بسم الله الرحمن الرحیم

پروژه سوم درس مهارت های پیشرفته کار با کامپیوتر

دکتر دوستی

سید علی تهامی - ۸۱۰۱۰۱۳۹۷

مهدی وجهی - ۸۱۰۱۰۱۵۵۸

فهرست

داکر 5
تولد بیبی 5
مقدمه 5
معرفی سرویس ها 5
تحلیل فایل docker-compose.yml 6
سرویس شهین: 8
بازی شاهین 9
سرویس Vote 10
سرویس Result 11
سرویس Worker 13
دیتابیس PostGreSQL 13
سرویس Redis 14
سرویس Nginx 14
شبکه 15
داکرفایل سرویس شهین 15
FROM golang:alpine AS builder 15
WORKDIR /app 16
COPY shahin.go 16
ENV GO111MODULE=off 16
RUN go build -o main 16
FROM alpine 16
COPY --from=builder /app/main 16
EXPOSE 80 16
ENTRYPOINT ["./main"] 17
داکرفایل سرویس شاهین 17
FROM nginx:alpine 17
COPY . /usr/share/nginx/html 17
فایل کانفیگ 18
events {} 18
http { ... } 19
server_name boy.shahin.ir → proxy_pass http://shahin_go:80; 19
server_name girl.shahin.ir → shahin_game 19

server_name vote.shahin.ir → vote 19
server_name result.shahin.ir 19
Swarm 20
ساخت خلاصه باکس Vagrant سفارشی Docker 20
۱. اسکریپت install.sh (اسکریپت Bootstrap) 20
۲. Vagrantfile برای ساخت Box سفارشی 20
۳. فرآیند ساخت و اضافه کردن Box 20
۴. نتیجه 21
ایجاد کلاستر داکر swarm 21
اجرای سرویس روی کلاستر 22
۱. اسکریپت deploy.sh 22
۲. فایل stack.yml (تعریف برنامه) 22
۳. یکپارچگی با Vagrantfile 23
۴. نتیجه Deployment 23
Ingress Routing Mesh 23
سوالات تشریحی 24
Compose watch 24
Docker bake 25
Docker wasm 25
داکر فایل 26
کوبرنتیز 27
تعریف کلاستر 27
تعداد گره ها در کلاستر 27
بروزرسانی 27
RC 27
روش های بروزرسانی 28
Deployment 29
Rolling update 29
Recreate 30
مقیاس پذیری خودکار 31
مانیفست 31
آزمایش 32
سوالات تشریحی 33
حافظه در کوبرنتیز 33

CNI 34
Affinity 34
KEDA 35
بررسی پورت ها 36
پیاده سازی 36
فیلتر کردن بسته های نامناسب 36
بررسی SYN, SYN-ACK در بسته ها 36
پیدا کردن آیپی های مشکوک 36
نمایش خروجی 37
تابع main 37
اجرا 37
بررسی فایل جهت شناخت مهاجم 38

داکر

https://chatgpt.com/share/684432a0-77c8-800f-9c5c-6f8b267fe75e

تولد بیبی

مقدمه

این بخش از پروژه ECS-CA3 به طراحی و پیاده‌سازی یک سیستم رأی‌گیری دوگزینه‌ای با استفاده از فناوری Docker اختصاص دارد. هدف، ایجاد یک اپلیکیشن توزیع‌شده است که از طریق Docker Compose سرویس‌های مختلف را هماهنگ کرده و در یک شبکه یکپارچه اجرا می‌کند. این سیستم امکان ثبت رأی، ذخیره‌سازی موقت، پردازش و نمایش نتایج را فراهم می‌کند.

معرفی سرویس ها

سیستم شامل چندین سرویس است که هر کدام نقش مشخصی در فرآیند رأی‌گیری دارند

سرویس db : این سرویس از دیتابیس **PostgreSQL** برای ذخیره‌سازی دائمی داده‌های مربوط به رأی‌گیری استفاده می‌کند. PostgreSQL به‌عنوان یک پایگاه داده رابطه‌ای قدرتمند و پایدار، تمامی رأی‌های پردازش‌شده را در جداول مشخصی ذخیره کرده و امکان اجرای کوئری‌های پیچیده برای تحلیل داده‌ها را فراهم می‌آورد. این سرویس از طریق متغیرهای محیطی تعریف‌شده در فایل .env پیکربندی شده و در شبکه داخلی پروژه با سایر سرویس‌ها ارتباط برقرار می‌کند.

سرویس redis : سرویس **Redis** به‌عنوان یک پایگاه داده سبک و سریع، برای ذخیره‌سازی موقت رأی‌ها در قالب یک صف عمل می‌کند. این سرویس با بهره‌گیری از ساختار داده‌ای ساده و عملکرد بالا، رأی‌های ارسالی از سرویس Vote را به‌صورت بلادرنگ دریافت و نگهداری می‌کند تا سرویس Worker آن‌ها را برای ذخیره دائمی به دیتابیس منتقل کند. Redis به دلیل سرعت بالا و مصرف منابع کم، انتخابی ایده‌آل برای این نقش است.

سرویس vote : این سرویس رابط کاربری **فرانت‌اند** پروژه را تشکیل می‌دهد و با استفاده از فریم‌ورک **Flask** پیاده‌سازی شده است. کاربران از طریق این سرویس می‌توانند رأی خود را از میان گزینه‌های موجود انتخاب کرده و ثبت کنند. رأی‌ها از طریق فرم‌های HTML جمع‌آوری شده و مستقیماً به سرویس Redis ارسال می‌شوند. طراحی ساده و کاربرپسند این سرویس، تجربه‌ای روان برای رأی‌دهندگان فراهم می‌کند.

سرویس: worker سرویس **Worker** که با زبان **Go** نوشته شده، نقش واسطه‌ای بین Redis و دیتابیس PostgreSQL را ایفا می‌کند. این سرویس رأی‌های ذخیره‌شده در صف Redis را به‌صورت دوره‌ای خوانده و پس از پردازش، آن‌ها را به دیتابیس منتقل می‌کند. استفاده از Go به دلیل عملکرد بالا و مدیریت کارآمد منابع، این سرویس را به گزینه‌ای مناسب برای پردازش‌های پس‌زمینه تبدیل کرده است.

سرویس result : این سرویس با استفاده از **Node.js** پیاده‌سازی شده و وظیفه نمایش نتایج رأی‌گیری را به‌صورت بلادرنگ بر عهده دارد. با اتصال به دیتابیس PostgreSQL، تعداد رأی‌های ثبت‌شده برای هر گزینه را محاسبه کرده و از طریق **Socket.IO** به کاربران نمایش می‌دهد. این سرویس با ارائه به‌روزرسانی‌های زنده، امکان مشاهده تغییرات نتایج را در لحظه فراهم می‌کند.

سرویس shahin_go : سرویس **shahin_go** که با زبان **Go** توسعه یافته، برای نمایش پیام‌های سفارشی در سیستم طراحی شده است. این سرویس می‌تواند پیام‌های اطلاع‌رسانی یا اعلان‌های خاص را بر اساس نیاز پروژه به کاربران نمایش دهد. انعطاف‌پذیری و سرعت بالای Go، این سرویس را به ابزاری کارآمد برای مدیریت پیام‌ها تبدیل کرده است.

سرویس shahin_game : این سرویس یک بازی **2048** را به‌عنوان یک قابلیت جانبی در پروژه ارائه می‌دهد. با استفاده از **Node.js** پیاده‌سازی شده و به کاربران امکان می‌دهد در کنار فرآیند رأی‌گیری، از یک تجربه سرگرم‌کننده نیز لذت ببرند. این سرویس به‌صورت مستقل عمل کرده و از طریق Nginx به کاربران ارائه می‌شود.

سرویس nginx : سرویس **Nginx** به‌عنوان یک **ریورس پروکسی** عمل کرده و وظیفه مدیریت درخواست‌های ورودی و مسیریابی آن‌ها به سرویس‌های مختلف را بر عهده دارد. با پیکربندی دامنه‌هایی مانند vote.shahin.ir و result.shahin.ir، این سرویس اطمینان می‌دهد که هر درخواست به سرویس مربوطه هدایت شود. Nginx با عملکرد بالا و قابلیت اطمینان، ترافیک سیستم را به‌خوبی مدیریت می‌کند.

تحلیل فایل docker-compose.yml

فایل docker-compose.yml هسته اصلی هماهنگی سرویس‌ها در این پروژه است.

کد docker به صورت زیر است:

```
services:
  #  Go ()
  shahin_go:
    build: ./shahin-go
    container_name: shahin_go
    networks:
      - front
  # 2048
```

```
shahin_game:
  build: ./2048
  container_name: shahin_game
  networks:
    - front

# (vote)
vote:
  build: ./voting-app/vote
  container_name: vote
  networks:
    - front
    - back
  depends_on:
    - redis

result:
  build: ./voting-app/result
  container_name: result
  environment:
    DB_USER: ${DB_USER}
    DB_PASS: ${DB_PASS}
  networks:
    - front
    - back
  depends_on:
    - db

worker:
  build: ./voting-app/worker
  container_name: worker
  networks:
    - back
  depends_on:
    - redis
    - db

#  PostgreSQL
db:
  image: postgres:15-alpine
  container_name: postgres
  restart: always
  environment:
    POSTGRES_USER: ${DB_USER}
    POSTGRES_PASSWORD: ${DB_PASS}
  networks:
    - back


#  Redis Queue
redis:
  image: redis:alpine
  container_name: redis
  networks:
```

```
      - back

  # Reverse Proxy of NGINX
  nginx:
    image: nginx:alpine
    container_name: nginx_proxy
    ports:
      - "80:80"
    volumes:
      - ./nginx/nginx.conf:/etc/nginx/nginx.conf
      - ./nginx/auth/.htpasswd:/etc/nginx/.htpasswd
    depends_on:
      - shahin_go
      - shahin_game
      - vote
      - result
    networks:
      - front

networks:
  front:
  back:
```

توضیحات بخش بخش هر یک از کد ها:

سرویس شهین:

```
services:
  #  Go ()
  shahin_go:
    build: ./shahin-go
    container_name: shahin_go
    networks:
      - front
```

- `build: ./shahin-go`: مسیر ساخت image از فایل `Dockerfile` برنامه Go.
- `container_name`: برای راحتی در شناسایی کانتینر در لاگ‌ها و دستورات مدیریتی.
- `networks`: این سرویس در شبکه‌ی `front` قرار دارد تا از طریق NGINX قابل دسترسی باشد.

کاربرد کلی: نمایش پیام تولد در دامنه boy.shahin.ir.

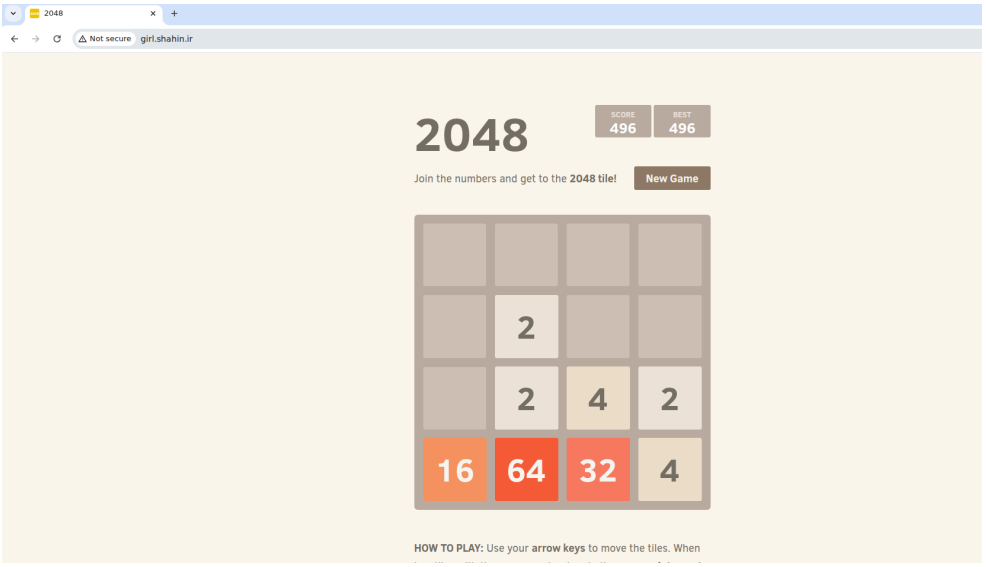
نتیجه:

بازی شاهین

```yaml
# 2048
shahin_game:
  build: ./2048
  container_name: shahin_game
  networks:
    - front
```

- ایمیج با استفاده از Dockerfile مخصوص بازی 2048 ساخته می‌شود.
- در شبکه‌ی `front` قرار دارد.

کاربرد کلی: سرویس‌دهی به بازی 2048 از طریق girl.shahin.ir. نتیجه :



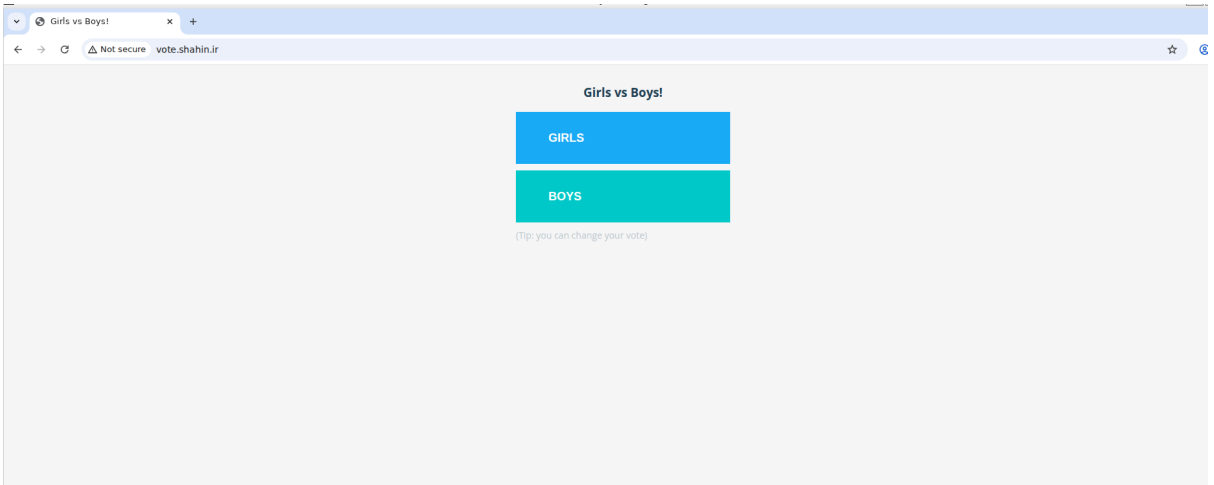
سرویس Vote

```yaml
# (vote)
vote:
  build: ./voting-app/vote
  container_name: vote
  networks:
    - front
    - back
  depends_on:
    - redis
```

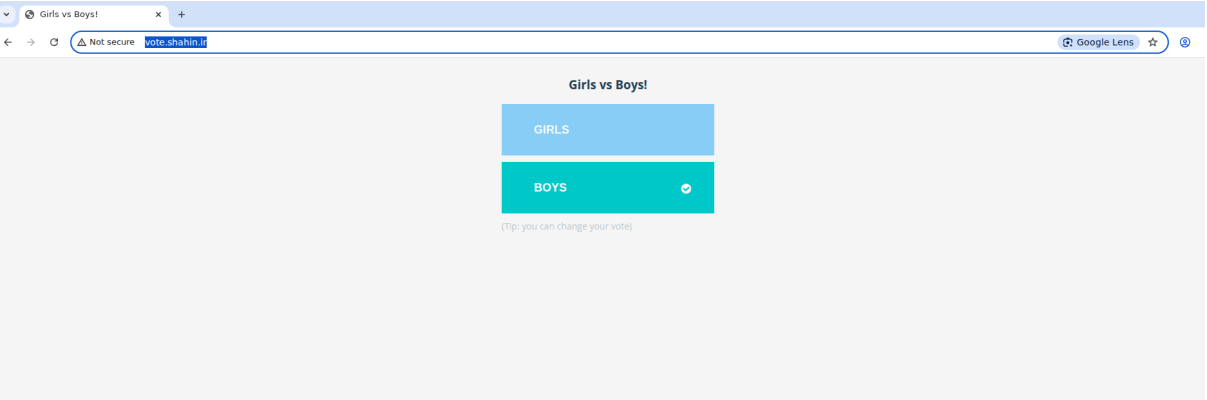
- فرانت‌اند رأی‌گیری با Flask.
- در دو شبکه `front` (برای ارتباط با NGINX) و `back` (برای ارتباط با Redis).
- `depends_on`: تضمین می‌کند که Redis قبل از vote اجرا شود.

کاربرد کلی: ثبت رأی کاربران برای boy/girl و ارسال آن به صف Redis.

نتیجه :



پس از ثبت رای به نفع یک نفر



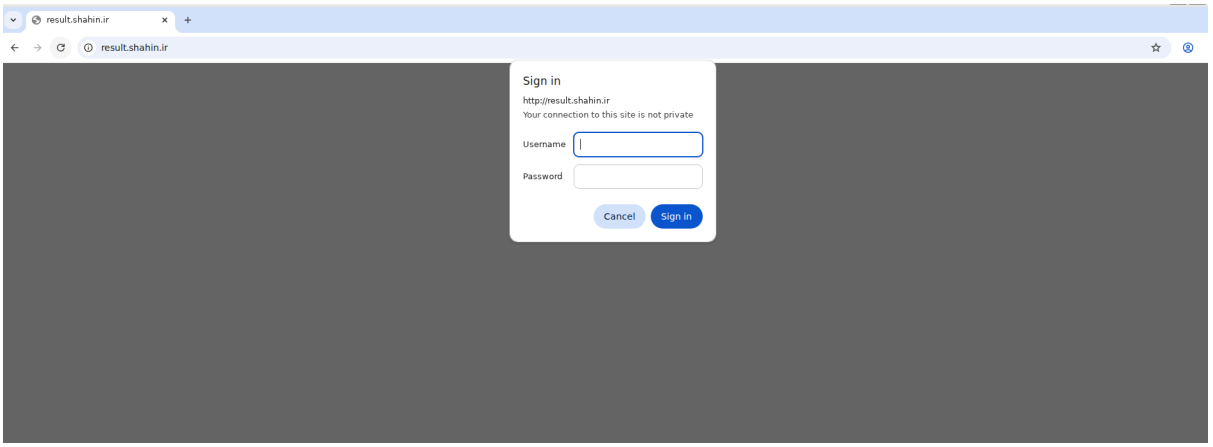
سرویس Result

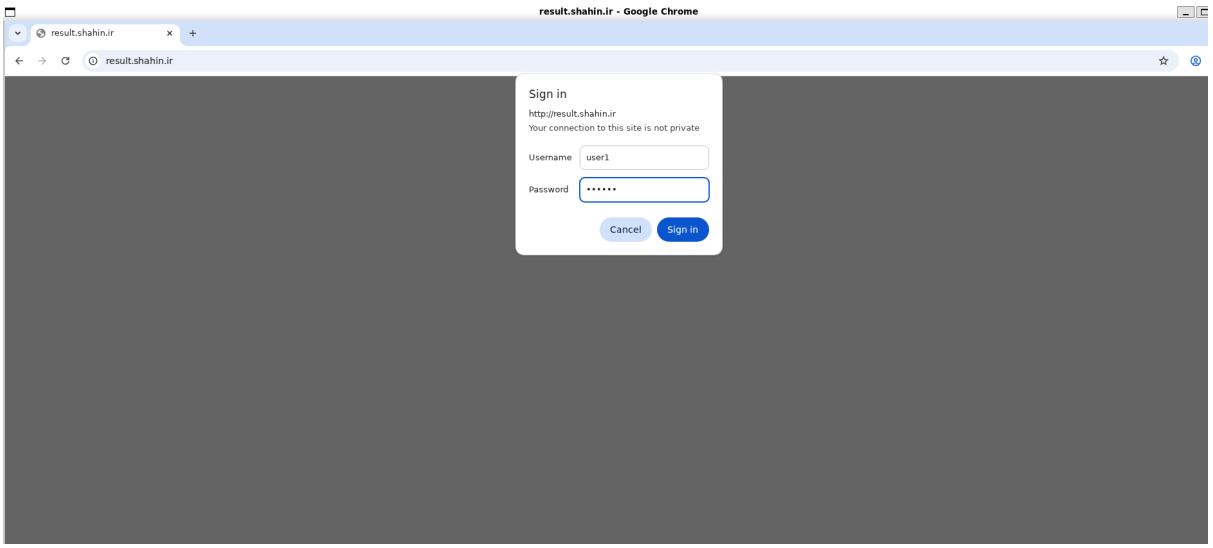
```yaml
result:
  build: ./voting-app/result
  container_name: result
  environment:
    DB_USER: ${DB_USER}
    DB_PASS: ${DB_PASS}
  networks:
    - front
    - back
  depends_on:
    - db
```

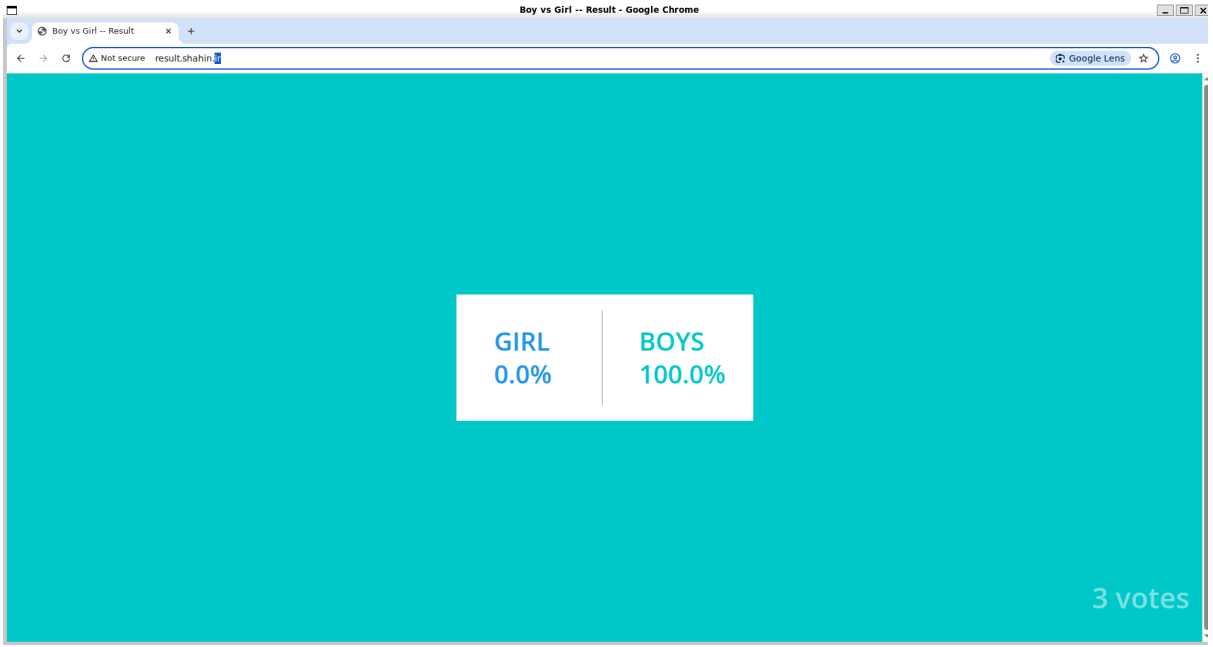
- اپلیکیشن Node.js برای نمایش زنده رأی‌ها.
- متصل به دیتابیس PostgreSQL با استفاده از env vars.
- شبکه دوگانه (برای اتصال به DB و NGINX).
- اجرای فقط بعد از بالا آمدن DB.

کاربرد کلی: نمایش لحظه‌ای نتایج رأی‌گیری از طریق Socket.io

نتیجه :


result.shahin.ir
result.shahin.ir
Sign in
http://result.shahin.ir
Your connection to this site is not private
Username
Password
Cancel
Sign in


result.shahin.ir - Google Chrome
result.shahin.ir
result.shahin.ir
Sign in
http://result.shahin.ir
Your connection to this site is not private
Username
user1
Password
Cancel
Sign in


Boy vs Girl -- Result - Google Chrome
Boy vs Girl -- Result
Not secure
result.shahin.ir
Google Lens
GIRL
0.0%
BOYS
100.0%
3 votes

سرویس Worker

```
worker:
  build: ./voting-app/worker
  container_name: worker
  networks:
    - back
  depends_on:
    - redis
    - db
```

- بدون رابط کاربری، فقط وظیفه‌ی پردازش رأی‌ها.
- دسترسی به Redis و DB.
- اول Redis و DB باید بالا باشند.

کاربرد کلی: دریافت رأی‌ها از Redis و ذخیره در PostgreSQL.

دیتابیس PostGreSQL

```
#  PostgreSQL
db:
  image: postgres:15-alpine
  container_name: postgres
  restart: always
  environment:
    POSTGRES_USER: ${DB_USER}
    POSTGRES_PASSWORD: ${DB_PASS}
  networks:
    - back
```

- پایگاه داده PostgreSQL برای ذخیره‌ی نتایج رأی‌گیری.
- استفاده از متغیرهای `env.` برای تعیین یوزر و پسورد.
- `restart: always`: در صورت کرش دوباره بالا بیاد.

- فقط در شبکه‌ی back.

کاربرد کلی: منبع اصلی نگهداری داده‌های رأی‌ها برای سرویس result.

سرویس Redis

```
#  Redis Queue
redis:
  image: redis:alpine
  container_name: redis
  networks:
    - back
```

- استفاده از Redis سبک برای صف پیام‌ها (MQ).
- نقش موقت در ذخیره‌سازی رأی‌ها پیش از ورود به DB.

کاربرد کلی: سریع‌ترین راه ذخیره‌سازی موقت رأی‌ها.

سرویس Nginx

```
# Reverse Proxy of NGINX
nginx:
  image: nginx:alpine
  container_name: nginx_proxy
  ports:
    - "80:80"
  volumes:
    - ./nginx/nginx.conf:/etc/nginx/nginx.conf
    - ./nginx/auth/.htpasswd:/etc/nginx/.htpasswd
  depends_on:
    - shahin_go
    - shahin_game
    - vote
    - result
  networks:
    - front
```

- سرویس مرکزی reverse proxy برای همه دامنه‌ها.
- استفاده از فایل‌های کانفیگ و احراز هویت محلی.
- روی پورت 80 اجرا می‌شود.
- وابسته به تمام سرویس‌هایی است که نیاز به مسیریابی دارند.

کاربرد کلی: نگهبان ورودی کل سیستم. درخواست‌ها را بر اساس دامنه به سرویس مناسب هدایت می‌کند.

شبکه

```
networks:
  front:
  back:
```

- `front`: شبکه‌ی خارجی برای ارتباط با کاربران و NGINX
- `back`: شبکه‌ی داخلی برای ارتباط بین سرویس‌ها مثل Worker، DB، Redis و Vote/Result

کاربرد کلی: جداسازی concerns برای امنیت، نظم و مقیاس‌پذیری بهتر.

داکرفایل سرویس شهین

```
FROM golang:alpine AS builder
WORKDIR /app
COPY shahin.go .
ENV GO111MODULE=off
RUN go build -o main .

FROM alpine
WORKDIR /app
COPY --from=builder /app/main .
EXPOSE 80
ENTRYPOINT ["./main"]
```

FROM golang:alpine AS builder

ایمیج پایه برای مرحله Build از نوع چندمرحله‌ای (multi-stage build).
استفاده از نسخه سبک `alpine` برای کاهش حجم.

کاربرد: ساخت باینری Go در محیط جدا، بدون انتقال ابزارهای توسعه به ایمیج نهایی.

```
WORKDIR /app
```

مکان اجرای دستورات بعدی داخل کانتینر.

```
COPY shahin.go .
```

کپی کد برنامه Go به کانتینر.

```
ENV GO111MODULE=off
```

غیرفعال کردن ماژول‌های Go (در صورتی که پروژه ساده بدون go.mod باشد).

```
RUN go build -o main .
```

کامپایل فایل `shahin.go` و تولید فایل اجرایی `main`.

```
FROM alpine
```

شروع مرحله دوم. استفاده از ایمیج سبک برای اجرای باینری بدون ابزار توسعه.

```
COPY --from=builder /app/main .
```

کپی فقط فایل اجرایی به ایمیج نهایی از مرحله Build.

```
EXPOSE 80
```

مشخص کردن پورت ۸۰ برای دسترسی HTTP.

```
ENTRYPOINT ["./main"]
```

نقطه اجرای برنامه Go در زمان اجرای کانتینر.

داکرفایل سرویس شاهین

در این مرحله ابتدا نیاز است تا بازی 2048 را از ریپو گیتهاب کلون کنیم. پس از انجام این مرحله، فایل داکرفایل را در درون دایرکتوری پیاده سازی میکنیم

```
FROM nginx:alpine
COPY . /usr/share/nginx/html
```

```
FROM nginx:alpine
```

استفاده از وب‌سرور NGINX به‌صورت سبک (Alpine) برای سرو فایل‌های HTML/CSS/JS.

```
COPY . /usr/share/nginx/html
```

کپی تمام فایل‌های پروژه (2048) به مسیر ریشه‌ی سرو NGINX.

کاربرد: نمایش بازی 2048 به‌صورت Static Website.

فایل کانفیگ

```
events {}

http {
    server {
        listen 80;
        server_name boy.shahin.ir;

        location / {
            proxy_pass http://shahin_go:80;
        }
    }

    server {
        listen 80;
        server_name girl.shahin.ir;

        location / {
            proxy_pass http://shahin_game:80;
        }
    }

    server {
        listen 80;
        server_name vote.shahin.ir;

        location / {
            proxy_pass http://vote:80;
        }
    }

    server {
        listen 80;
        server_name result.shahin.ir;

        auth_basic "Restricted";
        auth_basic_user_file /etc/nginx/.htpasswd;

        location / {
            proxy_pass http://result:80;
        }
    }
}
```

```
events {}
```

بخش الزامی ولی ساده برای تنظیمات پایه NGINX.

`http { ... }`

همه سرورها و ruleها درون این بلاک تعریف می‌شوند.

`server_name boy.shahin.ir` → `proxy_pass http://shahin_go:80;`

درخواست‌ها به این دامنه به کانتینر `shahin_go` هدایت می‌شوند.

`server_name girl.shahin.ir` → `shahin_game`

درخواست‌های مربوط به بازی 2048.

`server_name vote.shahin.ir` → `vote`

فرم رأی‌گیری Flask.

`server_name result.shahin.ir`

نمایش نتیجه رأی‌گیری. همراه با احراز هویت:

- `auth_basic`: فعال‌سازی پنجره ورود با عنوان دلخواه
- `auth_basic_user_file`: فایل رمز عبور htpasswd

کاربرد: فقط کاربران مجاز بتوانند نتیجه رأی‌گیری را ببینند.

Swarm

https://g.co/gemini/share/4a742da483ef

ساخت خلاصه باکس Vagrant سفارشی Docker

به طور خلاصه فرآیند ساخت یک باکس Vagrant سفارشی را بررسی می کنیم که شامل نصب و پیکربندی Docker برای استفاده بهینه در محیط‌های داخلی ایران، و آماده‌سازی سیستم عامل برای استقرار سریع ماشین‌های مجازی است.

۱. اسکریپت install.sh (اسکریپت Bootstrap)

این اسکریپت وظایف کلیدی زیر را برای آماده‌سازی ماشین مجازی پایه انجام می‌دهد:

- **بهینه‌سازی مخازن:** آدرس مخازن پیش‌فرض اوبونتو را از سرورهای `.us` به `.ir` جهت افزایش سرعت تغییر می‌دهد و لیست بسته‌ها را به‌روزرسانی می‌کند.
- **نصب پیش‌نیازها:** ابزارهای ضروری مانند `gnupg`، `curl`، `ca-certificates` و `bash-completion` را نصب می‌کند.
- **پیکربندی DNS:** سرورهای DNS شکن را برای رفع تحریم در داخل VM تنظیم می‌کند.
- **نصب Docker Engine:** مخازن رسمی داکر را اضافه کرده و Docker Engine، CLI و پلاگین‌های آن را نصب می‌کند. کاربر `vagrant` نیز به گروه `docker` اضافه می‌شود.
- **پیکربندی `daemon.json` داکر:** فایل پیکربندی `daemon.json` را برای استفاده از `registry-mirrors` آروان‌کلاود (جهت رفع تحریم) و تنظیمات IP داخلی شبکه داکر، ایجاد یا به‌روزرسانی می‌کند. سپس سرویس داکر ریستارت می‌شود.
- **پاکسازی سیستم:** کش پکیج‌های APT و فایل‌های موقت را حذف می‌کند و فضای خالی دیسک را با صفر پر می‌کند تا حجم Box نهایی کاهش یابد.

۲. Vagrantfile برای ساخت Box سفارشی

برای ساخت این Box سفارشی، از یک `Vagrantfile` ساده استفاده می‌شود که ماشین مجازی پایه را تعریف کرده و اسکریپت `install.sh` را برای پیکربندی آن اجرا می‌کند.

۳. فرآیند ساخت و اضافه کردن Box

پس از آماده‌سازی اسکریپت `install.sh` و `Vagrantfile` در یک دایرکتوری، فرآیند ساخت Box با یک دستور واحد اجرا می‌شود:

```
vagrant up && \
```

```
vagrant halt && \
vagrant package --output docker-ubuntu-2204.box && \
vagrant box add docker-ubuntu-2204 docker-ubuntu-2204.box && \
vagrant destroy -f
```

این دستور به ترتیب: ماشین مجازی را راه‌اندازی کرده و اسکریپت `install.sh` را روی آن اجرا می‌کند، سپس ماشین را خاموش کرده، از آن یک Box به نام `docker-ubuntu-2204.box` می‌سازد، آن را به لیست Boxهای Vagrant اضافه می‌کند و در نهایت ماشین مجازی موقت را حذف می‌کند.

۴. نتیجه

با اتمام این فرآیند، یک Box Vagrant جدید با نام `docker-ubuntu-2204` در سیستم شما موجود است. این Box شامل اوبونتو 22.04 با Docker از پیش نصب و کاملاً پیکربندی شده (از جمله تنظیمات `daemon.json` برای آینه‌های داخلی و DNS بهینه) است. این باکس امکان راه‌اندازی سریع و آفلاین چندین ماشین مجازی را برای ساخت کلاسترهای Docker Swarm فراهم می‌کند.

ایجاد کلاستر داکر swarm

فرآیند راه‌اندازی یک کلاستر Docker Swarm چند-Master و Worker را با استفاده از Vagrant و یک باکس سفارشی (شامل Docker از پیش نصب‌شده) به صورت مختصر شرح می‌دهد.

`Vagrantfile` وظیفه تعریف، پیکربندی و Orchestration نودهای Swarm را بر عهده دارد.

- **متغیرهای کلاستر:**
  - `IMAGE`: نام باکس سفارشی ساخته شده قبلی (شامل Docker).
  - `MASTER_COUNT` و `WORKER_COUNT`: تعداد نودهای Master و Worker.
  - `all_hosts`: لیست IP و Hostname تمام نودها برای تنظیم `etc/hosts`.
- **تعریف نودهای Master:** هر Master (مانند `master1`) با Hostname و IP ثابت تعریف می‌شود.
- **راه‌اندازی و پیوستن به Swarm (برای Masterها):**
  - **Master اولیه (Leader):** فقط `master1` وظیفه `docker swarm init` را بر عهده می‌گیرد. توکن‌های `manager.token` و `worker.token` را تولید کرده و در پوشه `shared /vagrant` ذخیره می‌کند. همچنین اسکریپت `deploy.sh` را در پس‌زمینه اجرا می‌کند.
  - **سایر Masterها:** با استفاده از `manager.token` و IP Master اولیه، به کلاستر به عنوان Manager می‌پیوندند.
- **تعریف نودهای Worker:** هر Worker (مانند `worker1`) با Hostname و IP ثابت تعریف می‌شود.

- **پیوستن به Swarm (برای Workerها):** Workerها با استفاده از `worker.token` و IP Master اولیه، به کلاستر به عنوان Worker می‌پیوندند.

اجرای سرویس روی کلاستر

۱. اسکریپت deploy.sh

این اسکریپت Shell وظیفه خودکارسازی استقرار برنامه‌ها را بر عهده دارد و روی Master اولیه (معمولاً `master1`) اجرا می‌شود.

عملکرد کلیدی:

- **انتظار برای پیوستن نودها:** اسکریپت ابتدا صبر می‌کند تا تمام نودهای مورد انتظار (در این مثال، 4 نود شامل 3 Master و 1 Worker که در مجموع 5 خط در خروجی `docker node ls` را شامل می‌شوند) به کلاستر Swarm بپیوندند و وضعیت Ready پیدا کنند. این کار از طریق یک حلقه `while` و بررسی خروجی `docker node ls` انجام می‌شود.
- **استقرار Stack:** پس از اطمینان از آمادگی کلاستر، دستور `docker stack deploy -c /vagrant/stack.yml mystack` را اجرا می‌کند. این دستور، برنامه تعریف شده در فایل `stack.yml` را با نام `mystack` در سراسر کلاستر Swarm مستقر می‌کند.
- **ثبت لاگ:** تمامی مراحل و خروجی‌ها در فایل `deploy.log` در دایرکتوری `/home/vagrant/` ثبت می‌شوند.

۲. فایل stack.yml (تعریف برنامه)

این فایل، برنامه چند-کانتینری (Multi-service application) را برای استقرار در Docker Swarm تعریف می‌کند:

```
services:
  nginx:
    image: nginx:1.27.5-alpine
    ports:
      - "80:80"
    deploy:
      replicas: 5
      restart_policy:
        condition: on-failure
```

جزئیات کلیدی:

- **Service `nginx`:** یک سرویس وب با استفاده از ایمیج Nginx را تعریف می‌کند.

- **image:** ایمیج کانتینر مورد استفاده (در اینجا `nginx:1.27.5-alpine`).
- **ports:** نگاشت پورت‌ها از کانتینر به هاست. Docker Swarm از قابلیت "Routing Mesh" برای دسترسی به این پورت‌ها از هر نود در کلاستر پشتیبانی می‌کند.
- **deploy.replicas:** تعیین می‌کند که چند نمونه (Instance) از این سرویس باید در کلاستر Swarm اجرا شود (در اینجا 5 نمونه). Swarm به طور خودکار این نمونه‌ها را بین نودهای در دسترس توزیع می‌کند.
- **deploy.restart_policy:** رفتار سرویس را در صورت بروز مشکل تعیین می‌کند (در اینجا، کانتینر در صورت خرابی به طور خودکار ریستارت می‌شود).

۳. یکپارچگی با Vagrantfile

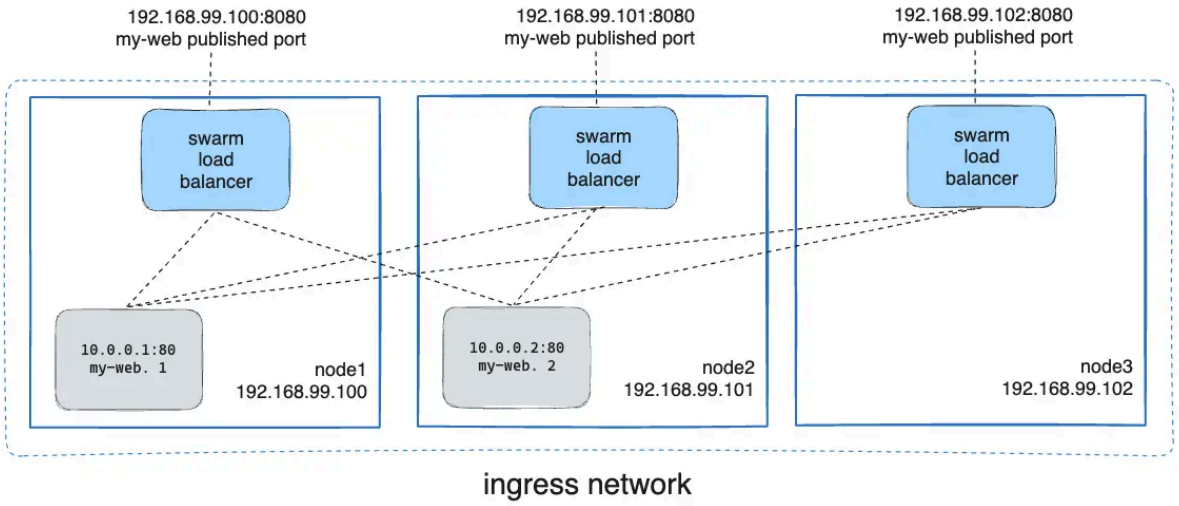
- **اجرای deploy.sh:** در `Vagrantfile` اصلی، خط `nohup bash /vagrant/deploy.sh &` اطمینان می‌دهد که اسکریپت `deploy.sh` به صورت خودکار و در پس‌زمینه روی Master اولیه اجرا می‌شود، پس از اینکه Master کلاستر را initialize کرد و توکن‌ها را تولید نمود.
- **دسترسی به stack.yml:** فایل `stack.yml` باید در همان دایرکتوری `Vagrantfile` قرار گیرد تا از طریق `/vagrant` Shared Folder در داخل VMها قابل دسترسی باشد.

۴. نتیجه Deployment

با این پیکربندی، کلاستر Docker Swarm به طور خودکار پس از آماده شدن تمامی نودها، برنامه Nginx را مستقر می‌کند. Docker Swarm مسئولیت مدیریت، توزیع و اطمینان از دسترس‌پذیری بالای 5 نمونه سرویس Nginx را بر عهده خواهد گرفت. این فرآیند یک راه‌حل کارآمد برای استقرار خودکار برنامه‌های کانتینری در محیط‌های توزیع‌شده فراهم می‌کند.

Ingress Routing Mesh

در داکر سوارم از Ingress Routing Mesh استفاده می کند که با دریافت یک درخواست روی هر کدام از گره ها آن را به یکی از کانتینر های موجود روی یکی از گره ها بر می گرداند و مسئولیت مسیریابی و لود بالانس بین کانتینر ها را دارد. ای موضوع را می توانید در تصویر زیر مشاهده کنید.

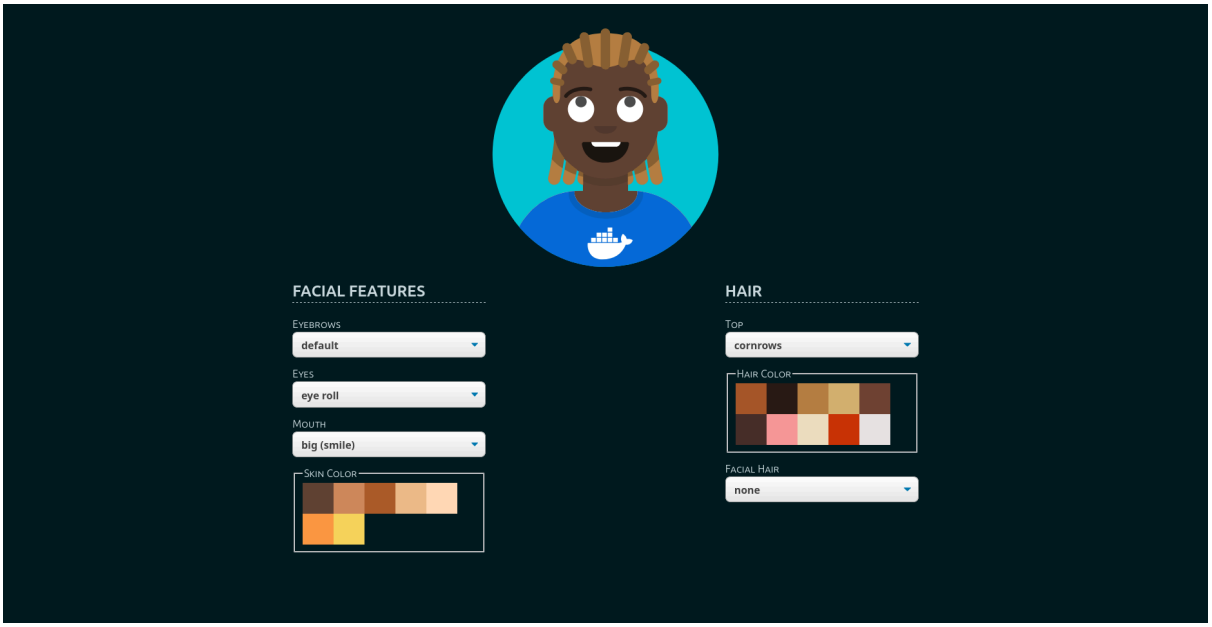


بنابراین می توان ip هر کدام از گره ها را در dns ثبت کرد.

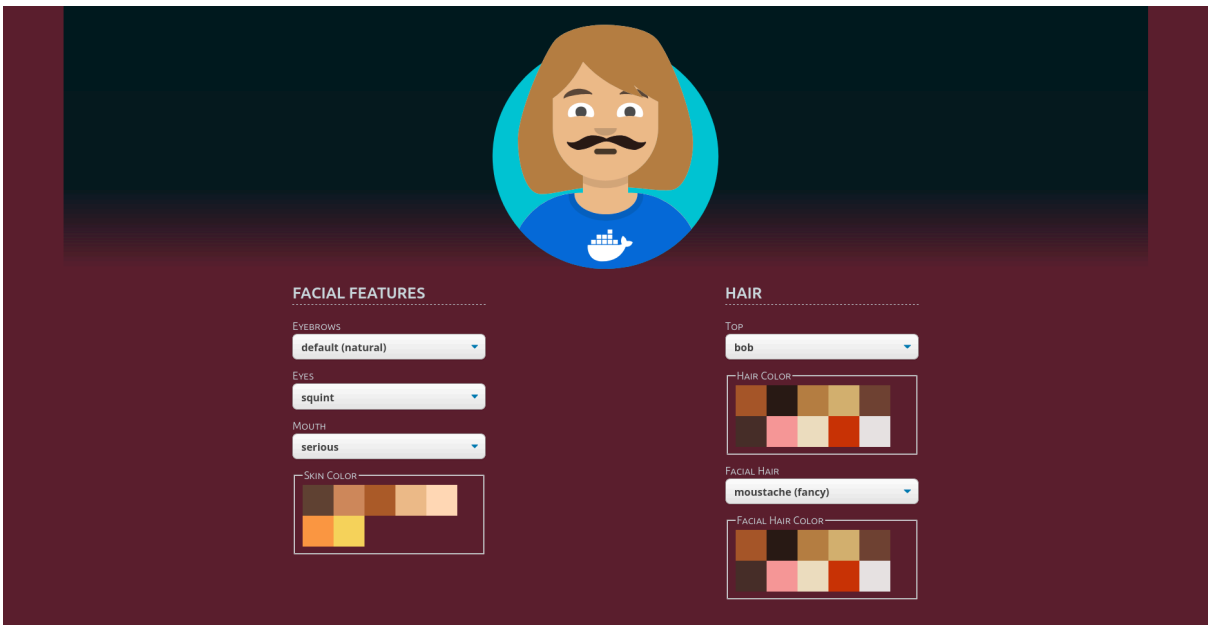
سوالات تشریحی

Compose watch

در زمان توسعه وقتی پروژه روی داکر است لازم است مدام آن را بسازیم و تست کنیم. مشکلی که هست این است که باید کل کانتینر را پایین بیاوریم و image را بسازیم و سپس مجدد آن را اجرا کنیم و این فرآیند دردسر زا و زمان بر است. در عوض می توانیم از compose watch استفاده کنیم. در این روش ما تعریف می کنیم که با هر تغییری که انجام می شود چه کاری باید انجام شود و دیگر این فرایند خودکار می شود و سرعت آن نیز افزایش می یابد. مثلا در پروژه قرار داده شده ابتدا سایت به شکل زیر است:



وقتی دستور docker compose watch را میزنیم (بعد از تنظیم در compose file) می توانیم تغییرات را به سرعت مشاهده کنیم. مثلا:



Docker bake

ابزاری است که به ما امکان می دهد تنظیمات ساخت image را ذخیره کنیم. نمونه ای از آن را می توانید در تصویر زیر مشاهده کنید.

```
target "myapp" {
  context = "."
  dockerfile = "Dockerfile"
  tags = ["myapp:latest"]
  args = {
      foo = "bar"
  }
  no-cache = true
  platforms = ["linux/amd64", "linux/arm64"]
}
```

همچنین امکان ساخت تصاویر به صورت موازی را نیز به ما می دهد. در نهایت می توان آن را مستقیم در کامپوز تعریف کرد که فرآیند استفاده ساده تر باشد.

Docker wasm

https://chatgpt.com/share/6841ddde-b948-8001-b578-2a9eb32ec022

WebAssembly یک فرمت باینری سبک و ایزوله است که امکان اجرای سریع و ایمن کدهای کامپایل‌شده از زبان‌هایی مانند Rust و C/C++ را در محیط‌هایی مثل مرورگر یا سرور فراهم می‌کند. ترکیب WASM با Docker، مخصوصاً از نسخه ۲۴ به بعد، این امکان را می‌دهد که برنامه‌های سبک و پرسرعت را بدون وابستگی به سیستم‌عامل اجرا کنیم. این ترکیب برای کاربردهایی مانند Edge Computing، اجرای ماژول‌های ایمن، و کاهش زمان راه‌اندازی بسیار مفید است. به‌عنوان مثال، می‌توان برنامه‌ای را با Rust نوشت، به

WASM کامپایل کرد و آن را در Docker با استفاده از یک WASM runtime مانند Wasmtime یا Wasmedge اجرا کرد.

داکر فایل

در فایل Dockerfile، دستورات مختلفی وجود دارند که هر کدام نقش خاصی در فرایند ساخت و اجرای کانتینر ایفا می‌کنند. دستور `RUN` برای اجرای دستورات در **مرحله ساخت (Build)** استفاده می‌شود، مثلاً نصب پکیج‌ها یا ایجاد فایل‌ها، و نتیجه آن در لایه نهایی ایمیج ذخیره می‌شود. در مقابل، دستور `CMD` در **مرحله اجرای کانتینر (Run)** استفاده می‌شود تا برنامه پیش‌فرض کانتینر را مشخص کند. `ENTRYPOINT` نیز مشابه `CMD` است اما معمولاً برای تعریف دستور اصلی و غیرقابل‌تغییر اجرای کانتینر استفاده می‌شود. اگر هر دو به‌صورت همزمان باشند، `CMD` نقش آرگومان را برای `ENTRYPOINT` ایفا می‌کند. همچنین، دستور `WORKDIR` برای تعیین مسیر کاری پیش‌فرض در داخل کانتینر استفاده می‌شود، که به‌جای استفاده از `cd` در هر `RUN`، یک مسیر ثابت برای تمام دستورات بعدی فراهم می‌کند.

از دیگر دستورات مهم می‌توان به `COPY` و `ADD` اشاره کرد. هر دو برای انتقال فایل از سیستم میزبان به داخل کانتینر در زمان build استفاده می‌شوند، اما `ADD` امکانات بیشتری دارد؛ مثلاً می‌تواند فایل‌های فشرده `tar.gz.` را به‌صورت خودکار استخراج کند. با این حال، به دلیل سادگی، وضوح عملکرد و امنیت بیشتر، در اکثر موارد استفاده از `COPY` پیشنهاد می‌شود، مگر در مواقعی که واقعاً به قابلیت استخراج فایل نیاز داریم. در مجموع، شناخت تفاوت این دستورات کمک می‌کند تا Dockerfile بهینه‌تری نوشته شود که هم سریع‌تر اجرا شود و هم خواناتر و قابل نگهداری‌تر باشد.

کوبرنتیز

تعریف کلاستر

برای تعریف اون یک کلاستر تعریف می کنیم که نسخه api و بعد هم کنترل کننده ها و کارگر ها رو همراه با نسخه image تعریف می کنیم.

```
kind: Cluster
apiVersion: kind.x-k8s.io/v1alpha4
nodes:
  - role: control-plane
      image: kindest/node:v1.33.1
  - role: control-plane
      image: kindest/node:v1.33.1
  - role: control-plane
      image: kindest/node:v1.33.1
  - role: worker
      image: kindest/node:v1.33.1
```

تعداد گره ها در کلاستر

برای این تعداد گره های کنترل فرد است که etcd در آن قرار دارد. در واقع تنها سرویس stateful در کلاستر برای کوبرنتیز پایگاه داده etcd هست و چون برای توزیع کردن این پایگاه داده از الگوریتم raft استفاده شده و برای وقتی که شبکه ۲ تیکه می شوند باید به اجماع برسند لازم است تعداد فرد باشد. که برای محصول معمولا مقادیر ۳، ۵ و ۷ را قرار می دهند و مقادیر بیشتر به دلیل دشواری در هماهنگی معمولا استفاده نمی شود. همچنین می توانیم با جدا کردن etcd باقی سرویس های مدیریتی را به تعداد زوج و با مقیاس بیشتر هم قرار داد.

بروزرسانی

RC

در ابتدا توقع داشتم که وقتی ۲ کنترلر را اعمال می کنیم تنها یک پاد ایجاد شود چون هر دو دنبال یک برچسب هستند وقتی یکی ایجاد می کنه دیگری نیز آن را پیدا می کند و روی همان نظارت می کند ولی ۲ پاد ایجاد شد. سپس یک پاد با همان برچسب ایجاد کردم و بعد کنترلر هارو اعمال کردم، اولی سراغ همان پاد قبلی رفت و دومی یکی ایجاد کرد. متوجه شدم که هر پاد یک بخش دارد که مشخص کرده توسط چه کسی کنترل می شود وقتی که آن بخش پر باشد دیگر کنترلر جدید آن را زیر مجموعه خود در نظر نمی گیرد.

```
Name:             nginx-pod
```

```
Namespace:        default
Priority:         0
Service Account:  default
Node:             kind-worker/172.30.2.2
Start Time:       Fri, 06 Jun 2025 10:28:00 +0330
Labels:           app=nginx
Annotations:      <none>
Status:           Running
IP:               10.244.1.5
IPs:
  IP:             10.244.1.5
Controlled By:  ReplicationController/nginx-rc-1
```

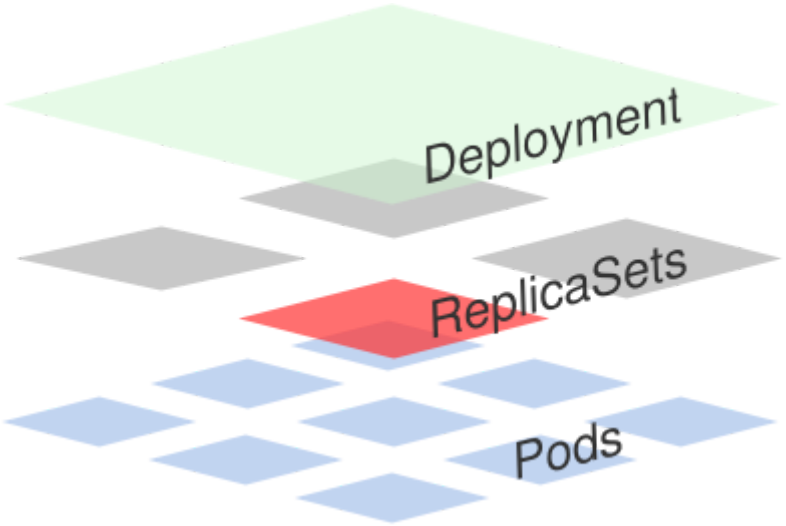
روش های بروزرسانی

برای بروزرسانی می توان از راه های زیر استفاده کرد:

Strategy	ZERO DOWNTIME	REAL TRAFFIC TESTING	TARGETED USERS	CLOUD COST	ROLLBACK DURATION	NEGATIVE IMPACT ON USER	COMPLEXITY OF SETUP
RECREATE version A is terminated then version B is rolled out	✗	✗	✗	■□□	■■■	■■■	□□□
RAMPED version B is slowly rolled out and replacing version A	✓	✗	✗	■□□	■■■	■□□	■□□
BLUE/GREEN version B is released alongside version A, then the traffic is switched to version B	✓	✗	✗	■■■	□□□	■■□	■■□
CANARY version B is released to a subset of users, then proceed to a full rollout	✓	✓	✗	■□□	■□□	■□□	■■□
A/B TESTING version B is released to a subset of users under specific condition	✓	✓	✓	■□□	■□□	■□□	■■■
SHADOW version B receives real world traffic alongside version A and doesn't impact the response	✓	✓	✗	■■■	□□□	□□□	■■■

مثلا برای حالت اول در مانیفست تعداد رپلیکا ها را صفر می کنیم و در مانیفست دیگر به همان تعداد از سرویس جدید بالا می آوریم. برای حالت دوم مثل حالت اول عمل می کنیم ولی یکی از این می آوریم بالا و یکی از قدیمی پایین می آوریم تا تمام شود. باقی حالت ها هم به طرز مشابه. حالت اول برای stateful ها نیز مناسب است.

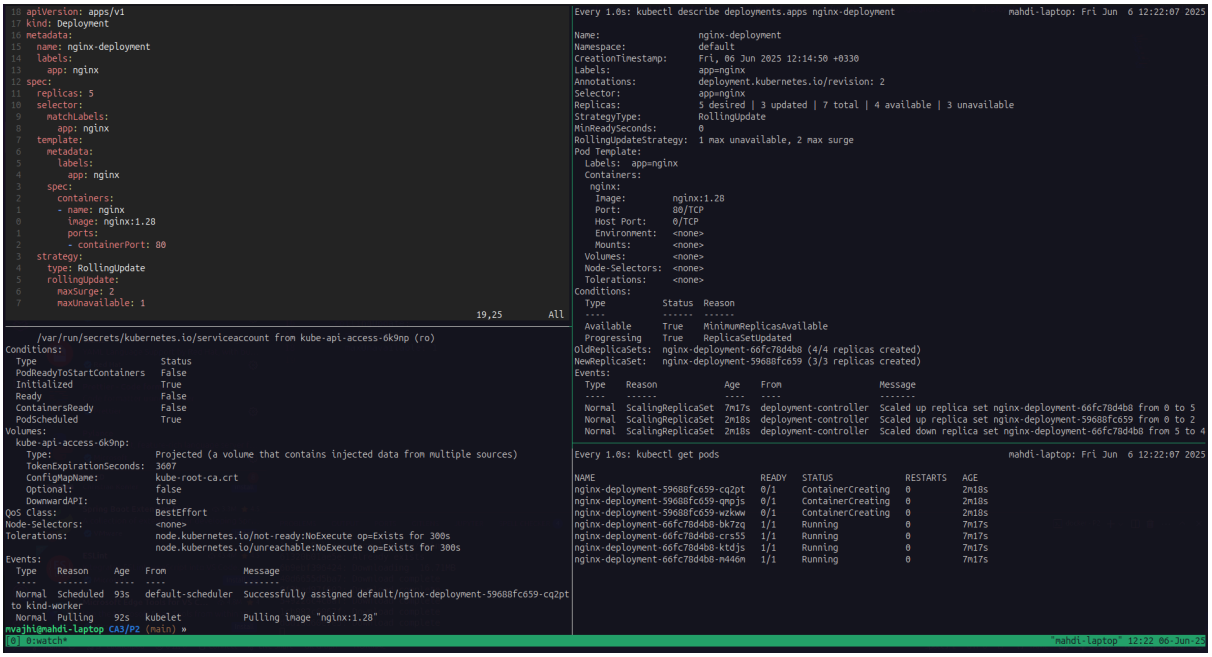
Deployment



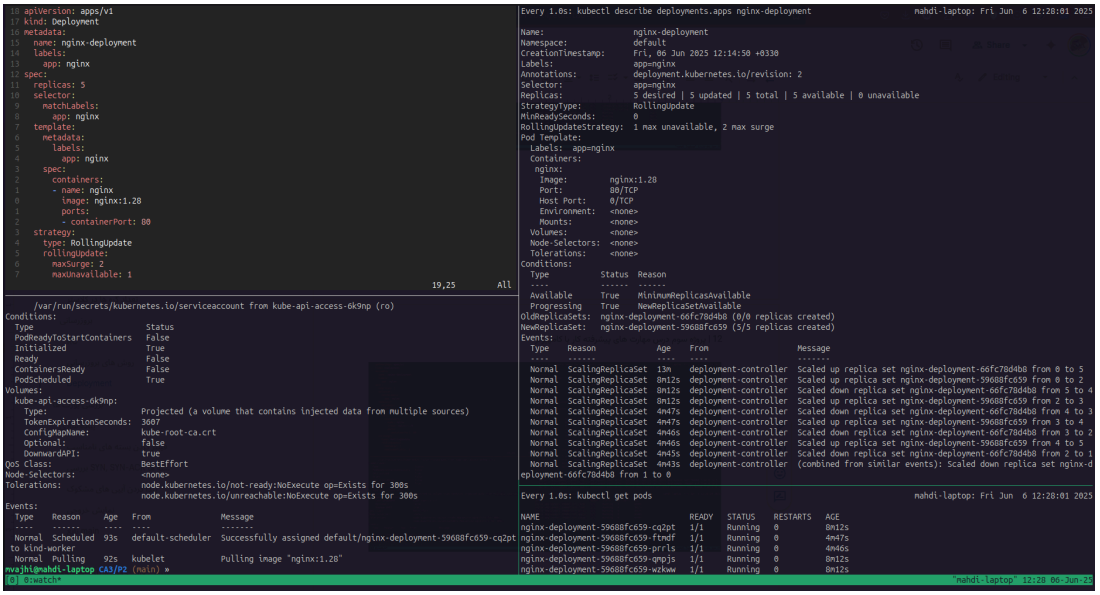
دیپلوی در واقع در لایه بالاتر از رپلیکا ست قرار می گیرد و همانطور که از نام آن مشخص است وظیفه استقرار سرویس ها و پاد ها را دارد به این صورت که رپلیکا ست ها را تعریف می کند و آنها را کنترل می کند. این موضوع روند بروزرسانی را بسیار آسان می کند زیرا آن فرآیند دستی ای که گفته شد در این ورکلود به صورت خودکار انجام می شود.

Rolling update

در این مثال از نسخه ۲۷ به ۲۸ می رویم همانطور که می بینید در تصویر اول طبق شرایطی که تعریف کردیم ۱ پاد قبلی را خاموش کرده و در حال لود کردن ۳ پاد جدید است.



سپس وقتی کار اون سه تا تموم کرد یکی یکی از اون کم می کنه به این میده.

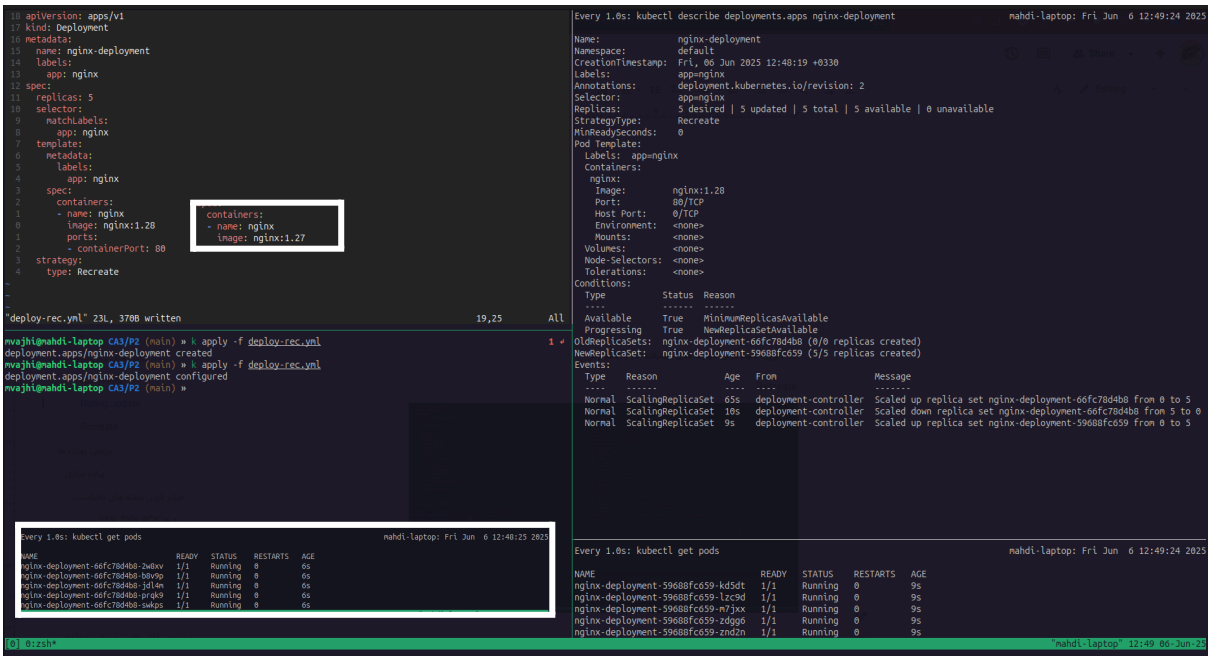


برای درک بهتر به لاگ های دیپلوی توجه کنید.

```
Scaled up replica set nginx-deployment-66fc78d4b8 from 0 to 5
Scaled up replica set nginx-deployment-59688fc659 from 0 to 2
Scaled down replica set nginx-deployment-66fc78d4b8 from 5 to 4
Scaled up replica set nginx-deployment-59688fc659 from 2 to 3
Scaled down replica set nginx-deployment-66fc78d4b8 from 4 to 3
Scaled up replica set nginx-deployment-59688fc659 from 3 to 4
Scaled down replica set nginx-deployment-66fc78d4b8 from 3 to 2
Scaled up replica set nginx-deployment-59688fc659 from 4 to 5
Scaled down replica set nginx-deployment-66fc78d4b8 from 2 to 1
(combined from similar events): Scaled down replica set nginx-d
```

Recreate

در این روش همانطور که در تصویر مشاهده می کنید ابتدا همه ی پاد ها از بین می روند و سپس پاد های جدید درست می شوند. مواردی که با کادر سفید هستند تفاوت ها قبل از دیپلوی نسخه جدید هستند.



در لاگ ها دقیقا می توانید روند بروزرسانی را مشاهده کنید.

```
Message
-------
Scaled up replica set nginx-deployment-66fc78d4b8 from 0 to 5
Scaled down replica set nginx-deployment-66fc78d4b8 from 5 to 0
Scaled up replica set nginx-deployment-59688fc659 from 0 to 5
```

مقیاس پذیری خودکار

مانیفست

ابتدا برای دیپلوی خود محدودیت می گذاریم. در نهایت برای مقایس پذیری تنها از محدودیت cpu استفاده می کنیم.

```
      resources:
            limits:
            cpu: "0.5"
            ephemeral-storage: 0.5G
            memory: 1G
            requests:
            cpu: "0.5"
            ephemeral-storage: 0.5G
            memory: 1G
```

سپس یک سرویس به عنوان لود بالانسر تعریف می کنیم که ترافیک را پخش کند.

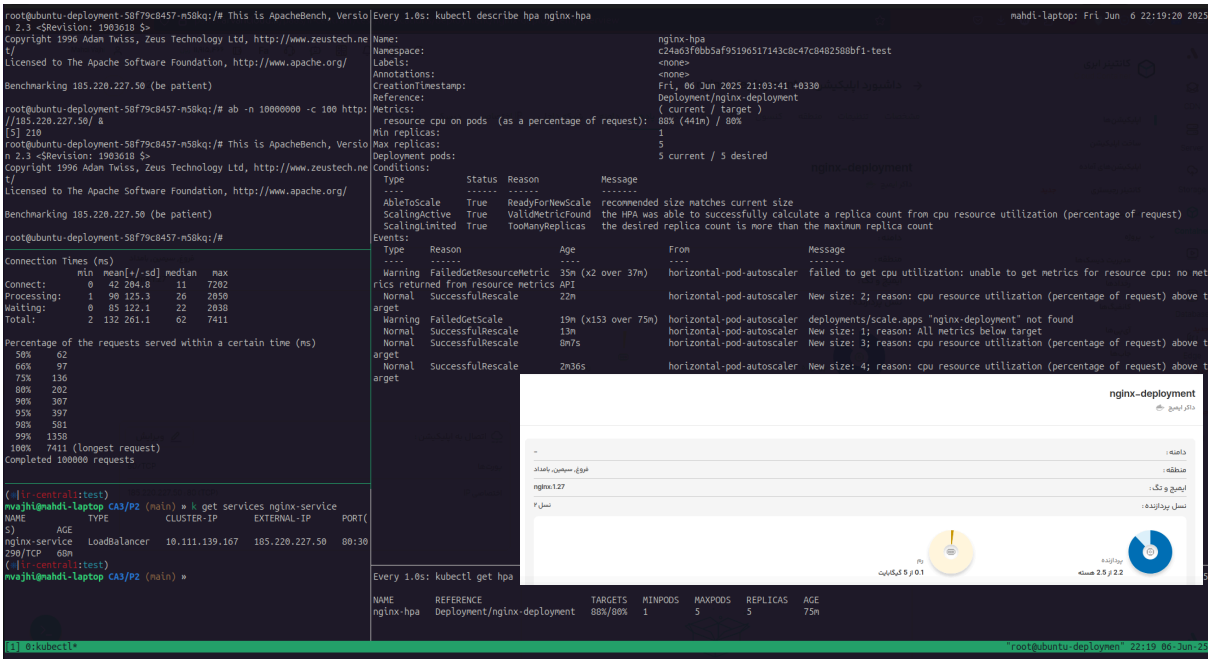
```
apiVersion: v1
kind: Service
metadata:
  name: nginx-service
  labels:
      run: nginx
spec:
  ports:
  - port: 80
      targetPort: 80
  selector:
      app: nginx
  type: LoadBalancer
```

در نهایت اسکیل خودکار رو تعریف می کنیم که از ۱ تا ۵ اسکیل شود و شرط آن را مشغول بود ۸۰ درصد cpu می گذاریم.

```
apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
metadata:
  name: nginx-hpa
spec:
  scaleTargetRef:
      apiVersion: apps/v1
      kind: Deployment
      name: nginx-deployment
  minReplicas: 1
  maxReplicas: 5
  metrics:
  - type: Resource
      resource:
      name: cpu
      target:
      type: Utilization
      averageUtilization: 80
```

آزمایش

بعد از تلاش های فراوان موفق نشدیم روی kind متریک سرور را به درستی بیاوریم. بنابراین روی ابر امتحان می کنیم. همچنین در کنار آن ۲ پاد بالا می آوریم تا از طریق آنها روی nginx لود بگذاریم. همانطور که از لاگ ها مشخص است مقیاس سرور تا ۵ زیاد می شود.

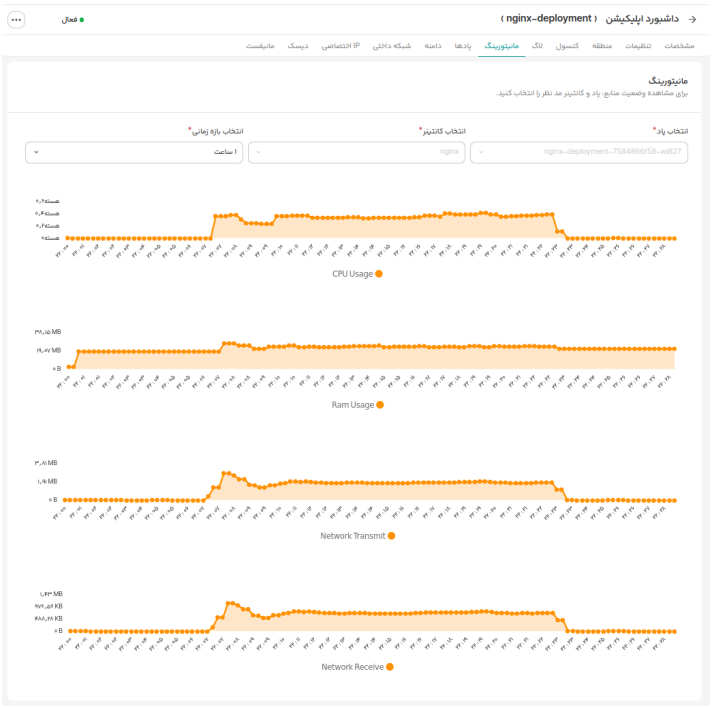


مجدد بعد از چند دقیقه و اتمام لود مقیاس کاهش یافت.

```
Events:
  Type     Reason                   Age                From                       Message
  ----     ------                   ----               ----                       -------
  Warning  FailedGetResourceMetric  44m (x2 over 46m)  horizontal-pod-autoscaler  failed to get cpu utilization: unable to get metrics for resource cpu: no met
rics returned from resource metrics API
  Normal   SuccessfulRescale        31m                horizontal-pod-autoscaler  New size: 2; reason: cpu resource utilization (percentage of request) above t
arget
  Warning  FailedGetScale           28m (x153 over 84m)  horizontal-pod-autoscaler  deployments/scale.apps "nginx-deployment" not found
  Normal   SuccessfulRescale        17m                horizontal-pod-autoscaler  New size: 3; reason: cpu resource utilization (percentage of request) above t
arget
  Normal   SuccessfulRescale        11m                horizontal-pod-autoscaler  New size: 4; reason: cpu resource utilization (percentage of request) above t
arget
  Normal   SuccessfulRescale        48s                horizontal-pod-autoscaler  New size: 4; reason: All metrics below target
  Normal   SuccessfulRescale        33s (x2 over 22m)  horizontal-pod-autoscaler  New size: 1; reason: All metrics below target
```

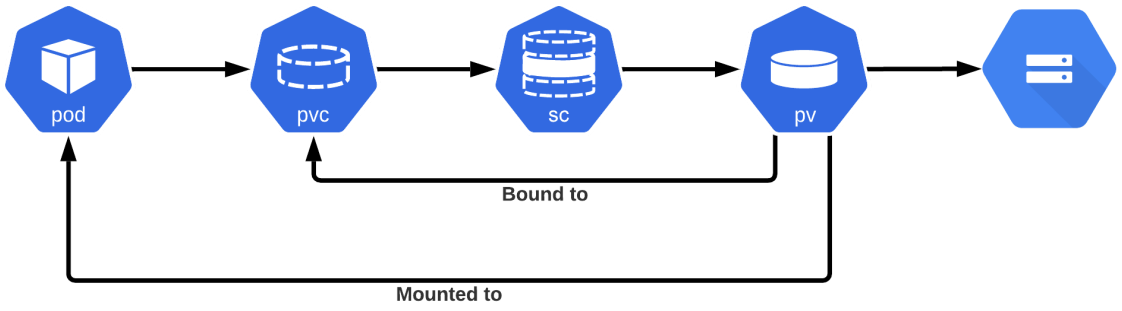
```
Every 1.0s: kubectl get hpa                                                    mahdi-laptop: Fri Jun  6 22:28:32 2025

NAME        REFERENCE                     TARGETS   MINPODS   MAXPODS   REPLICAS   AGE
nginx-hpa   Deployment/nginx-deployment   0%/80%    1         5         1          84m
```

مانیتورینگ پاد اول در طول تست:



سوالات تشریحی

حافظه در کوبرنتیز



در سرویس های stateful ما نیاز به حافظه ثابت داریم. برای این کار ما لازم است به پاد حافظه تخصیص دهیم. اما یک تفاوتی با داکر وجود دارد، در کوبرنتیز ما به چندین مشتری خدمات می دهیم بنابراین هر

مشتری نمی تواند مستقیم به حافظه ما دسترسی داشته باشد و تغییر ایجاد کند همچنین مسائل حسابرسی نیز در میان است. بنابراین می آییم به این صورت کار می کنیم که مشتری در پاد خود از pvc استفاده می کند و ما به صورت دستی یا با sc از حافظه خود یک pv می گیریم و به آن pvc متصل می کنیم. همانطور که گفته شد sc این روند را برای ما خودکار می کند.

CNI

https://g.co/gemini/share/c5e94eae54af

Calico و Cilium دو CNI (Container Network Interface) محبوب در Kubernetes هستند که وظیفه شبکه‌بندی پادها و اعمال سیاست‌های امنیتی شبکه را بر عهده دارند. Calico عمدتاً بر پایه پروتکل‌های مسیریابی لایه 3 (مانند BGP) و استفاده از IPtables برای اعمال Network Policy کار می‌کند. این پلاگین با رویکرد مسیریابی مستقیم (Underlay) یا تونل‌زنی (Overlay با IPIP/VXLAN) پادها را به شبکه متصل کرده و از قابلیت‌های قدرتمند خود در تعریف Global Network Policy و ادغام با زیرساخت‌های شبکه موجود، شناخته شده است. Calico برای محیط‌های On-premise و Hybrid Cloud که نیاز به کنترل دقیق بر شبکه زیرین و سیاست‌های امنیتی لایه 3/4 دارند، انتخابی قوی و مطمئن محسوب می‌شود.

در مقابل، Cilium از تکنولوژی مدرن و پرسرعت eBPF (Extended Berkeley Packet Filter) در هسته لینوکس بهره می‌برد. این رویکرد به Cilium اجازه می‌دهد تا قابلیت‌های شبکه و امنیتی را با عملکرد بسیار بالا و بدون نیاز به IPtables سنتی پیاده‌سازی کند. Cilium علاوه بر عملکرد برتر، قابلیت‌های امنیتی پیشرفته‌ای مانند Network Policy های لایه 7 (بر اساس پروتکل‌های کاربردی مانند HTTP/DNS) و ابزارهای Observability عمیق (مانند Hubble) را ارائه می‌دهد. همچنین، Cilium می‌تواند جایگزین kube-proxy شده و Load Balancing سرویس‌ها را نیز به صورت کارآمدتری انجام دهد. Cilium برای محیط‌هایی که به نهایت کارایی، امنیت پیشرفته لایه 7 و دید عمیق در ترافیک شبکه نیاز دارند، گزینه‌ای عالی است، البته با این پیش‌نیاز که هسته لینوکس سیستم از eBPF پشتیبانی کند.

Affinity

شرط افینیتی گفته شده پیچیده نیست و با بررسی برچسب می توان اعتبار سنجی کرد.

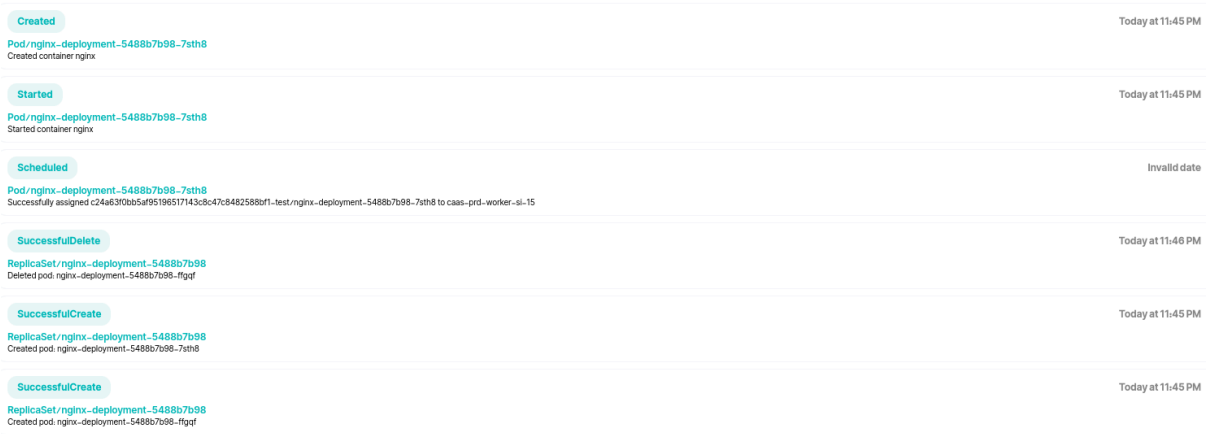
```
affinity:
podAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
      - labelSelector:
      matchExpressions:
      - key: app
            operator: In
            values:
            - redis
      topologyKey: "kubernetes.io/hostname"
```

ابتدا بدون افینیتی می آوریم:

عنوان	منطقه	وضعیت	پادها
redis-deployment	بامداد, سیمین	فعال	۲/۲
nginx-deployment 185.220.227.50	بامداد, فروغ	فعال	۲/۲

وقتی افینیتی رو اعمال می کنیم پاد از روی فروغ برداشته می شود و روی سیمین می رود.

عنوان	منطقه	وضعیت	پادها
redis-deployment	بامداد, سیمین	فعال	۲/۲
nginx-deployment 185.220.227.50	بامداد, سیمین	فعال	۲/۲



KEDA

https://chatgpt.com/share/68434fbc-97cc-8001-a273-36bea7116c22

KEDA (مخفف Kubernetes Event-driven Autoscaling) ابزاری متن‌باز برای کوبرنتیز است که امکان مقیاس‌پذیری خودکار برنامه‌ها را بر اساس رویدادهای خارجی فراهم می‌کند. برخلاف autoscaler پیش‌فرض کوبرنتیز که بیشتر بر اساس منابع سیستمی مانند CPU و RAM عمل می‌کند، KEDA می‌تواند به منابعی مانند Prometheus، HTTP، Azure Queue، RabbitMQ، Kafka و ده‌ها منبع دیگر متصل شود و با افزایش یا کاهش بار، تعداد پادها را به‌صورت هوشمند تنظیم کند. این ابزار برای برنامه‌های رویداد محور بسیار کاربردی است و باعث استفاده بهینه از منابع و پاسخ‌گویی سریع‌تر به تغییرات بار می‌شود.

بررسی پورت ها

پیاده سازی

فیلتر کردن بسته های نامناسب

در ابتدا بسته های با ساختار گفته شده را انتخاب می کنیم.

```
def validate_layer(packet):
   required_layers = [Ether, IP, TCP]
   return all(layer in packet for layer in required_layers)
valid_packets = [p for p in all_packets if validate_layer(p)]
```

بررسی SYN, SYN-ACK در بسته ها

با بررسی پرچم های tcp این موضوع را بررسی می کنیم.

```
def update_tcp_flag_dict(packet, syn_count, syn_ack_count):
   if packet[TCP].flags == "S":
       syn_count[packet[IP].src] = syn_count.get(packet[IP].src, 0) + 1
   elif packet[TCP].flags == "SA":
       syn_ack_count[packet[IP].dst] = syn_ack_count.get(packet[IP].dst,
0) + 1
```

و سپس با یک حلقه روی همه بررسی می کنیم و می شماریم.

```
def count_S_SA(packets):
   syn_count = {}
   syn_ack_count = {}
   for p in packets:
       update_tcp_flag_dict(p, syn_count, syn_ack_count)
   return syn_count, syn_ack_count
```

پیدا کردن آیپی های مشکوک

با پیمایش روی آیپی های SYN و بررسی همان آیپی در SYN-ACK شرط ۳ برابری را بررسی می کنیم.

```
def find_suspicious_ips(syn_counts, synack_counts):
   suspicious_ips = []
   for ip in syn_counts.keys():
       syn_sent = syn_counts[ip]
       synack_received = synack_counts.get(ip, 0)

       if syn_sent > synack_received * RATE:
           suspicious_ips.append((ip, syn_sent, synack_received, syn_sent -
```

```
synack_received))
    return suspicious_ips
```

نمایش خروجی

درنهایت داده ها را در یک جدول پانداس نمایش می دهیم.

```
def display_suspicious_ips(suspicious_list):
    df = pd.DataFrame(suspicious_list, columns=['IP', 'SYN_sent',
'SYN_ACK_received', 'Difference'])
    df.sort_values(by='Difference', ascending=False, inplace=True)
    print("\nSuspicious IPs:")
    print(df.to_string(index=False))
    print(f"\nTotal suspicious IPs found: {len(suspicious_list)}")
```

تابع main

```
def main():
    if len(sys.argv) != 2:
        print("Usage: python syn_scanner.py <pcap_file>")
        return

    pcap_file = sys.argv[1]

    print(f"Reading {pcap_file}...")
    all_packets = rdpcap(pcap_file)
    print(f"Total packets read: {len(all_packets)}")

    print("Validating packets...")
    valid_packets = [p for p in all_packets if validate_layer(p)]
    print(f"Valid packets (Ether+IP+TCP): {len(valid_packets)}")

    print("Counting SYN and SYN-ACK packets...")
    syn_c, synack_c = count_S_SA(valid_packets)

    print("Finding suspicious IPs...")
    suspicious_list = find_suspicious_ips(syn_c, synack_c)

    display_suspicious_ips(suspicious_list)
```

اجرا

ابتدا روی داده کوچک کد را اجرا می کنیم و نتیجه به شکل زیر می شود.

```
Reading reduced_sample.pcap...
Total packets read: 1497
Validating packets...
```

```
Valid packets (Ether+IP+TCP): 1157
Counting SYN and SYN-ACK packets...
Finding suspicious IPs...

Suspicious IPs:
            IP  SYN_sent  SYN_ACK_received  Difference
 128.3.23.117        44                 8          36
   128.3.23.5        34                 1          33
 128.3.23.158        23                 2          21
   128.3.23.2        16                 0          16
128.3.164.248         4                 0           4
128.3.164.249         1                 0           1

Total suspicious IPs found: 6
```

سپس روی داده بزرگ تر اجرا می کنیم بعد از تقریبا ۱۰ دقیقه و مصرف تقریبا ۱۴ گیگ رم و پر شدن رم و سواپ لپ تاپ کرش کرد. سپس داده را روی کگل بردیم و آنجا اجرا کردیم بعد از تقریبا یک ربع و با مصرف ۲۰ گیگ رم بالاخره تمام داده بارگزاری شد و توانستیم تحلیل را روی آن انجام دهیم.

```
Reading /kaggle/input/pcap-data/sample.pcap...
Total packets read: 5700526
Validating packets...
Valid packets (Ether+IP+TCP): 2313433
Counting SYN and SYN-ACK packets...
Finding suspicious IPs...

Suspicious IPs:
            IP  SYN_sent  SYN_ACK_received  Difference
 128.3.23.117        44                 8          36
   128.3.23.5        34                 1          33
 128.3.23.158        23                 2          21
   128.3.23.2        16                 0          16
128.3.164.248         4                 0           4
128.3.164.249         1                 0           1

Total suspicious IPs found: 6
```

RAM
20.4GiB
Max 30GiB

بررسی فایل جهت شناخت مهاجم

در تحلیل ترافیک شبکه و بررسی فایل‌های PCAP، برخی نشانه‌ها می‌توانند حاکی از فعالیت مشکوک یا حمله باشند. برای نمونه، مشاهده ارسال درخواست از یک دستگاه با چند آدرس IP متفاوت می‌تواند نشان‌دهنده تلاش برای پنهان‌کاری یا دور زدن فیلترها باشد. همچنین، اگر درخواست‌های مکرری از سوی IPهای ناشناس یا سرورهای دیگر شبکه به پورت‌های مدیریتی مانند SSH یا Remote Desktop صورت

گیرد، باید به‌عنوان تلاشی برای دسترسی غیرمجاز تلقی شود. در همین راستا، ارتباطات HTTP نیز باید به‌دقت بررسی شوند تا هیچ داده حساسی مانند رمز عبور یا اطلاعات شخصی به‌صورت واضح و بدون رمزنگاری منتقل نشده باشد، چرا که در حالت ایده‌آل نباید هیچ ترافیکی به مقصد بیرون از طریق HTTP برقرار شود و تنها HTTPS مجاز است.

از دیگر الگوهای رایج حمله می‌توان به اسکن پورت‌ها، حملات brute-force، حملات DoS یا DDoS و حملات ARP poisoning اشاره کرد. پروتکل‌هایی همچون DNS، DHCP، ARP و ICMP معمولاً هدف مناسبی برای این‌گونه حملات هستند و باید ترافیک مربوط به آن‌ها به‌دقت بررسی شود. مشاهده ارتباط میان سرورها از طریق پورت‌های غیر معمول مانند SSH روی پورت متفاوت نیز می‌تواند نشانه‌ای از دور زدن نظارت‌های امنیتی باشد. اهداف این حملات ممکن است شامل ایجاد اختلال در سرویس، دسترسی غیرمجاز، بهره‌برداری از آسیب‌پذیری‌های نرم‌افزار یا سیستم‌عامل، یا انجام شناسایی و نقشه‌برداری شبکه باشند. برخی حملات نیز از طریق ارسال بسته‌هایی با ساختار نامعمول و طراحی‌شده برای آسیب‌زدن به حافظه (مثلاً buffer overflow) انجام می‌شوند و باید با دقت بررسی شوند.