela opterećenja** - svaki poslužitelj dobiva jednaku količinu zahtjeva
- **Visoka dostupnost** - ako jedan poslužitelj prestane raditi, drugi preuzimaju njegovo opterećenje
- **Prevencija da jedan poslužitelj postane usko grlo** - ako jedan poslužitelj postane preopterećen, load balancer preusmjerava zahtjeve na druge poslužitelje
- **Povećanje performansi** - load balancer može koristiti različite algoritme za raspodjelu opterećenja, ovisno o potrebama sustava

Postoje različite vrste load balančera, međutim mi se nećemo baviti detaljima. U ovom primjeru koristit ćemo **nginx** kao load balancer za naše mikroservise.

**nginx** je popularan web poslužitelj i *reverse proxy server* koji se koristi za posluživanje web stranica, aplikacija i API-ja. Osim toga, **nginx** se može koristiti kao load balancer za distribuciju opterećenja između više poslužitelja.



Ilustracija rada load balančera

`nginx` nije dio Dockera, niti Pythona, već je zaseban softver koji se može instalirati na računalo.

Međutim, možemo koristiti `nginx` kao Docker kontejner i konfigurirati ga kao load balancer za naše mikroservise.

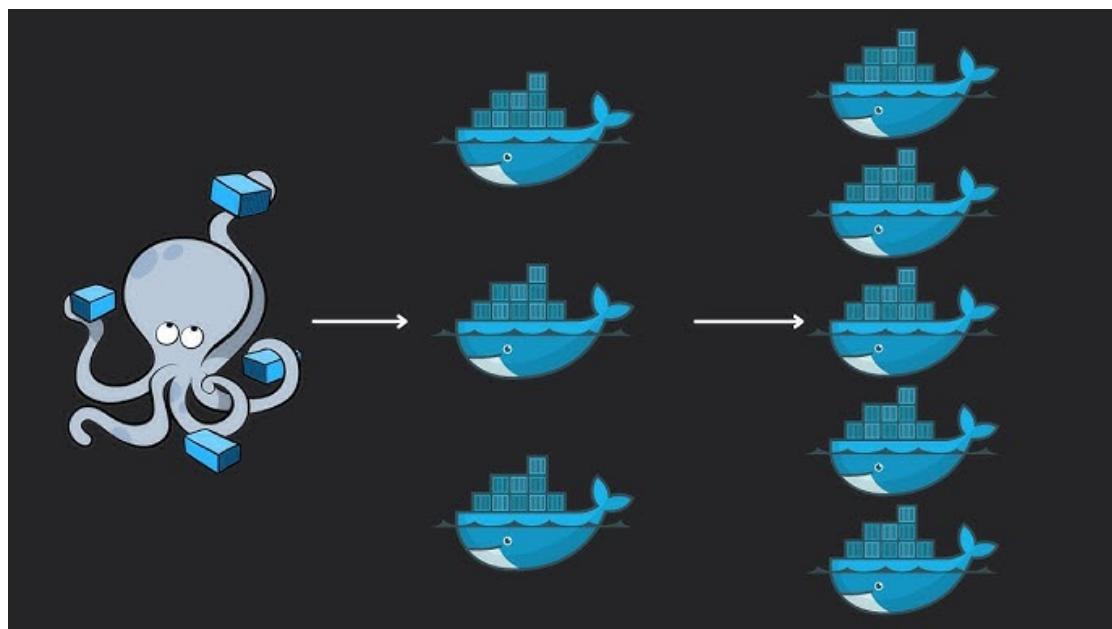
Možemo ga preuzeti preko Docker Huba, poveznice: [https://hub.docker.com/\\_/nginx](https://hub.docker.com/_/nginx)

```
docker pull nginx
```

## 3.1 Horizontalno skaliranje koristeći samo Docker Compose

**Horizontalno skaliranje** (eng. *Horizontal scaling*) mikroservisa odnosi se na povećanje broja instanci mikroservisa kako bi se povećala dostupnost i performanse sustava. Primjerice, ako iz našeg primjera imamo samo jednu instancu `weather-fastapi` mikroservisa, možemo dodati još jednu instancu u slučaju da prva prestane raditi.

Dakle, u ovom kontekstu samo povećavamo **broj instanci mikroservisa**.



Ilustracija horizontalnog skaliranja mikroservisa

Na primjer, želimo dodati 3 replike `weather-fastapi` mikroservisa i 2 replike `aiohttp-regije` mikroservisa. To radimo kroz `docker-compose.yml` datoteku:

Sintaksa:

```
version: '3.8'

services:
  naziv_servisa:
    image: ime_docker_predloska
    ports:
      - "host_port:container_port"
    deploy:
      replicas: broj_replika
```

Odnosno na našem primjeru:

```
version: '3.8'

services:
```

```

aiohttp-regije:
  image: aiohttp-regije:1.0
  ports:
    - "${PORT}:${PORT}"
  env_file:
    - .env
  networks:
    - interna_mreza
  deploy:
    replicas: 2

weather-fastapi:
  image: weather-fastapi:1.0
  ports:
    - "8000:8000"
  networks:
    - interna_mreza
  deploy:
    replicas: 3

networks:
  interna_mreza: # proizvoljno ime mreže
  driver: bridge # tip mreže

```

Možemo pokrenuti ove kontejnere, međutim dobit ćemo **grešku** prilikom pokretanja budući da Docker pokušava mapirati isti port na više kontejnera prema domaćinu, što nije dozvoljeno.

**Containers** [Give feedback](#)

View all your running containers and applications. [Learn more](#)

Container CPU usage		Container memory usage		Show charts	
0.39% / 800% (8 CPUs available)		69.95MB / 7.57GB			
<input type="checkbox"/>	<input type="checkbox"/> Search	<input type="checkbox"/>	<input checked="" type="checkbox"/> Only show running containers		
	Name	Container ID	Image	Port(s)	Actions
<input type="checkbox"/>	compose-example	-	-	-	<input type="checkbox"/> ⋮ <input type="checkbox"/> ⌂
<input type="checkbox"/>	aiohttp-regije-1	952a5232b867	aiohttp-regije:1.0	4000:4000 ⌂	<input type="checkbox"/> ⋮ <input type="checkbox"/> ⌂
<input type="checkbox"/>	aiohttp-regije-2	3786219894d8	aiohttp-regije:1.0	4000:4000	<input type="checkbox"/> ⋮ <input type="checkbox"/> ⌂
<input type="checkbox"/>	weather-fastapi-1	6687eb09a7a7	weather-fastapi:1.0	8000:8000	<input type="checkbox"/> ⋮ <input type="checkbox"/> ⌂
<input type="checkbox"/>	weather-fastapi-2	fe83aa3c38ba	weather-fastapi:1.0	8000:8000	<input type="checkbox"/> ⋮ <input type="checkbox"/> ⌂
<input type="checkbox"/>	weather-fastapi-3	c588b7ed2691	weather-fastapi:1.0	8000:8000 ⌂	<input type="checkbox"/> ⋮ <input type="checkbox"/> ⌂

Problem možemo riješiti koristeći **nginx** kao load balancer koji će **distribuirati zahtjeve na različite mikroservise**.

Prvo ćemo dodati `nginx` kontejner u `docker-compose.yml` datoteku:

- radi pojednostavljenja, trenutno ćemo maknuti dinamičko mapiranje portova i staviti fiksne portove za svaki mikroservis

```

version: '3.8'

services:

```

```

aiohttp-regije:
  image: aiohttp-regije:1.0
  ports:
    - "4000:4000" # fiksni port za aiohttp-regije
  networks:
    - interna_mreza
  deploy:
    replicas: 2

weather-fastapi:
  image: weather-fastapi:1.0
  ports:
    - "8000:8000" # fiksni port za weather-fastapi
  networks:
    - interna_mreza
  deploy:
    replicas: 3

nginx: # dodajemo nginx load balancer
  image: nginx
  ports:
    - "80:80"
  volumes: # mapiramo konfiguracijsku datoteku
    - ./nginx.conf:/etc/nginx/nginx.conf # konfiguracijska datoteka za nginx je
nginx.conf
  networks:
    - interna_mreza

networks:
  interna_mreza: # proizvoljno ime mreže
    driver: bridge # tip mreže

```

`nginx` definiramo unutar konfiguracijske datoteke `nginx.conf` koja se mora nalaziti u istom direktoriju kao i `docker-compose.yml` datoteka:

Struktura direktorija:

```

compose-example/
  └── aiohttp-regije/
      ├── app.py
      ├── requirements.txt
      └── Dockerfile
  └── weather-fastapi/
      ├── main.py
      ├── models.py
      ├── requirements.txt
      └── Dockerfile
  └── nginx.conf
  └── .env
  └── docker-compose.yml

```

**Reverse proxy** odnosi se na tehniku koja omogućuje da se zahtjevi preusmjere s jednog poslužitelja na drugi. U našem slučaju, `nginx` će **preusmjeravati zahtjeve na različite mikroservise**.

Unutar `nginx.conf` datoteke, prvo ćemo definirati `upstream` blok u kojem ćemo navesti sve mikroservise na koje će `nginx` preusmjeravati zahtjeve, to su `aiohttp-regije` i `weather-fastapi` mikroservisi:

**VAŽNO!** Bez obzira na interne portove unutar kontejnera, ovdje možemo definirati na koje portove će `nginx` preusmjeravati zahtjeve, odnosno koje portove će koristiti domaćin (**krajnji korisnik**).

Trenutni portovi definirani unutar `docker-compose.yml` su:

- `aiohttp-regije`: 4000
- `weather-fastapi`: 8000

Otvorite `nginx.conf` datoteku:

1. korak: definicija `events` bloka gdje navodimo najveći broj konekcija koje `nginx` može obraditi istovremeno

```
events {  
    worker_connections 1024;  
}
```

2. korak: definicija `http` bloka gdje navodimo `upstream` blok i `server` blok

Prvo ćemo navesti `upstream` blokove u kojima navodimo naše mikroservise:

```
http {  
    upstream aiohttp-regije {  
        server aiohttp-regije:4000;  
    }  
  
    upstream weather-fastapi {  
        server weather-fastapi:8000;  
    }  
}
```

3. korak: definiramo reverse-proxy na proizvoljnem portu (npr. 80) i **preusmjeravamo sve zahtjeve** na `aiohttp-regije` i `weather-fastapi` mikroservise:

- na lokaciji `/aiohttp` preusmjeravamo sve zahtjeve na `aiohttp-regije` mikroservis
- na lokaciji `/fastapi` preusmjeravamo sve zahtjeve na `weather-fastapi` mikroservis

Ukupna konfiguracija `nginx.conf` datoteke:

```
events {  
    worker_connections 1024;  
}  
  
http {
```

```

upstream aiohttp-regije {
    server aiohttp-regije:4000;
}

upstream weather-fastapi {
    server weather-fastapi:8000;
}

server {
    listen 80;

    location /aiohttp {
        proxy_pass http://aiohttp-regije;
        proxy_set_header Host $host;
        proxy_set_header X-Real-IP $remote_addr;
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
    }

    location /fastapi {
        proxy_pass http://weather-fastapi;
        proxy_set_header Host $host;
        proxy_set_header X-Real-IP $remote_addr;
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
    }
}
}

```

Jednostavno pokrećemo opet mikroservise koristeći `docker-compose up` naredbu:

```
docker compose up
```

Otvorite `http://localhost/aiohttp` i `http://localhost/fastapi` u web pregledniku i provjerite radi li load balancer kako treba.

Vidimo da nema grešaka, `nginx` uspješno preusmjerava zahtjeve na `aiohttp-regije` i `weather-fastapi` mikroservise.

```

nginx-1          | /docker-entrypoint.sh: /docker-entrypoint.d/ is not empty, will
attempt to perform configuration
nginx-1          | /docker-entrypoint.sh: Looking for shell scripts in /docker-
entrypoint.d/
nginx-1          | /docker-entrypoint.sh: Launching /docker-entrypoint.d/10-listen-on-
ipv6-by-default.sh
nginx-1          | 10-listen-on-ipv6-by-default.sh: info: IPv6 listen already enabled
nginx-1          | /docker-entrypoint.sh: Sourcing /docker-entrypoint.d/15-local-
resolvers.envsh
nginx-1          | /docker-entrypoint.sh: Launching /docker-entrypoint.d/20-envsubst-on-
templates.sh

```

```

nginx-1           | /docker-entrypoint.sh: Launching /docker-entrypoint.d/30-tune-worker-
processes.sh
nginx-1           | /docker-entrypoint.sh: Configuration complete; ready for start up
weather-fastapi-1 | INFO:      Started server process [1]
weather-fastapi-1 | INFO:      Waiting for application startup.
weather-fastapi-1 | INFO:      Application startup complete.
weather-fastapi-1 | INFO:      Uvicorn running on http://0.0.0.0:8000 (Press CTRL+C to
quit)
nginx-1           | 172.20.0.1 -- [22/Jan/2025:08:12:35 +0000] "GET /aiohttp HTTP/1.1"
404 14 "-" "Mozilla/5.0 (Macintosh; Intel Mac OS X 10_15_7) AppleWebKit/537.36 (KHTML,
like Gecko) Chrome/132.0.0.0 Safari/537.36"
weather-fastapi-1 | INFO:      172.20.0.4:59704 - "GET /fastapi HTTP/1.0" 404 Not Found
nginx-1           | 172.20.0.1 -- [22/Jan/2025:08:12:38 +0000] "GET /fastapi HTTP/1.1"
404 22 "-" "Mozilla/5.0 (Macintosh; Intel Mac OS X 10_15_7) AppleWebKit/537.36 (KHTML,
like Gecko) Chrome/132.0.0.0 Safari/537.36"
nginx-1           | 172.20.0.1 -- [22/Jan/2025:08:16:49 +0000] "GET /aiohttp HTTP/1.1"
404 14 "-" "Mozilla/5.0 (Macintosh; Intel Mac OS X 10_15_7) AppleWebKit/537.36 (KHTML,
like Gecko) Chrome/132.0.0.0 Safari/537.36"
weather-fastapi-1 | INFO:      172.20.0.4:33340 - "GET /fastapi HTTP/1.0" 404 Not Found
nginx-1           | 172.20.0.1 -- [22/Jan/2025:08:16:51 +0000] "GET /fastapi HTTP/1.1"
404 22 "-" "Mozilla/5.0 (Macintosh; Intel Mac OS X 10_15_7) AppleWebKit/537.36 (KHTML,
like Gecko) Chrome/132.0.0.0 Safari/537.36"

```

Vidimo da u Docker Desktopu nemamo više duple instance `weather-fastapi` i `aiohttp-regije` mikroservisa, već samo jednu instancu svakog mikroservisa, a `nginx` uspješno preusmjerava zahtjeve na njih.

Dakle, **horizontalno skaliranje** mikroservisa možemo postići kroz `docker-compose.yml` datoteku i `nginx` kao load balancer, a cijelu apstrakciju balansiranja izvršava sam `nginx` kontejner 😎

<input type="checkbox"/>	Name	Container ID	Image	Port(s)	Actions	
<input type="checkbox"/>	compose-example	-	-	-	<input type="checkbox"/>	<input type="checkbox"/> <input type="checkbox"/>
<input type="checkbox"/>	nginx-1	e10776ff5f12	nginx	80:80 ↗	<input type="checkbox"/>	<input type="checkbox"/> <input type="checkbox"/>
<input type="checkbox"/>	weather-fastapi-1	7f490a630a08	weather-fastapi:1.0	8000:8000 ↗	<input type="checkbox"/>	<input type="checkbox"/> <input type="checkbox"/>
<input type="checkbox"/>	aiohttp-regije-1	74fabd5e00d3	aiohttp-regije:1.0	8001:8001 ↗	<input type="checkbox"/>	<input type="checkbox"/> <input type="checkbox"/>

Load balancer `nginx` uspješno preusmjerava zahtjeve na `aiohttp-regije` i `weather-fastapi` mikroservise