

# 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) Docker kontejnerizacija

#7

RS

Naučili smo kako definirati asinkrone mikroservise s konkurentnom obradom podataka koji svoje funkcionalnosti otvaraju putem FastAPI i aiohttp poslužitelja. Jednom kad imamo robusne mikroservise, sljedeći korak je njihovo raspoređivanje i upravljanje resursima, bilo na lokalnom ili u proizvodnjском 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 pritom mogućnost čestih grešaka vezanih uz promjenu okruženja gdje se aplikacija izvodi. Docker je trenutno najpopularnija platforma za kontejnerizaciju aplikacija, a u ovom poglavlju naučit ćemo kako kontejnerizirati naše mikroservise koristeći Docker tehnologiju.

**Posljednje ažurirano: 15.1.2026.**

## Sadržaj

- [Raspodijeljeni sustavi \(RS\)](#)
- [\(7\) Docker kontejnerizacija](#)
  - [Sadržaj](#)
- [1. Uvod u kontejnerizaciju](#)
  - [1.1 Instalacija Docker Desktop aplikacije](#)
  - [1.2 Dockerfile](#)
    - [1.2.1 Osnovne Dockerfile naredbe](#)
  - [1.3 Kontejnerizacija osnovnog Python programa](#)
    - [1.3.1 Izgradnja Docker predloška i pokretanje kontejnera](#)
  - [1.4 Kontejnerizacija aiohttp mikroservisa](#)
    - [1.4.1 Mapiranje portova \(eng. Port mapping\)](#)

- [1.5 Tablica osnovnih Dockerfile naredbi](#)
- [1.6 Tablica osnovnih Docker naredbi](#)
- [1.7 Kontejnerizacija FastAPI mikroservisa](#)
  - [1.7.1 Implementacija jednostavnog FastAPI mikroservisa za dohvat vremenske prognoze](#)
  - [1.7.2 Kontejnerizacija FastAPI 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](#)

# 1. Uvod u kontejnerizaciju

---

**Docker** je popularna platforma otvorenog koda koja se koristi za razvoj, isporuku i pokretanje aplikacija korištenjem tehnologije kontejnerizacije (*eng. containerization*).

U računarstvu, kontejnerizacija predstavlja oblik virtualizacije na razini operacijskog sustava koji omogućuje pokretanje aplikacija u izoliranim okruženjima zvanim kontejneri (*eng. containers*).



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

U usporedbi s virtualnim strojevima (*eng. virtual machines – VM*), kontejneri su **znatno učinkovitiji**, brže se pokreću i lakše su prenose. Međutim, kontejneri izravno ovise o operacijskom sustavu domaćina (*eng. host OS*) te s njim dijele resurse, zbog čega ne pružaju potpunu izolaciju kao virtualni strojevi koji imaju vlastiti operacijski sustav.

Ipak, upravo **dijeljenje jezgre operacijskog sustava** (*eng. operating system kernel*) domaćina omogućuje brže pokretanje i manju potrošnju resursa, što kontejnerizaciju čini posebno pogodnom tehnologijom za razvoj i isporuku mikroservisnih aplikacija.

## Kontejneri VS Virtualni strojevi

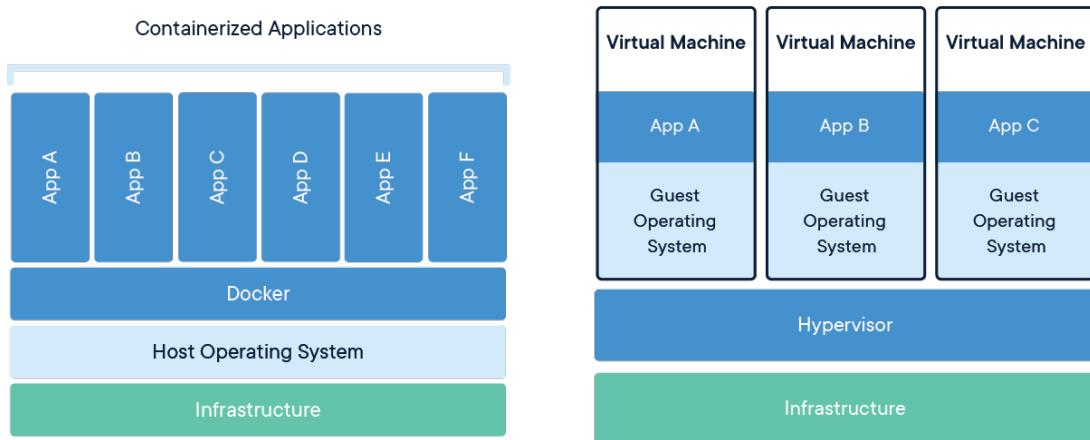
### Kontejneri

- Kontejneri su **apstrakcija aplikacijskog sloja** koja omogućuje pakiranje aplikacijskog koda i svih njegovih ovisnosti u jednu cjelinu.
- **Na istom stroju moguće je pokretati više kontejnera**, pri čemu svi dijele jezgru operacijskog sustava domaćina.
- Svaki kontejner je **izoliran** i koristi **vlastiti datotečni sustav** u vlastitom user space-u OS-a (prostor OS-a gdje se izvršavaju korisnički programi, odvojen od jezgre OS-a).
- Kontejneri zauzimaju **znatno manje memorije od virtualnih strojeva** (u pravilu od **nekoliko desetaka do nekoliko stotina MB-a**, ovisno o aplikaciji).

### Virtualni strojevi

- Virtualni strojevi predstavljaju **apstrakciju fizičkog hardvera**, koja omogućuje podjelu jednog fizičkog stroja na više virtualnih strojeva.
- Hipervizor (*eng. Hypervisor*) omogućuje pokretanje i **upravljanje više VM-ova na istom fizičkom stroju**.

- **Svaki virtualni stroj sadrži potpunu instancu operacijskog sustava**, zajedno s aplikacijama, bibliotekama i driverima, što rezultira značajnom potrošnjom memorije (**u pravilu nekoliko GB-a**).
- Zbog inicijalizacije punog operacijskog sustava, **VM-ovi se sporije pokreću u usporedbi s kontejnerima**.



Ilustracija lijevo (*Containerized Applications*) prikazuje Docker tehnologiju kao prenosnicu između operacijskog sustava domaćina i aplikacija koje se izvode unutar kontejnera (App A - App F). Ilustracija desno prikazuje virtualne strojeve koji pokreću 3 različite aplikacije (App A, App B, App C), gdje svaki VM ima vlastiti (Guest OS) operacijski sustav koji se virtualizira na fizičkom stroju (Host OS). Hipervizor je softver koji omogućuje virtualizaciju fizičkog stroja.

Virtualni strojevi i kontejneri **mogu raditi zajedno** te na taj način iskoristiti prednosti obje tehnologije.

Koga zanima više o virtualnim strojevima, dostupne su skripte [OS4](#) i [OS5](#) iz kolegija [Operacijski sustavi](#).

## 1.1 Instalacija Docker Desktop aplikacije

Fokusirat ćemo se na rad s kontejnerima putem platforme Docker. Na osobnim računalima Docker se najčešće koristi kroz grafičko korisničko sučelje **Docker Desktop**, dok se na serverima i drugim radnim stanicama u pravilu koristi Linux verzija [Docker Enginea](#), koja se koristi isključivo putem naredbenog retka.

Docker Engine je *open-source runtime* okruženje koje omogućuje izgradnju i pokretanje Docker kontejnera. Sastoji se od Docker *daemona* (`dockerd`), REST API-ja za komunikaciju s Docker *daemonom*, te CLI alata (`docker`) koji omogućuje korisnicima interakciju s Docker platformom putem naredbenog retka.

Mi ćemo primarno koristiti aplikaciju Docker Desktop, koja pojednostavljuje rad s Dockerom na osobnim računalima. Uz to, koristit ćemo i Docker CLI naredbe za izgradnju predložaka (*Docker image*) te za pokretanje, zaustavljanje i upravljanje kontejnerima.

Potrebno je prvo instalirati Docker Desktop platformu na vaše računalo:

- [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 2) koji se može instalirati preko CLI naredbe.

Otvorite [Powershell](#) ili novi [Windows Terminal](#) kao **administrator** i unesite sljedeću naredbu za instalaciju WSL-a:

```
→ wsl --install  
  
# ili ažurirati naredbom:  
  
→ wsl --update  
  
# Provjerite verziju  
  
→ wsl --version
```

Ako vam ne rade navedene CLI naredbe ili nemate administratorske ovlasti, WSL možete instalirati i ručno koristeći MSI installer: <https://github.com/microsoft/WSL/releases>

Napomena: WSL smo već instalirali na prvim vježbama, stoga ako ste to već napravili, ovaj korak možete preskočiti. U kontekstu Dockera, WSL se koristi kako bi Windows mogao pokretati Linux-bazirane Docker kontejnere, koji se nativno oslanjaju na Linux kernel. Obzirom da Windows nema nativni Linux kernel, WSL omogućuje pokretanje istog unutar Windowsa. Postoje i Windows-bazirani Docker kontejneri koji ne zahtijevaju WSL, ali se puno rjeđe koriste u praksi.

Dodatno, potrebno je omogućiti [virtualizaciju](#) za računala s Windows OS-om.

Kako biste provjerili je li virtualizacija omogućena na Windowsu, otvorite **Task Manager** i odaberite **Performance** -> **CPU** - provjerite je li opcija "Virtualization" postavljena na **Enabled**.

Utilisation	Speed	Base speed:	3.70 GHz
9%	4.46 GHz	Sockets:	1
Processes	Threads	Handles	Logical processors: 12
281	3858	127608	Virtualisation: Enabled
Up time		L1 cache:	384 KB
0:00:38:42		L2 cache:	3.0 MB
		L3 cache:	32.0 MB

Ako nije, trebate ući u **BIOS/UEFI** postavke vašeg računala i omogućiti virtualizaciju (obično se naziva **Intel VT-x** za Intel procesore ili **AMD-V** za AMD procesore).

Ovisno o proizvođaču matične 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 Apple računala gdje je virtualizacija već omogućena**).

- Najbolji način je pretražiti na internetu kako pristupiti BIOS-u za vaš model računala/matične ploče. 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 setup and your planned use case.

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

- WSL version 2.1.5 or later. To check your version, see [WSL: Verification and setup](#)
- Windows 10 64-bit: Enterprise, Pro, or Education version 22H2 (build 19045).
- Windows 11 64-bit: Enterprise, Pro, or Education version 23H2 (build 22631) 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/UEFI. For more information, see [Virtualization](#).

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

Na Windowsu je moguće koristiti **WSL** (Windows Subsystem for Linux) ili **Hyper-V Windows-native** platformu za virtualizaciju, detaljne upute potražite ovdje: <https://docs.docker.com/desktop/setup/install/windows-install/>. Međutim, preporuka je koristiti WSL jer je lakši i bolji integriran s Linux baziranim kontejnerima.

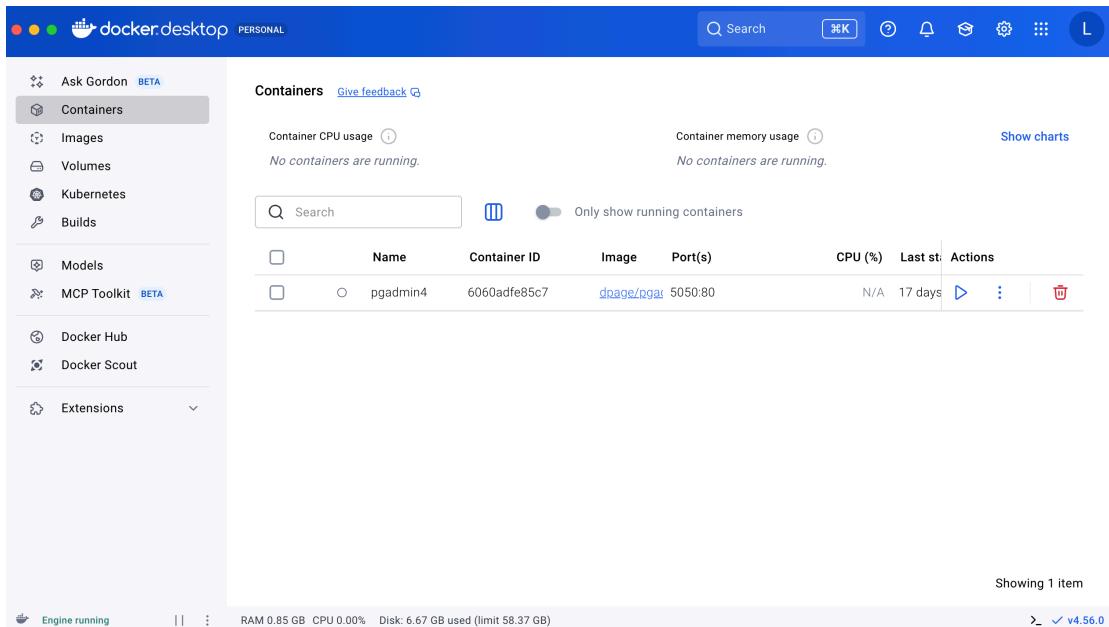
Docker je moguće koristiti i na **Linux** (dostupno za: Ubuntu, Debian, RHEL, Fedora, ...) i **macOS** (dostupno za: Apple silicon, Intel chip) operacijskim sustavima bez dodatnih postavki. [Na Linuxu možete instalirati Docker i bez grafičkog sučelja preko terminala](#), međutim za početnike je preporuka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

Nakon što ste uspješno instalirali **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 izraditi na [Docker Hubu](#).

Što je sad Docker Hub? Docker Hub je javni repozitorij Docker predložaka (Docker images) koji održava Docker Inc. i zajednica. Na Docker Hubu možete pronaći veliki broj gotovih predložaka koje možete koristiti kao bazne za vaše aplikacije, ali i dijeliti vlastite predloške s drugima ili ih samo pohraniti za vlastitu, privatnu upotrebu.



Grafičko sučelje Docker Desktop aplikacije nakon uspješne instalacije i prijave

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

1. **Containers** – prikaz svih trenutno pokrenutih kontejnera (eng. *Docker containers*). Docker kontejner predstavlja instancu izgrađenog Docker predloška (*image*) koja se izvršava u izoliranom okruženju.
2. **Images** – prikaz svih preuzetih Docker predložaka (eng. *Docker images*). Docker image je nepromjenjivi predložak koji definira kako se kontejner gradi i pokreće.
3. **Volumes** – prikaz svih Docker volumena (eng. *Docker volumes*). Docker volume služi za trajnu pohranu podataka, budući da se podaci unutar kontejnera brišu prilikom njegovog gašenja ili uklanjanja.
4. **Kubernetes** – služi za upravljanje lokalnom Kubernetes instancom (neće se koristiti u okviru ove skripte).
5. **Builds** – prikaz svih izvršenih Docker build procesa (eng. *Docker builds*). Ovdje su evidentirani svi buildovi koji su se izvršavali na vašem računalu.
6. **Docker Hub** – integracija s Docker Hub servisom za preuzimanje i dijeljenje Docker predložaka.
7. **Docker Scout** – napredna analiza pohranjenih Docker predložaka s ciljem identifikacije potencijalnih sigurnosnih ranjivosti (eng. *vulnerabilities*).
8. **Extensions** – dodatne ekstenzije za Docker Desktop aplikaciju (trenutno ih nećemo koristiti).

Za početak će nam najzanimljiviji biti **Container** i **Images** tabovi koji omogućuju pregled i upravljanje kontejnerima i predlošcima. *Stay tuned!* 😊

Napomena: U novim verzijama Dockera dodani su i AI alati za pomoć pri radu s Dockerom, poput **Gordon AI chatbota** za upravljanje Dockerom putem prirodnog jezika, **Models** kategorije za brzo preuzimanje velikih jezičnih modela u obliku kontejnera i **MCP Toolkit** za instalaciju **MCP** poslužitelja u obliku Docker kontejnera. U ovoj skripti neće se koristiti navedeni alati, ali su svakako zanimljivi za istraživanje.

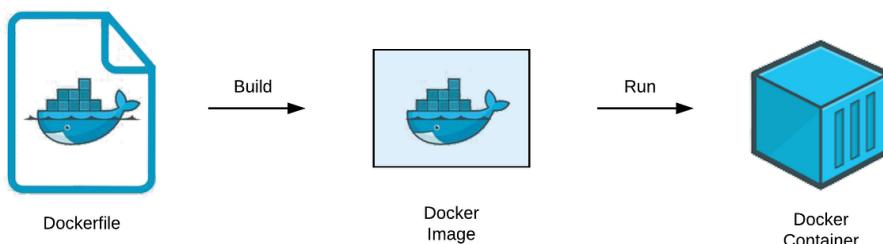
## 1.2 Dockerfile

**Dockerfile** je tekstualna datoteka koju koristimo za definiranje predložaka kontejnera. **Predložak** (Docker image) je binarni artefakt sastavljen od niza slojeva koji sadrže sve potrebne komponente za pokretanje aplikacije unutar kontejnera. **Dockerfile definira kako se gradi taj predložak.**

**Dockerfile** može biti vrlo jednostavan, ali i vrlo složen, ovisno koliko je kompleksna aplikacija koju želimo kontejnerizirati i koje su njene ovisnosti.

U okviru ovog kolegija naučit ćemo kako definirati **Dockerfile**-ove za mikroservise koje razvijamo u Pythonu, konkretno koristeći **FastAPI** i **aiohttp**. Iako ćemo se oslanjati na te tehnologije kao primjere, **ista načela vrijede i za druge programske jezike, tehnologije te različite oblike softverskih rješenja** - ne nužno isključivo za web poslužitelje ili samo Python aplikacije.

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 (**Docker Image**) za izradu kontejnera, a **Docker Container** je radna instanca tog predloška koja se pokreće u izoliranom okruženju

Dockerfile definiramo **doslovnim nazivom datoteke**: **Dockerfile** (bez ekstenzije), a on se mora nalaziti u **korijenskom direktoriju našeg mikroservisa/aplikacije**.

**Sintaksa:**

```
# komentar
INSTRUCTION argument
```

- **INSTRUCTION** - naredba koja se izvršava prilikom izgradnje Docker predloška
- **argument** - argument naredbe

## 1.2.1 Osnovne Dockerfile naredbe

### FROM

- Svrha: definira **bazni predložak** (eng. *Base image*) na temelju kojeg "se gradi" naš Docker predložak
- Svaki Docker predložak mora početi s **FROM** naredbom - dakle to je prva naredba u Dockerfileu

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

Uobičajeno je koristiti službene verzije baznih predložaka koje su dostupne na [Docker Hubu](#). Konkretno za [Python ih ima jako puno](#), npr. `python:3` za Python 3.x aplikacije, `node:<tag>` za Node.js aplikacije itd.

Primjer: Koristimo Python 3 kao bazni predložak za naš Docker predložak:

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

Napomena: Korištenje generičkih tagova poput `python:3` u produkciji je loša praksa budući da ćemo na taj način preuzimati uvijek najnoviju verziju Pythona 3.x, što vrlo lako može dovesti do nekompatibilnosti s našim mikroservisom. Bolja praksa je koristiti točne verzije poput `python:3.11.4` i/ili optimizirane varijante - više o tome u nastavku.

## WORKDIR

- postavlja radni direktorij **unutar datotečnog sustava kontejnera**
- sve druge naredbe nakon `WORKDIR` odnose se na taj direktorij, odnosno sve relativne putanje će biti u odnosu 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 **iz datotečnog sustava domaćina** (eng. *host filesystem*) **u datotečni sustav kontejnera** (eng. *container filesystem*)
- 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.
- ako postoji više `CMD` naredbi u Dockerfileu, samo će se posljednja uzeti u obzir
- 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 očekivanog 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
```

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

Napomena: U praksi je bolje koristiti `requirements.txt` datoteku za instalaciju svih Python ovisnosti odjednom, umjesto pojedinačne `RUN pip install <package>` naredbe za svaki paket. Više o tome u nastavku.

## EXPOSE

- služi za dokumentaciju porta na kojem će kontejner slušati u mreži.

- **neće otvoriti port na hostu**, već samo **dokumentira** koji port koristi kontejner
- port se na ovaj način dokumentira u metapodacima predloška, uključujući unutar `docker inspect` naredbe, ali i za komunikaciju između developera i korisnika predloška

```
EXPOSE <port>
```

**Primjer:** dokumentiranje porta `8080`

```
# dokumentira port 8080 na kojem se očekuje da kontejner sluša
EXPOSE 8080
```

---

Dakle, osnovne naredbe su `FROM`, `WORKDIR`, `COPY`, `CMD`, `RUN` i `EXPOSE`.

Nastavljamo s jednostavnim primjerom kontejnerizacije najjednostavnijeg mogućeg Python programa.

## 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.

Napisat ćemo super jednostavan Python program koji ispisuje "Hello, World!" u terminalu:

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

Program jednostavno pokrećemo 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`. Naredba python se izvršava u odnosu na WORKDIR.

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

Brisanjem komentara, `Dockerfile` svodimo na sljedeće četiri linije:

```
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

### 1.3.1 Izgradnja Docker predloška i pokretanje kontejnera

Kako bismo **izgradili predložak** iz definiranog Dockerfile-a, koristimo sljedeću CLI naredbu:

**Sintaksa:**

```
docker build -t <ime>:<verzija> <putanja/do/Dockerfilea>
```

- opcionalnom zastavicom `-t` možemo odrediti ime i verziju našeg predloška u formatu `<ime>:<verzija>`, npr. `hello-world:1.0`
- `<putanja/do/Dockerfilea>` je putanja do direktorija gdje se nalazi Dockerfile, uobičajeno je to trenutni direktorij, što označavamo s točkom `.`

```
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 Dockerfile-a 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 nije pokrenut. Pokrenite Docker Desktop aplikaciju i pokušajte ponovno.

Prva izgradnja Docker predloška potrajat će neko vrijeme budući da je prvi korak preuzimanje i priprema baznog predloška `python:3`. Nakon toga, izgradnja će ići pun brže jer Docker kešira prethodno preuzete slojeve.

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

Containers Images Volumes Builds Docker Hub Docker Scout Extensions PGAdmin4 Volumes

Images Give feedback View and manage your local and Docker Hub images. Learn more Local Hub repositories 1.02 GB / 4.35 GB in use 19 images Last refresh: 2 hours ago

hello-world

Name	Tag	Created	Size	Actions	Image ID
hello-world	1.0	2 minutes ago	1.02 GB		3d534d406bcb

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!

Osim toga, možemo provjeriti i preko terminala je li predložak uspješno izgrađen naredbom `docker images`:

```
→ docker images
```

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

### Sintaksa:

```
→ docker run <naziv_predloska>:<verzija>
```

U našem slučaju, pokrećemo predložak `hello-world:1.0`

```
→ docker run hello-world:1.0
```

**Napomena:** Naredbu je moguće pokrenuti u bilo kojem terminalu, ne samo u terminalu gdje se nalazi `Dockerfile`. Ovo je korisno jer jednom kad je predložak izgrađen, možemo ga pokretati s bilo kojeg mesta na našem računalu, dakle **globalno**.

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

Containers Give feedback View all your running containers and applications. Learn more

Container CPU usage: No containers are running.

Container memory usage: No containers are running.

Show charts

Search

Only show running containers

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

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

Pokretanjem kontejnera na ovaj način, Docker automatski dodjeljuje **naziv i ID kontejneru** koji predstavljaju identifikator trenutne radne instance tog Docker predložka.

ID ili naziv kontejnera možemo iskoristiti za upravljanje kontejnerom, ili provjeru aktivnih *logova* (ispisa u terminalu) tog kontejnera. Za to koristimo naredbu `docker logs`:

```
→ docker logs <container_id_or_name>

# Potražite ID kontejnera u Docker Desktop aplikaciji:

→ docker logs 5ceal8cf300d295cd52a8ac4295f04342ebb6a2ac42551b73389e7764d09e44

# Ispis logova kontejnera
Hello, World!
```

## 1.4 Kontejnerizacija aiohttp mikroservisa

Na opisani način možemo kontejnerizirati bilo koji Python program koji se izvršava sinkrono. Međutim, kod kontejnerizacije asinkronog mikroservisa s ugrađenim poslužiteljem, odnosno svake aplikacije koja svoje funkcionalnosti izlaže putem komunikacijskog protokola, poput aiohttp mikroservisa koji smo razvijali na prethodnim vježbama, **potrebno je primijeniti nešto drugačiji pristup izradi Dockerfile-a**.

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

Napraviti ćemo novi direktorij `aiohttp-microservice` u kojem ćemo izraditi 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) i naziv virtualnog okruženja:

```
→ 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`

```
# aiohttp-microservice/app.py  
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`.

**Naš program je sada složeniji:** imamo asinkroni mikroservis koji sluša na portu `8080` kroz `aiohttp` poslužitelj, stoga je potrebno definirati nekoliko dodatnih naredbi u `Dockerfile`-u. 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:

```
# Dockerfile
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 prilikom pokretanja kontejneriziranog mikroservisa ako se verzija biblioteke i verzija Pythona ne poklapaju ili se promijeni sintaksa između verzija pa naš kod postaje nekompatibilan.

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

**Ovisnosti je uobičajeno definirati u posebnoj tekstualnoj 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**:

**Sintaksa:**

```
→ pip freeze  
  
# ili direktno u datoteku:  
  
→ pip freeze > requirements.txt # Ovo je overwrite operacija, za append koristimo operator  
>> umjesto >
```

Primjer: Izlistane ovisnosti za naš `aiohttp` mikroservis:

```
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 koristimo naredbu iznad da ih automatski pohranimo u datoteku.

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 iz poglavlja 1.3  
FROM python:3  
WORKDIR /app  
COPY app.py /app  
CMD [ "python", "app.py" ]
```

1. korak je zamjena `python:3` baznog predloška 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:

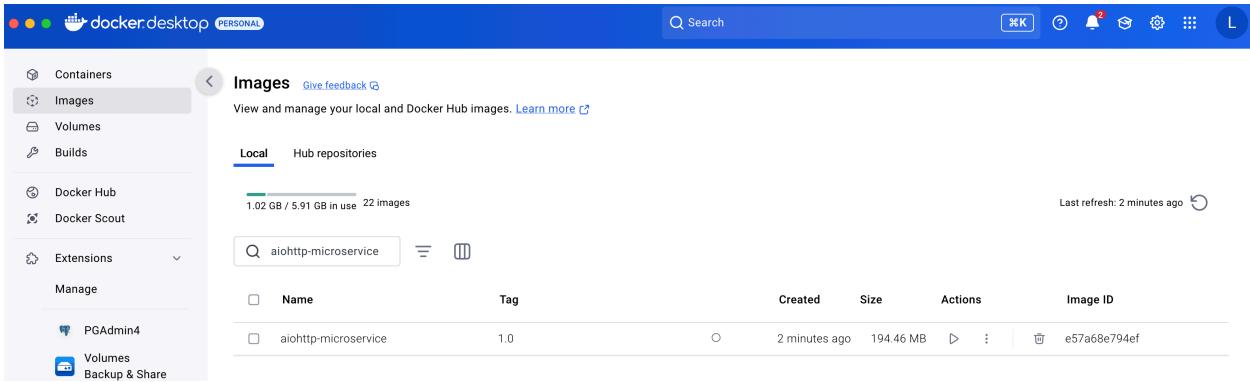
```
FROM python:3.11-slim
WORKDIR /app
COPY . /app
RUN pip install -r requirements.txt
EXPOSE 8080
CMD ["python", "app.py"]
```

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`



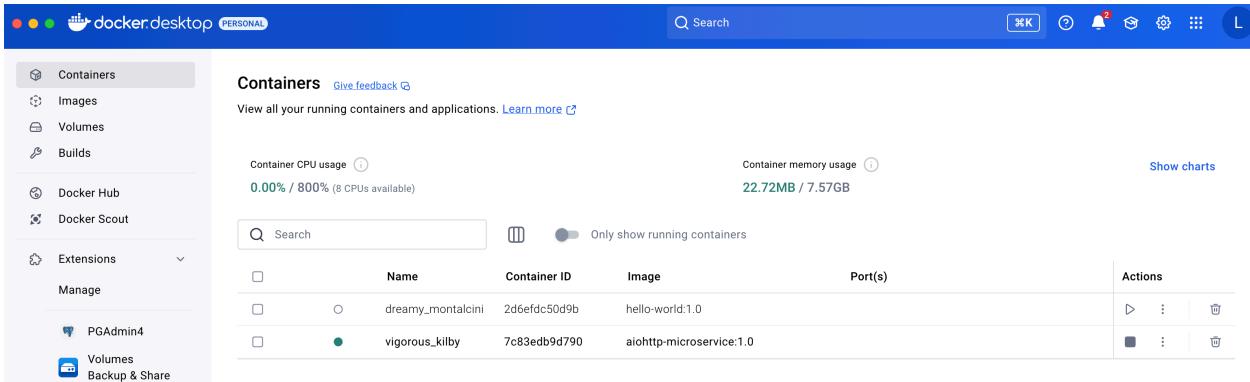
Otvorite Docker desktop i provjerite je li predložak uspješno izgrađen. Trebali biste vidjeti novi predložak `aiohttp-microservice:1.0` pod tabom `Images`.

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!



Kontejner `aiohttp-microservice:1.0` je uspješno pokrenut i mikroservis sluša na portu `8080` unutar kontejnera

**Važna napomena!** Prilikom izgradnje Python Docker predložaka, NIJE POTREBNO unutar `Dockerfile`-a izraditi virtualno okruženje (npr. `venv` ili `conda`), budući da je svaki Docker kontejner već izolirano okruženje samo za sebe sa vlastitim datotečnim sustavom i instaliranim točno onim paketima koji su potrebni za pokretanje aplikacije unutar kontejnera. Stoga, instalacija virtualnog okruženja unutar kontejnera je suvišna i predstavlja lošu praksu.

## 1.4.1 Mapiranje portova (eng. Port mapping)

Naredbom `docker ps` možemo vidjeti **sve pokrenute kontejnere na našem računalu**:

```
→ docker ps
```

Ispisuje **aktivne** (eng. *running*) 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 na temelju kojeg je kontejner pokrenut
- `COMMAND` - naredba koja se izvršava prilikom pokretanja kontejnera (definirana u `CMD` naredbi)
- `CREATED` - prije koliko je vremena kontejner izrađen
- `STATUS` - trenutno stanje kontejnera (eng. *runtime status*)
- `PORTS` - portovi na kojima kontejner sluša
- `NAMES` - proizvoljni naziv kontejnera koji je Docker automatski generirao/ili koji smo mu mi dodijelili

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? 😕

Napomena: Preostale (*non-running*) kontejnere možemo vidjeti dodavanjem zastavice `-a` naredbi `docker ps`: `docker ps -a`

► Spoiler alert! Odgovor na pitanje

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

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 znamo da 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
```

Ako u `Dockerfile`-u navedemo dokumentiramo jedan ili više portova pomoću naredbe `EXPOSE`, možemo ih mapirati na proizvoljne portove domaćina na sljedeći način:

```
→ docker run -P aiohttp-microservice:1.0 # Mapiraj container port definiran unutar EXPOSE na random dostupan host port
```

- ovo može biti korisno u nekim situacijama, npr. kada aplikacija koristi više portova, a mi ne želimo ručno mapirati svaki od njih

Zastavicom `--name` moguće je i dodijeliti ime kontejneru; **u suprotnom, Docker će automatski generirati proizvoljno ime** (kao u našem slučaju `trusting_spence`):

```
→ 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>`:

### Sintaksa:

```
→ docker stop <container_id_or_name>
```

Primjer:

```
→ docker stop a604911ac56a
```

```
# ili preko imena (NE IMENA PREDLOŠKA VEĆ IMENA KONTEJNERA):
```

```
→ docker stop aiohttp-microservice
```

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

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS
PORTS	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 (Docker Desktop - tab `Containers`)

Ipak, stvari niti sada neće raditi! Što mislite zašto? 🤔

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

```
# aiohttp-microservice/app.py
web.run_app(app, host='localhost', port=8080)
```

- "slušaj na `localhost hostu`". `localhost` je ustvari adresa petlje tj. `loopback` adresa mrežnog sučelja na računalu], a najčešće se asocira s IPv4 adresom `127.0.0.1`. Adresa petlje predstavlja internu adresu koja preusmjerava mrežni promet natrag prema istom računalu.
- port je `8080` i to je u redu.

**Problem:** mikroservis se pakira u kontejner, a kontejner je izolirano okruženje, odnosno **ne dijeli mrežni adapter 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`.

Adresa `0.0.0.0` je specijalna adresa koja označava "ovu mrežu", tj. sve lokalne adrese - dozvoljava poslužiteljima da slušaju na svim dostupnim mrežnim sučeljima.

**U produkcijskim okruženjima, ovo je 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 samom mikroservisu:

```
# aiohttp-microservice/app.py
web.run_app(app, host='0.0.0.0', port=8080) # zamjenili 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** - predstavlja "snapshot" stanja u trenutku izgradnje.

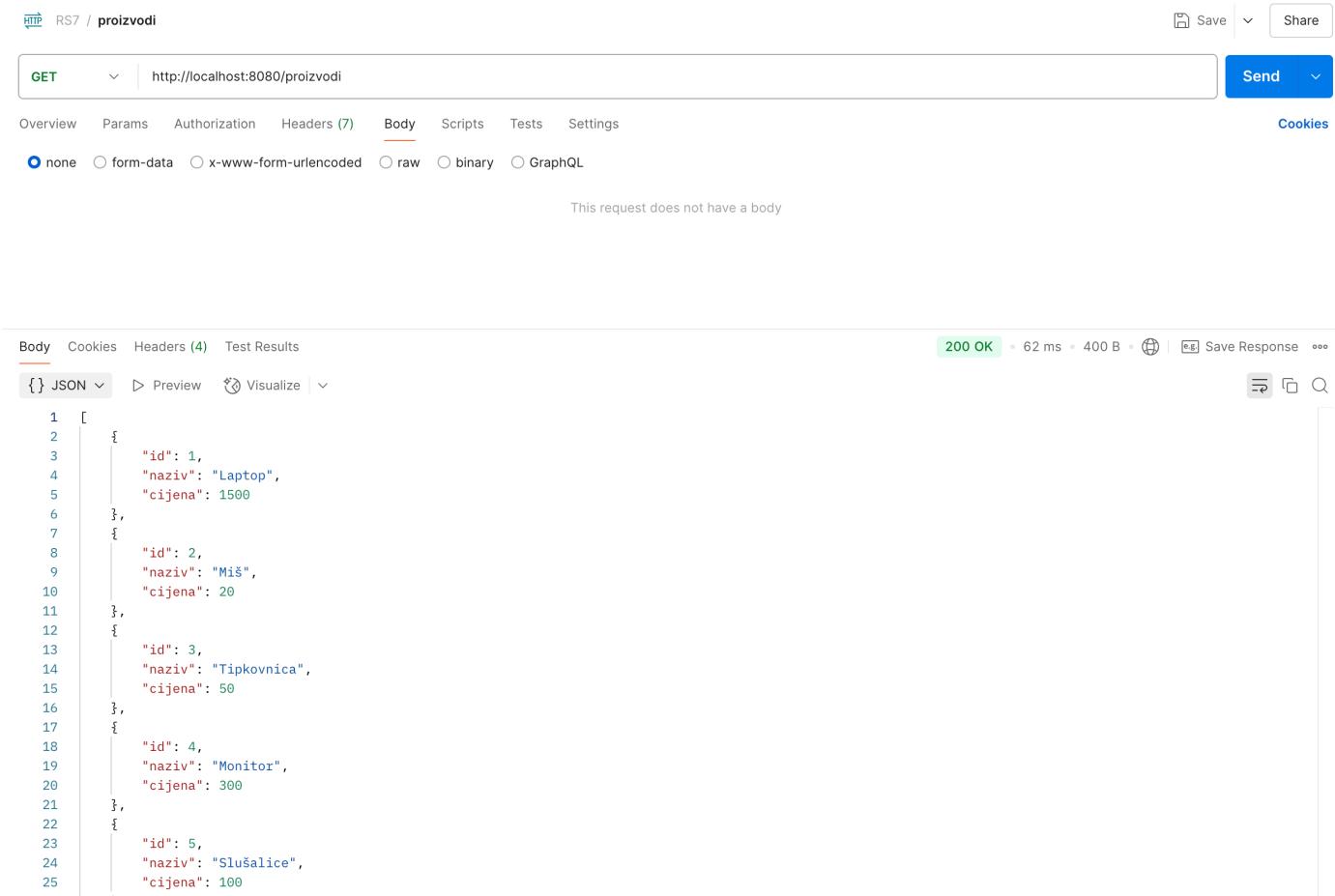
Izgradimo ponovo predložak:

```
→ 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.



The screenshot shows a Postman request to `localhost:8080/proizvodi`. The response body is a JSON array containing five products:

```
[{"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}]
```

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 ispis)
- 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, there's a sidebar with options like Containers, Images, Volumes, Builds, Docker Hub, Docker Scout, Extensions, PGAdmin4, and Volumes Backup & Share. The main area shows a container named 'aiohttp-microservice' with ID '87b6899b267f'. It's running on port 8080. The 'Logs' tab is selected, displaying the output: '2025-01-21 20:44:25 ====== Running on http://0.0.0.0:8080 ======' and '2025-01-21 20:44:25 (Press CTRL+C to quit)'. There are also tabs for Inspect, Bind mounts, Exec, Files, and Stats.

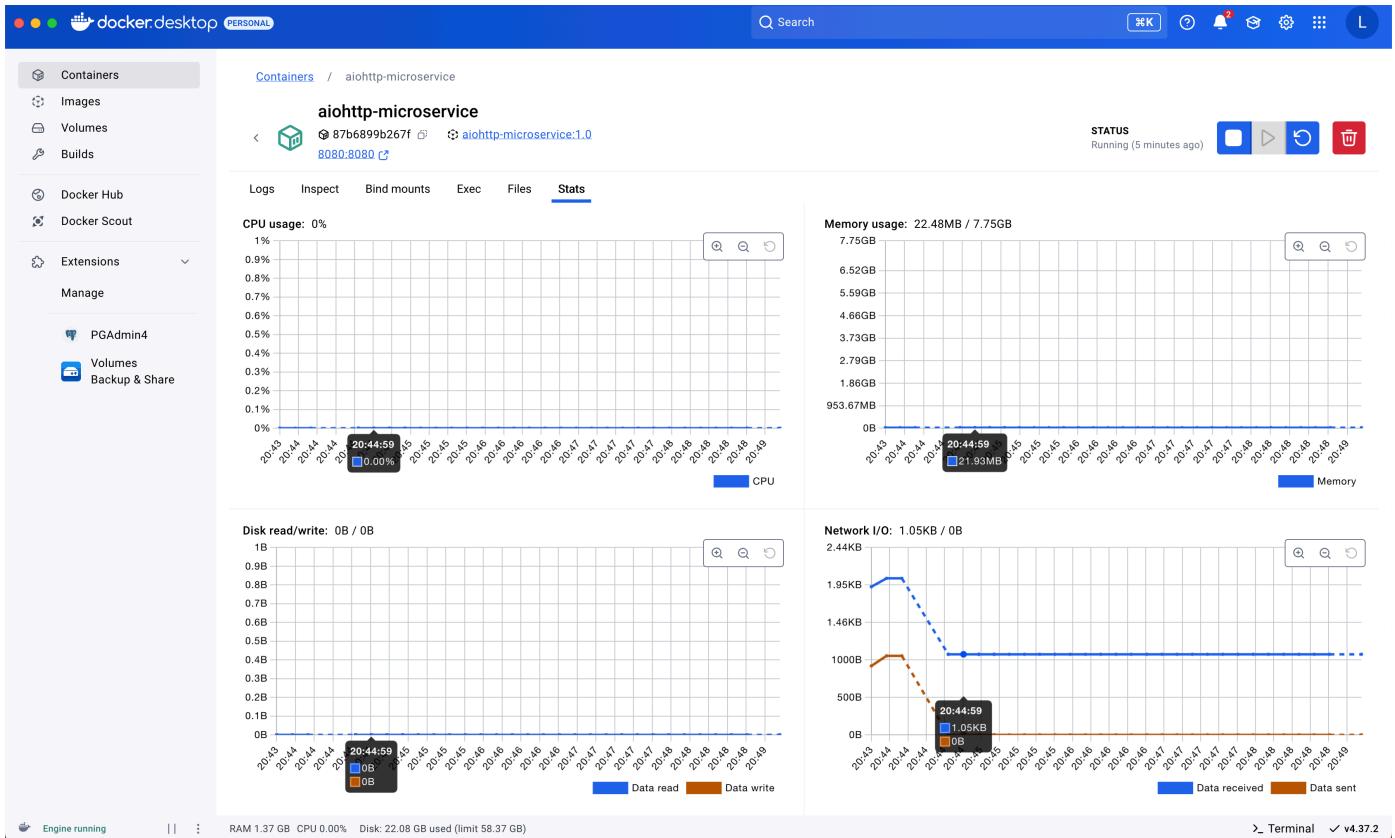
Pregled logova aktivnog kontejnera iz Docker Desktop sučelja; može se pregledavati i naredbom

```
docker logs <container_id_or_name>
```

The screenshot shows the 'Files' tab for the 'aiohttp-microservice' container. It lists files and directories within the container's file system. The 'app' directory is expanded, showing 'app.py', 'Dockerfile', and 'requirements.txt'. Other directories like .dockerenv, boot, dev, etc., are also listed. The table includes columns for Name, Note, Size, Last modified, and Mode.

Name	Note	Size	Last modified	Mode
.dockerenv		0 Bytes	4 minutes ago	-rwxr-xr-x
app			4 minutes ago	drwxr-xr-x
app.py		945 Bytes	18 minutes ago	-rw-r--r--
Dockerfile		119 Bytes	46 minutes ago	-rw-r--r--
requirements.txt		181 Bytes	22 hours ago	-rw-r--r--
bin -> usr/bin		7 Bytes	9 days ago	Lrwxrwxrwx
boot			21 days ago	drwxr-xr-x
dev			3 minutes ago	drwxr-xr-x
etc			4 minutes ago	drwxr-xr-x
home			21 days ago	drwxr-xr-x
lib -> usr/lib		7 Bytes	9 days ago	Lrwxrwxrwx
media			9 days ago	drwxr-xr-x
mnt			9 days ago	drwxr-xr-x
opt			9 days ago	drwxr-xr-x
proc			3 minutes ago	dr-xr-xr-
root			4 minutes ago	drwx-----
run			9 days ago	drwxr-xr-x
sbin -> usr/sbin		8 Bytes	9 days ago	Lrwxrwxrwx
srv			9 days ago	drwxr-xr-x
sys			3 minutes ago	dr-xr-xr-x
tmp			4 minutes ago	dtrwxrwxrwx

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/O i disk I/O**.

Uočite da je kod grafa `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.

Uspješno smo kontejnerizirali `aiohttp` mikroservis i pristupili mu s našeg računala putem mapiranog porta!

## 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**:

Dockerfile 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. Ako ne postoji, Docker će ga automatski izraditi.	<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. Može se nadjačati kroz Docker CLI tijekom build faze.	<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>	Dokumentira na kojim će portovima kontejner slušati. <b>Ne otvara port!</b>	<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**.

Docker CLI Naredba	Sintaksa	Objašnjenje	Primjer
<b>docker build</b>	<code>docker build -t &lt;image_name&gt;:&lt;tag&gt; &lt;path&gt;</code>	Kreira Docker predložak iz <code>Dockerfile</code> -a i dodjeljuje mu ime i tag (opcionalno). Nekad je korisno koristiti i opcionalnu zastavicu <code>--no-cache</code> koja će preskočiti sve keširane slojeve prilikom izgradnje.	<code>docker build -t myapp:1.0 .</code>
<b>docker 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 na temelju Docker predloška, mapira portove ( <code>-p</code> ) i daje ime ( <code>--name</code> ) kontejneru.	<code>docker run -p 8080:80 --name mycontainer myapp</code>
<b>docker ps</b>	<code>docker ps</code> za aktivne, <code>docker ps -a</code> za sve	Prikazuje listu trenutno aktivnih kontejnera ili svih kontejnera (-a)	<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 predlošku	<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>
<b>docker start</b>	<code>docker start &lt;container_id_or_name&gt;</code>	Pokreće zaustavljeni kontejner.	<code>docker start mycontainer</code>
<b>docker logs</b>	<code>docker logs &lt;container_id_or_name&gt;</code>	Prikazuje logove kontejnera.	<code>docker logs mycontainer</code>
<b>docker images</b>	<code>docker images</code>	Prikazuje listu svih lokalno spremljenih Docker predložaka.	<code>docker images</code>
<b>docker rmi</b>	<code>docker rmi &lt;image_name&gt;:&lt;tag&gt;</code>	Briše lokalno spremljeni Docker predložak.	<code>docker rmi myapp:1.0</code>

# 1.7 Kontejnerizacija FastAPI mikroservisa

Pokazat ćemo kako kontejnerizirati i nešto složenije mikroservise, poput `FastAPI` mikroservisa sa svim njegovim ovisnostima. Kod `aiohttp` proces je bio jednostavniji jer nam je jedina ovisnost bila `aiohttp` paket, dok su preostali 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 i koristite ORM, 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.

Nama se u tom kontekstu ne mijenja puno toga, osim što ćemo morati definirati više ovisnosti u `requirements.txt` datoteci i izmijeniti naredbu za pokretanje mikroservisa.

## 1.7.1 Implementacija jednostavnog FastAPI mikroservisa za dohvat vremenske prognoze

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)

Kako bismo proučili strukturu, možemo ju otvoriti u web pregledniku ili `curl` HTTP klijentom poslati zahtjev i preuzeti rezultat u datoteku (zastavica `-o`):

```
→ curl -o 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>
```

Nadalje, glavni element je `section` koji sadrži više `station` elemenata, gdje svaki `station` predstavlja mjernu lokaciju s pripadajućim podacima o vremenu:

```

<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">
        <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:

```
# weather-fastapi/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 atributе:

- `name` - ime mjernog mjesta
- `lon` - geografska dužina
- `lat` - geografska širina
- `vrijeme` - prognoza vremena (npr. 4 - oblačno, 6 - sunčano)
- `Tmn` - minimalna temperatura
- `Tmx` - maksimalna temperatura
- `wind` - stupanska 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:

```
# weather-fastapi/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:

```
# weather-fastapi/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 - 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`:

```
# weather-fastapi/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:

```
# weather-fastapi/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 `xml.etree.ElementTree` iz paketa `xml`. Ovaj modul je dio standardne biblioteke Pythona, tako da nije potrebna dodatna instalacija.

```
# weather-fastapi/main.py
import xml.etree.ElementTree as ET
```

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

```
# weather-fastapi/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 dohvatanju 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

```
# weather-fastapi/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 biste vidjeti dokumentiranu rutu `/vrijeme` koja vraća podatke o vremenu u JSON formatu.

The screenshot shows the Swagger UI interface for the `/vrijeme` endpoint. The endpoint is a `GET` method that returns current weather data in JSON format. The documentation page includes:

- Parameters:** No parameters.
- Responses:**
  - Curl:** A command-line tool to execute the request. The command is:
`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:** A table showing the response code (200) and the response body content.

The response body content is a JSON array containing two elements, each representing a city with its minimum and maximum temperature and wind speed:

```

[{"mjesto": "sredinsjko", "temperatura_min": -1, "temperatura_max": 4, "vjetar": 6}, {"mjesto": "istocna", "temperatura_min": -1, "temperatura_max": 3, "vjetar": 0}]

```

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

## 1.7.2 Kontejnerizacija FastAPI 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:

```
# weather-fastapi/requirements.txt
aiohappyeyeballs==2.4.4
aiohttp==3.11.11
aiosignal==1.3.2
annotated-types==0.7.0
anyio==4.8.0
email_validator==2.2.0
fastapi==0.115.6
fastapi-cli==0.0.7

...
uvicorn==0.34.0
watchfiles==1.0.4
websockets==14.2
yarl==1.18.3
```

Napraviti ćemo `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"]
```

Izgradite 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
```

```
lukablasovic - docker run -p 8000:8000 --name weather-fastapi weather-fastapi:1.0 -- 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://0.0.0:8000 (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

- Definirajte jednostavni `aiohttp` mikroservis `authAPI` koji će slušati na portu `9000`. Mikroservis pohranjuje *in-memory* podatke o korisnicima, s hashiranim lozinkama. U komentarima pored svakog zapisa možete pronaći stvarnu lozinku koja je korištena za generiranje hash vrijednosti funkcijom `hash_data`.

```
import hashlib

korisnici = [
    {"korisnicko_ime": "admin", "lozinka_hash" :
     "8d43d8eb44484414d61a18659b443fbfe52399510da4689d5352bd9631c6c51b"}, # lozinka =
     "lozinka123"
    {"korisnicko_ime": "markoMaric", "lozinka_hash" :
     "5493c883d2b943587ea09ab8244de7a0a88d331a1da9db8498d301ca315d74fa"}, # lozinka =
     "markoKralj123"
    {"korisnicko_ime": "ivanHorvat", "lozinka_hash" :
     "a31d1897eb84d8a6952f2c758cdc72e240e6d6d752b33f23d15fd9a53ae7c302"}, # lozinka =
     "1111111111lozinka_123"
    {"korisnicko_ime": "Nada000",
     "lozinka_hash": "492f3f38d6b5d3ca859514e250e25ba65935bcdd9f4f40c124b773fe536fee7d"} # lozinka =
     "blablabla"
]

def hash_data(data: str) -> str:
    return hashlib.sha256(data.encode()).hexdigest()
```

- implementirajte rutu `POST /register` koja dodaje novog korisnika u listu korisnika. Pohranite samo hashiranu lozinku korisnika.
- implementirajte rutu `POST /login` koja pronađe korisnika po korisničkom imenu u listi korisnika i provjerava je li unesena lozinka u tijelu HTTP zahtjeva ispravna, odnosno podudaraju li se hash vrijednosti. Ako se pokuša prijaviti korisnik koji ne postoji, vratite odgovarajući statusni kod i poruku. Ako se lozinke ne podudaraju, vratite odgovarajući statusni kod i poruku.

- definirajte `Dockerfile` za `authAPI` mikroservis i pokrenite ga u Docker kontejneru. Servis treba slušati na portu `9000` domaćina.
2. Definirajte `FastAPI` mikroservis `socialAPI` koji će služiti za dohvaćanje izmišljenih objava na društvenoj mreži. Objave su pohranjene u listi rječnika, gdje svaki rječnik predstavlja jednu objavu. Svaka objava ima sljedeće atribute:
- `id` - jedinstveni identifikator objave (integer)
  - `korisnik` - korisničko ime autora objave (do 20 znakova)
  - `tekst` - tekst objave (do 280 znakova)
  - `vrijeme` - vrijeme kada je objava napravljena (`timestamp`)
- definirajte odgovarajuće Pydantic modele za izradu nove objave i dohvaćanje objave.
  - implementirajte rutu `POST /objava` koja dodaje novu objavu u listu objava. Prije dodavanja u listu, obavezno validirajte ulazne podatke. Prilikom dodavanja objave, sve vrijednosti su obavezne, osim `id` atributa koji se automatski dodjeljuje. Logiku dodjeljivanja jedinstvenog identifikatora možete implementirati sami po želji.
  - implementirajte rutu `GET /objava/{id}` koja dohvaća objavu po jedinstvenom identifikatoru.
  - implementirajte rutu `GET /korisnici/{korisnik}/objave` koja dohvaća sve objave korisnika s određenim korisničkim imenom.
  - definirajte `Dockerfile` za `socialAPI` mikroservis i pokrenite ga u Docker kontejneru. Servis treba slušati na portu `3500` domaćina.

## 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 mikroservisne arhitekture koje se sastoje od više mikroservisa, baza podataka, korisničkih sučelja, a sve to možemo pokrenuti jednom naredbom - kao da se radi o jednoj aplikaciji.

Datoteka koju koristi Docker Compose za definiranje kontejnera i drugih resursa naziva se `docker-compose.yml` i predstavlja **glavnu konfiguracijsku datoteku** za Docker Compose alat.

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

- `frontend` Docker kontejner s frontend aplikacijom (npr. Vue.js) (u praksi bi se koristio Nginx ili neki drugi web server za posluživanje statičkih datoteka frontenda - više u sljedećoj skripti)
- `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:

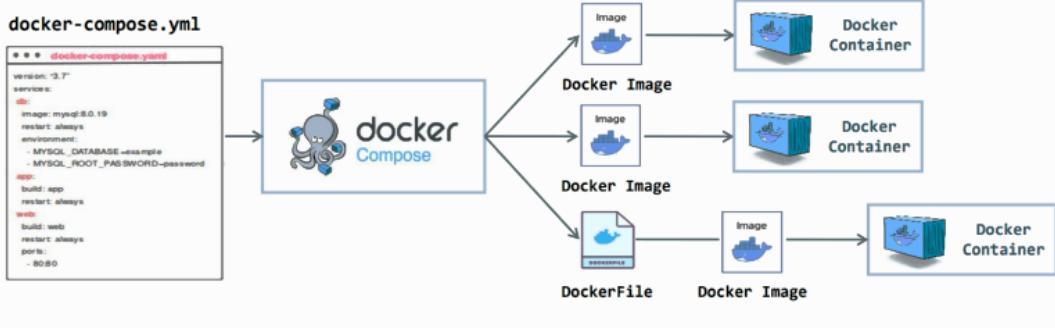
- `fileAPI` Docker kontejner s mikroservisom za pohranu i dijeljenje datoteka
- `encryptionAPI` Docker kontejner s mikroservisom za enkripciju i dekripciju datoteka
- `userAPI` Docker kontejner s mikroservisom za upravljanje korisnicima
- `notificationAPI` Docker kontejner s mikroservisom za obavijesti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

Uočite zajedničke termine u svim ovim primjerima: to su **Raspodijeljeni sustav, Mikroservisi i Docker kontejner**.

**Raspodijeljeni sustav** čini skup više skalabilnih i nezavisnih mikroservisa, pri čemu je svaki od njih poželjno zapakirati u zaseban Docker kontejner - ovo je korisno iz više razloga:

- **Izolacija:** Svaki mikroservis radi u svom izoliranom okruženju, što smanjuje rizik od sukoba između različitih servisa.
- **Skalabilnost:** Svaki mikroservis se može skalirati neovisno, ovisno o potrebama opterećenja.
- **Jednostavnije upravljanje:** Docker olakšava upravljanje ovisnostima i konfiguracijama za svaki mikroservis.
- **Brze izmjene:** Izgradnja kontejnera je puno brža u usporedbi s postavljanjem cijelog virtualnog stroja. Izmjene u mikroservisima se mogu brzo testirati, implementirati i distribuirati u produkcijsko okruženje.

**Važno!** Ipak, pakiranjem mikroservisa u jedinstvenu cjelinu pomoću Docker Composea uvodi se **centralizirani model upravljanja sustavom**, u kojem se svi mikroservisi promatraju i kontroliraju kao jedan logički entitet. Time se uspostavlja zajednički operativni kontekst, a **servisi se u pravilu pokreću i zaustavljaju istovremeno**. Takav pristup je prikladan za razvojna i testna okruženja, ali **nije optimalan za veće produkcijske sustave**, budući da se cijelokupan sustav izvršava na jednom računalu - čime se stvarni **koncept raspodijeljenog sustava ponovno zamagljuje**.

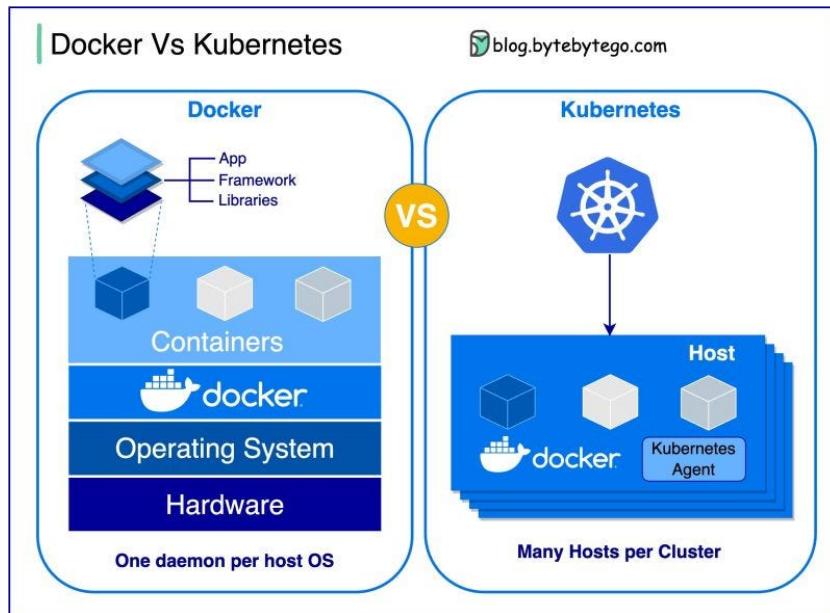


Ilustracija rada Docker Compose alata

Dakle, **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](#) (K8s) i [Docker Swarm](#). Ova složena rješenja omogućuju automatsko upravljanje kontejnerima, skaliranje, nadzor i druge napredne značajke.

Više o ovoj temi na sljedećim vježbama.



Ilustracija usporedbi 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 izraditi `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:

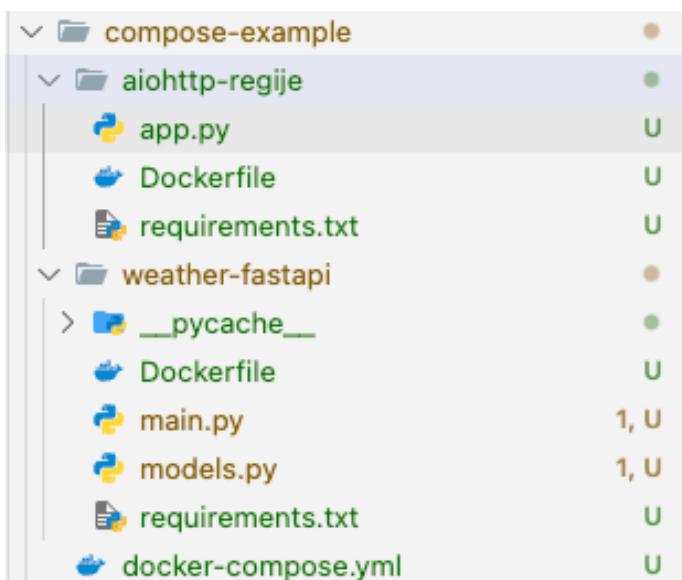
```
compose-example/
├── aiohttp-regije/
│   ├── app.py
│   ├── requirements.txt
│   └── Dockerfile
└── docker-compose.yml
```

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 (korisno za lakšu orientaciju u većim projektima):

- [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 .
```

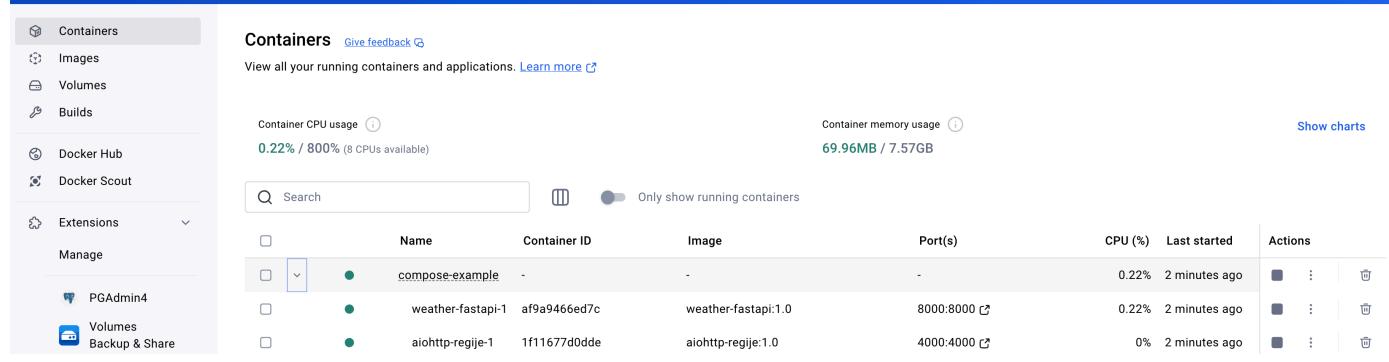
Nakon što smo izgradili oba predloška, možemo pokrenuti oba mikroservisa kao cjelinu pomoću Docker Compose alata. Navigirajte u `compose-example` direktorij i pokrenite sljedeću naredbu:

```
→ 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:

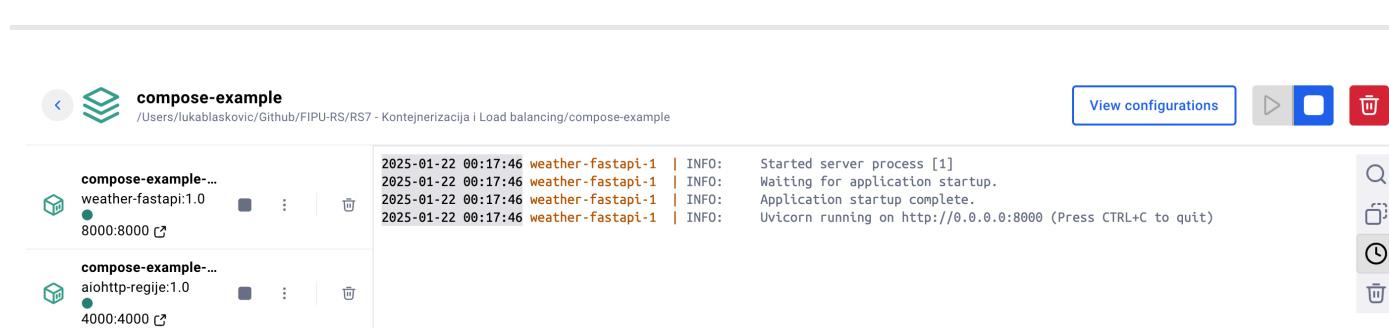
```
→ docker login
```

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



The screenshot shows the Docker Desktop interface. On the left, there's a sidebar with options like Containers, Images, Volumes, Builds, Docker Hub, Docker Scout, Extensions, Manage, PGAdmin4, Volumes, Backup & Share. The main area is titled 'Containers' with a 'Give feedback' link. It displays usage statistics: Container CPU usage (0.22% / 800%) and Container memory usage (69.96MB / 7.57GB). A search bar and a filter button ('Only show running containers') are present. Below is a table listing the three containers:

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

The screenshot shows the 'compose-example' project details. It lists two services: 'weather-fastapi:1.0' (port 8000) and 'aiohttp-regije:1.0' (port 4000). The log output for the weather-fastapi service shows the following startup messages:

```
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 posrednika** (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
```

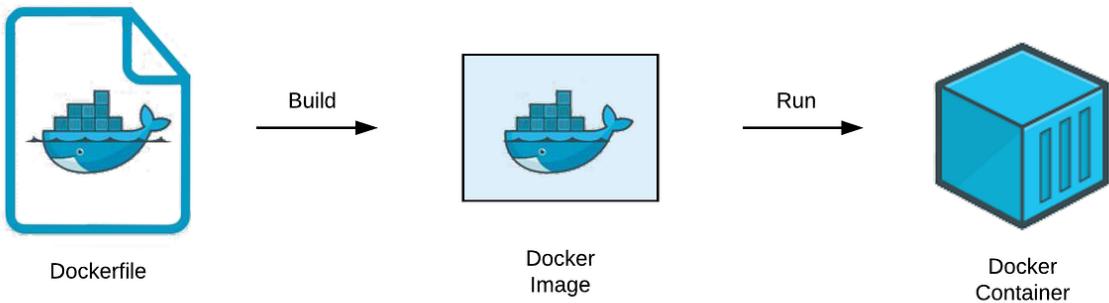
Obzirom da smo izmijenili kod, moramo ponovno izgraditi predložak:

```
→ 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, premještamo ju 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		
71a1a86cccd89	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

Napravite novi direktorij `social-network` i unutar njega kopirajte mikroservise izrađene u **Zadacima za vježbu 1.8:** `authAPI` i `socialAPI`.

Definirajte `docker-compose.yml` datoteku koja će pokrenuti oba mikroservisa kao cjelinu. Mikroservisi trebaju biti povezani na istoj mreži i svaki raditi na svom portu.

Jednom kad ste pokrenuli mikroservise zajedno koristeći Docker Compose i to uredno radi, napravite sljedeće izmjene:

- u mikroservisu `socialAPI` izmjenite rutu `GET /korisnici/{korisnik}/objave` na način da se očekuje **tijelo HTTP zahtjeva** s korisničkim imenom i lozinkom, isto validirajte koristeći novi Pydantic model.
- prije nego ruta `GET /korisnici/{korisnik}/objave` vrati podatke, mikroservis `socialAPI` treba poslati HTTP zahtjev na `authAPI` mikroservis ( ruta `/login`) kako bi provjerio korisničke podatke.
- implementirajte *dummy* autorizaciju u `authAPI` mikroservisu, tako da vraća `True` ako su korisničko ime i lozinka ispravni, inače vraća `False`.

Dakle, mikroservis `socialAPI` treba poslati HTTP zahtjev na `authAPI` mikroservis kako bi provjerio korisničke podatke prije nego što vrati podatke o objavama korisnika. Ako korisničko ime i lozinka nisu ispravni, `socialAPI` mikroservis treba vratiti grešku.

Nakon toga pokrenite oba mikroservisa zajedno koristeći Docker Compose i provjerite radi li nova funkcionalnost. **Napomena:** morate implementirati internu komunikaciju između 2 kontejnera, kao što je opisano u **poglavlju 2.2**.