A DevOps Engineer's Journey

`Author: Kasra Jamshidi`

Github

HTTP
HTTP (HTTP status cods)
Set IP
NGINX
CA
CERTBOT (CA)
DNS
ACL
JAIL (BIND)
LVM
NFS
TTFB
FIREWALL
ICMP(echo)
NAMESPACES
DOCKER
DOCKER HUB
HARBOR
DOCKER VOLUME
DOCKER NETWORK
DOCKER COMPOSE

ANSIBLE

To avoid confusion, since I'm using the history of my server, my first server's IP is 192.168.69.54 and the second one is 192.168.69.55.

"This repository contains personal notes and resources related to DevOps. It is not intended for production use, and should not be relied upon for critical tasks. Use at your own discretion."

HTTP (Hypertext Transfer Protocol) :

یکی از پروتکل‌های اصلی در وب است که برای انتقال اطلاعات بین کلاینت (معمولاً مرورگر) و سرور استفاده می شود .

اجزای اصلی HTTP

HTTP Request (درخواست)

درخواست HTTP از سمت کلاینت به سرور ارسال می‌شود و شامل دو بخش اصلی است:

1. **Header**: اطلاعات متا درباره درخواست، مثل نوع درخواست (GET، POST، و غیره)، نوع محتوا، و غیره.
2. **Body**: داده‌هایی که همراه درخواست ارسال می‌شوند (معمولاً در POST یا PUT).

HTTP Response (پاسخ)

پاسخ HTTP از سمت سرور به کلاینت ارسال می‌شود و شامل دو بخش اصلی است:

1. **Header**: اطلاعات متا درباره پاسخ، مثل نوع محتوا و وضعیت درخواست.
2. **Body**: محتوای واقعی پاسخ، مثل HTML، JSON، یا فایل.

روش‌های HTTP (HTTP Methods)

برای تعریف نوع درخواست کلاینت به سرور استفاده می‌شوند:

متد	کاربرد
GET	درخواست برای دریافت یک منبع مشخص بدون تغییر در سرور.
POST	ارسال داده به سرور برای پردازش (معمولاً برای ارسال فرم).
PUT	ارسال داده برای ذخیره در یک مکان مشخص.
DELETE	حذف یک منبع مشخص روی سرور.
HEAD	مشابه GET، اما فقط هدر را بدون بدنه پاسخ می‌گیرد.
OPTIONS	اطلاعات درباره متدهای پشتیبانی شده برای یک منبع خاص را برمی‌گرداند.
TRACE	برای آزمایش و اشکال‌زدایی مسیر درخواست به سرور استفاده می‌شود.

کدهای وضعیت HTTP (HTTP Status Codes)

کدهای وضعیت در پاسخ HTTP نشان‌دهنده وضعیت پردازش درخواست هستند. این کدها به چند دسته اصلی تقسیم می‌شوند:

دسته	معنی
1XX	اطلاعاتی (در حال پردازش درخواست)
2XX	موفقیت (درخواست با موفقیت پردازش شد)
3XX	تغییر مسیر (ریدایرکت به یک آدرس دیگر)
4XX	خطای کلاینت (مشکل در درخواست کلاینت)
5XX	خطای سرور (مشکل در سرور در پردازش درخواست)

کدهای مهم HTTP Status

دسته 1XX: Informational

- **100 Continue**: سرور آماده دریافت ادامه درخواست است.
- **101 Switching Protocols**: این کد نشان می‌دهد که سرور درخواست کلاینت برای تغییر پروتکل را پذیرفته است. به‌عنوان مثال، اگر کلاینت بخواهد از HTTP به WebSocket تغییر کند، این کد ارسال می‌شود.
- **102 Processing** (WebDAV): این کد نشان می‌دهد که سرور در حال پردازش درخواست است، اما پاسخ نهایی هنوز آماده نیست. معمولاً در عملیات‌های طولانی‌مدت در WebDAV استفاده می‌شود.

دسته 2XX: Success

- **200 OK**: درخواست با موفقیت انجام شد.
- **201 Created**: منبع جدیدی ایجاد شده است.
- **202 Accepted**: این کد به این معنی است که درخواست دریافت شده و در حال پردازش است، اما هنوز تکمیل نشده است.
- **204 No Content**: این کد نشان می‌دهد که درخواست با موفقیت انجام شده، اما پاسخی از سمت سرور برنمی‌گردد (هیچ محتوایی ارسال نمی‌شود). معمولاً در پاسخ به درخواست‌هایی مثل حذف منابع استفاده می‌شود.
- **206 Partial Content** این کد زمانی استفاده می‌شود که کل محتوا ارسال نشده و فقط بخشی از آن (بر اساس درخواست کاربر) ارسال شده است. به‌طور مثال، در دانلودهای قطع‌شده یا دانلودهای تکه‌ای.

دسته 3XX: Redirect

- **301 Moved Permanently**: منبع به صورت دائمی به مکان جدید منتقل شده است.
- **302 Found (Temporary Redirect)**: منبع به صورت موقت به مکان جدید منتقل شده است.
- 304 **Not Modified**: این کد نشان می‌دهد که منبع مورد درخواست از زمان آخرین کش شدن توسط مرورگر، تغییر نکرده است. در نتیجه، مرورگر می‌تواند نسخه کش‌شده را استفاده کند.
- 307 **Temporary Redirect**: مشابه کد 302 است، اما در اینجا مرورگر باید از همان متد HTTP (مانند POST یا GET) که در درخواست اصلی استفاده شده است، برای درخواست جدید نیز استفاده کند.
- 308 **Permanent Redirect**: مشابه کد 301 است، با این تفاوت که مرورگر ملزم است از همان متد HTTP استفاده کند. این کد نشان‌دهنده انتقال دائمی است.

دسته 4XX: Client Error

- **400 Bad Request**: درخواست نادرست (Syntax مشکل دارد).
- **401 Unauthorized**: احراز هویت لازم است.
- **403 Forbidden**: دسترسی به منبع ممنوع است.
- **404 Not Found**: منبع پیدا نشد.
- 405 Method Not Allowed: متد HTTP استفاده‌شده (مثلاً GET یا POST) برای این منبع مجاز نیست.
- **406 Not Acceptable** : سرور نمی‌تواند پاسخی بر اساس محتوای درخواستی کلاینت ارائه دهد (مثلاً اگر کلاینت فرمت خاصی را درخواست کرده باشد و سرور نتواند آن را تولید کند).
- **408 Request Timeout**: کلاینت برای ارسال درخواست بیش از حد طول کشیده است، بنابراین سرور ارتباط را قطع کرده است.
- **409 Conflict**: این کد نشان می‌دهد که در اجرای درخواست یک تضاد وجود دارد، مثلاً هنگام تلاش برای به‌روزرسانی یک منبع که هم‌زمان توسط درخواست دیگری تغییر کرده است.

- **410 Gone**:،404 برخلاف احتمالاً به صورت دائمی حذف شده است. منبع موردنظر دیگر در سرور موجود نیست و .این کد نشان می‌دهد که حذف منبع آگاهانه بوده است

دسته 5XX: Server Error

- **500 Internal Server Error**: خطای داخلی سرور.چ
- **501 Not Implemented**: سرور توانایی یا قابلیت اجرای متد درخواست‌شده را ندارد. مثلاً اگر کلاینت از متدی استفاده کند که سرور آن را پشتیبانی نمی‌کند
- **502 Bad Gateway**: مشکل در ارتباط با سرور بالادستی
- **503 Service Unavailable**: سرویس در دسترس نیست
- **504 Gateway Timeout**: زمان پاسخ‌دهی سرور بالادستی تمام شده است
- **505 HTTP Version Not Supported**: سرور نسخه HTTP استفاده‌شده در درخواست کلاینت را پشتیبانی نمی‌کند.

ساختار HTTP Request

```
GET /example HTTP/1.1
Host: www.example.com
User-Agent: Mozilla/5.0
Accept: text/html,application/xhtml+xml
```

توضیحات:

1. خط اول: متد، آدرس منبع، و نسخه پروتکل.
2. هدرها: اطلاعاتی مثل نوع مرورگر (User-Agent) و نوع داده مورد پذیرش.

ساختار HTTP Response

```
HTTP/1.1 200 OK
Content-Type: text/html
Content-Length: 125

<html>
<body>
<h1>Success!</h1>
</body>
</html>
```

توضیحات:

1. خط اول: نسخه پروتکل، کد وضعیت، و توضیح وضعیت.
2. هدرها: نوع داده و اندازه محتوا.

تنظیم IP

as a webserver
as a reverse proxy
as a loadbalance

بر روی ماشین مجازی :

1. set IP Address
2. change password
3. Install bash-completion

   username : root password ==> روی ماشین ها کلیک کرده
4. IPa : چک کردن IP روی ماشین
5. puTTY : باز کردن پوتی و وارد کردن IP مورد نظر
6. username : root password : -
7. Redhat ==> vi ete/ sysconfig/Network-scripts/
8. Systemctl restart Netwark Manager
9. iptables -F

ifcfg.ens32 ==> space نمایش کانفیگ کارت شبکه با زدن

تنظیم دستی IP

```
| BOOTPROTO = None or Static
| IPADDR = مورد نظر ip
| MASK = 255.255.255.0
| GATEWAY = 192.168.67.254
| ONBOOT = YES
```

اگر server ری استارت شود تنظیمات شبکه UP می شود .

راه دوم

1. nmtui
2. Edit a connection
3. Edit
4. IPV4 { Address , Gateway , DNS }
5. Aatomatically connect
6. Back
7. Active

مرحله دوم : ssh زدن به سرور
مرحله سوم : تغییر دادن پسورد
مرحله چهارم : `dnf -y install vim bash-completion`

NGINX

یک وب سرور است ولی عموما به عنوان یک reverse proxy استفاده می شود . :NIGNX

- نباید nignx رو از ریپوز دیفالت دانلود کرد چون ممکن است نسخه قدیمی و دچار باگ امنیتی باشد .

https://nginx.org/en/linux_packages.html

روی NGINX می‌توانیم برنامه‌های نوشته‌شده با جاوا و .NET را تنظیم و اجرا کنیم. برای نصب، باید ریپوزیتوری مرتبط با آن را اضافه کنیم.

نصب NGINX:

مراحل نصب در RedHat:

ورود به دایرکتوری مخازن :

```
cd /etc/yum.repos.d/
```

ویرایش فایل مخزن:

```
vim nginx.repo
```

تغییر نام فایل مخزن قدیمی:

```
mv rabbitmq.local.repo rabbitmq.local.bak
```

نصب NGINX:

```
yum -y install nginx
```

فعال‌سازی و شروع سرویس NGINX :

```
systemctl enable --now nginx
```

پاک کردن قوانین فایروال:

```
iptables -F
```

غیرفعال کردن سرویس firewalld:

```
systemctl disable --now firewalld
```

دایرکتوری ریشه پیش‌فرض NGINX:

دایرکتوری ریشه‌ی NGINX (جایی که محتوای وب‌سایت ذخیره می‌شود):

1. ورود به دایرکتوری NGINX HTML:

```
cd /usr/share/nginx/html/
```

تغییر نام فایل index.html:

```
mv index.html index.html.bak
```

ایجاد یک فایل جدید index.html:

```
echo "hello" > index.html
```

HTTPS:

ما انواع حملات تحت وب داریم که مربوط به هم به انواع مختلف است. آیا داشتن HTTPS در سایت می‌تواند جلوی حملات SQL Injection را بگیرد؟ خیر، این نوع حمله مستقیماً به دیتابیس دسترسی پیدا می‌کند و داده‌ها را تغییر می‌دهد. HTTPS فقط داده‌های منتقل شده را رمزنگاری می‌کند.

ء Hypertext Transfer Protocol Secure** مخفف HTTPS است . این پروتکل داده‌های شما را به صورت رمزنگاری شده جابه‌جا می‌کند، اما به تنهایی نمی‌تواند امنیت کامل سایت را تضمین کند.

رمزنگاری (Encryption):

رمزنگاری به معنای تبدیل داده‌ها به کدهایی است که از دسترسی غیرمجاز جلوگیری می‌کند. الگوریتم‌های مختلفی برای این کار وجود دارد که اطمینان حاصل می‌کند مهاجمان نمی‌توانند داده‌های شما را تغییر داده یا بخوانند.

هش (Hash):

هش توابعی هستند که داده ورودی را به یک رشته ثابت از کاراکترها تبدیل می‌کنند. این امر برای اطمینان از صحت و یکپارچگی داده‌ها استفاده می‌شود.

شرایط یک گواهینامه معتبر (CA):

یک **مرجع صدور گواهینامه** (Certificate Authority یا CA) معتبر باید موارد زیر را تضمین کند:

1. **اعتبار گواهینامه**: اطمینان از این که گواهینامه توسط یک مرجع معتبر صادر شده و به صورت قانونی و قابل اعتماد قابل استفاده است.
2. **تاریخ اعتبار**: بررسی این که گواهینامه همچنان معتبر بوده و از تاریخ انقضا یا شروع خارج نشده است.
3. **نام گواهینامه**: تطابق نام (Subject Name) گواهینامه با مالک یا دامنه‌ای که برای آن صادر شده است.
4. **فهرست ابطال گواهینامه (CRL)**: گواهینامه نباید در لیست ابطال گواهینامه (Certificate Revocation List یا CRL) یا پایگاه داده مشابهی که گواهینامه‌های لغوشده را نگهداری می‌کند، قرار داشته باشد.

HPKP (HTTP Public Key Pinning):

HPKP یک ویژگی امنیتی برای وب‌سایت‌های HTTPS بود که از حملات جعل گواهی‌نامه (Certificate Forgery) جلوگیری می‌کرد. این کار با مشخص کردن یک یا چند کلید عمومی (Public Key) مورد اعتماد برای مرورگر انجام می‌شد.

عملکرد HPKP:

1. سایت یک هدر خاص به نام `Public-Key-Pins` به مرورگر ارسال می‌کرد.
2. این هدر لیستی از کلیدهای عمومی مورد اعتماد سرور را در خود داشت.
3. مرورگر این کلیدها را ذخیره می‌کرد (Pin می‌کرد) و هنگام بازدید دوباره از سایت، بررسی می‌کرد که آیا گواهی دریافتی با کلیدهای مشخص‌شده در لیست مطابقت دارد یا خیر.
4. اگر مطابقت نداشت، مرورگر از برقراری ارتباط جلوگیری می‌کرد تا از حملاتی مانند **Man-in-the-Middle (MITM)** جلوگیری شود.

چرا HPKP دیگر استفاده نمی‌شود؟

- اگر کلیدها اشتباه یا منقضی می‌شدند، سایت ممکن بود برای کاربران غیرقابل دسترس شود.
- مدیریت آن پیچیده بود و خطاهای پیکربندی می‌توانست به مشکلات جدی منجر شود.
- این مکانیزم جای خود را به مکانیزم‌های ساده‌تر و امن‌تری مانند Certificate Transparency و HSTS داده است.

تنظیمات Self-Signed CA و ایجاد Certificate Chain

1. مسیردهی به فایل‌های تنظیمات NGINX

```
cd /etc/nginx/conf.d/
```

2. حذف فایل‌های قبلی (در صورت لزوم)

```
rm -rf *
```

3. ایجاد کلید خصوصی (Private Key)

```
openssl genrsa -des3 -out server.key 2048
```

4. حذف رمز عبور از کلید خصوصی

```
openssl rsa -in server.key -out private.key
```

5. **تبدیل کلید خصوصی برای استفاده توسط CA**
   فایل `private.key` برای استفاده آماده است.
6. **ایجاد CSR (Certificate Signing Request)**
   - اگر نیاز به CSR برای ارسال به CA داشتید، این مرحله انجام می‌شود.
7. **ایجاد گواهی (Certificate)**

```
openssl req -x509 -nodes -days 365 -newkey rsa:2048 -keyout private.key
```

مشخص کردن اطلاعات گواهی

8. شامل نام، ایمیل، CA و دیگر اطلاعات.

قرار دادن گواهی‌های دیجیتال در سخت‌افزار به جای NGINX یا نرم‌افزارهای دیگر، یک روش بهینه برای افزایش عملکرد و امنیت است. در ادامه دلایل و نحوه انجام این کار توضیح داده شده است:

- **افزایش امنیت:**
  - کلید خصوصی هرگز از سخت‌افزار خارج نمی‌شود، بنابراین امکان سرقت آن کاهش می‌یابد.
  - سخت‌افزارهای اختصاصی مانند HSM (Hardware Security Module) مکانیزم‌های امنیتی داخلی برای جلوگیری از دسترسی غیرمجاز دارند.
- **بهبود عملکرد:**
  - سخت‌افزارهای اختصاصی قادر به پردازش عملیات رمزنگاری (مانند TLS Handshake) با سرعت بسیار بیشتری هستند.
  - کاهش بار CPU سرور اصلی، مخصوصاً در سرورهایی با درخواست‌های رمزنگاری بالا.
- **مدیریت آسان‌تر:**
  - سخت‌افزارهای مخصوص گواهی‌ها امکان مدیریت متمرکز کلیدها و گواهی‌ها را فراهم می‌کنند.
  - امکان استفاده از گواهی‌ها برای چندین سرور یا برنامه به طور همزمان.

گام‌های گرفتن گواهی از Let's Encrypt

1. نصب Certbot :

سرت بات (Certbot) ابزاری است که به صورت خودکار برای شما گواهی Let's Encrypt دریافت می‌کند. برای نصب آن، از دستورات زیر استفاده کنید:

در CentOS/RHEL:

```
yum install epel-release -y
yum install certbot python3-certbot-apache -y
```

در Ubuntu/Debian:

```
apt update
apt install certbot python3-certbot-apache -y
```

2. درخواست گواهی

بعد از نصب Certbot، کافی است دستور زیر را اجرا کنید:

2. درخواست گواهی

بعد از نصب Certbot، کافی است دستور زیر را اجرا کنید:

```
certbot --apache
```

این دستور:

- دامنه‌های پیکربندی‌شده در **Apache** را شناسایی می‌کند.
- فایل‌های لازم برای گواهی را ایجاد می‌کند.
- گواهی از **Let's Encrypt** دریافت می‌کند.
- سرور را برای استفاده از HTTPS به‌روز می‌کند.

3. اگر سرور Apache ندارید (DNS-based)

اگر از وب سروری غیر از Apache استفاده می‌کنید، یا فقط قصد دارید گواهی بگیرید بدون تغییر سرور :

```
certbot certonly --standalone -d example.com -d www.example.com
```

`-d example.com` : دامنه‌ای که می‌خواهید گواهی برای آن بگیرید.

TIP : این روش نیاز دارد که پورت 80 روی سرور باز باشد

4. تمدید خودکار گواهی

Let's Encrypt گواهی‌های 90 روزه صادر می‌کند، بنابراین باید تمدید خودکار فعال شود. Certbot به صورت پیش‌فرض یک cron job یا systemd timer برای تمدید خودکار ایجاد می‌کند.

برای تست تمدید:

```
certbot renew --dry-run
```

DNS

دی‌ان‌اس (DNS) سیستمی است که وظیفه تبدیل نام دامنه‌ها به آدرس‌های IP را بر عهده دارد تا دستگاه‌ها در شبکه بتوانند با یکدیگر ارتباط برقرار کنند.

انواع سرور DNS:

1. Master Zone (زون اصلی)

این زون مثل دفتر اصلی تلفن است. شما تمام اطلاعات دامنه‌ها و IPها را در اینجا ذخیره می‌کنید.
مثال: فرض کنید شما مالک دامنه `jamshidi.ir` هستید. اطلاعاتی مثل `www.jamshidi.ir` برابر است با `192.168.5.131` را در این زون وارد می‌کنید. این زون مرکز اصلی داده‌های دامنه است.

2. Slave Zone (زون پشتیبان)

این زون کپی زون Master است و اگر Master در دسترس نباشد، از این زون استفاده می‌شود.
مثال: شما یک سرور پشتیبان دارید. اگر سرور اصلی DNS شما خراب شود، سرور Slave می‌تواند به درخواست‌های DNS پاسخ دهد.

3. Forward Zone (زون ارسال‌کننده)

این زون وقتی استفاده می‌شود که سرور DNS شما خودش اطلاعات را ندارد و باید سوال‌ها را به یک سرور دیگر ارسال کند.
مثال: سرور DNS شما نمی‌داند `example.com` کجاست، پس از یک سرور خارجی (مثل Google DNS) می‌پرسد.

4. Hint Zone (زون راهنما)

این زون لیستی از آدرس سرورهای Root DNS است و وقتی استفاده می‌شود که هیچ اطلاعاتی در دسترس نیست.
مثال: سرور DNS شما نمی‌داند از کجا شروع کند، پس به سرورهای Root (مثل `com.` یا `ir.`) مراجعه می‌کند و اطلاعات اولیه را از آنجا می‌گیرد.

اصطلاحات:

- **TLD (Top Level Domain):** سطح اول دامنه (مانند `.com` , `.ir`).
- **FQDN (Fully Qualified Domain Name):** نام کامل دامنه (مانند `mail.cando.com`).
- **Authoritative DNS:** پاسخ‌دهنده اصلی و دقیق به درخواست‌ها.
- **Non-Authoritative DNS:** پاسخ‌هایی که کش شده‌اند و معتبر نیستند.

مفاهیم اولیه:

1. **Recursion:**
   - باید غیر فعال باشد تا سرور DNS توسط بات‌ها مورد پرسش قرار نگیرد.
2. **DNSSEC:**
   - یک پروتکل امنیتی برای تضمین صحت و اعتبار اطلاعات DNS.
3. **TTL (Time to Live):**
   - برای تغییر آدرس IP یک وب‌سایت، مقدار **TTL** باید کاهش یابد (۱ یا ۲ دقیقه) تا تغییرات سریع‌تر اعمال شوند.

کانفیگ فایل‌ها:

نصب DNS Server:

```
yum -y install bind bind-utils
```

شروع سرویس:

```
systemctl enable named
systemctl start named
```

بررسی وضعیت سرویس:

```
systemctl status named
```

فایل تنظیمات اصلی DNS:

```
vim /etc/named.conf
```

فایل اطلاعات zone :

```
vim /var/named/data/jamshidi.db
```

نکات مدیریتی:

1. **قرار دادن سرور اصلی (Master) در DMZ:**
   - ممنوع است، امنیت کاهش می‌یابد.
2. **ساخت سرور Slave:**
   - یک سرور ثانویه تنظیم کنید تا در صورت خرابی سرور اصلی، عملکرد DNS ادامه داشته باشد.
3. **به‌روزرسانی Slave:**
   - از دستور زیر استفاده کنید:

```
rndc reload
```

مقدار **Serial** در فایل زون باید ۱ عدد افزایش یابد.

در فایل **زون** (zone)، مقدار **Serial** یک شماره نسخه برای رکوردهای DNS آن زون است. این مقدار نشان‌دهنده تغییرات در فایل زون است و به سرورهای **Slave** (ثانویه) کمک می‌کند تا تشخیص دهند که آیا فایل زون به‌روزرسانی شده است یا نه.

چرا باید مقدار Serial را افزایش دهیم؟

وقتی تغییری در رکوردهای DNS فایل زون اعمال می‌کنید (مثل افزودن یک رکورد جدید یا ویرایش رکوردهای موجود)، مقدار **Serial** باید **۱ واحد افزایش یابد**. این افزایش ضروری است زیرا سرورهای Slave تنها زمانی فایل زون جدید را از

سرور Master می‌گیرند که مقدار Serial در فایل زون جدید، بزرگ‌تر از مقدار فعلی باشد. اگر مقدار Serial را افزایش ندهید:

- سرورهای Slave تصور می‌کنند فایل زون تغییر نکرده است.
- رکوردهای قدیمی در سرورهای Slave باقی می‌مانند.
- درخواست‌ها به‌درستی به سرورهای مقصد هدایت نمی‌شوند.

فرمت مقدار Serial

بهترین روش برای مقداردهی به Serial استفاده از فرمت تاریخ + شماره نسخه است. برای مثال:

- **2024120801**: سال 2024، ماه 12، روز 08، نسخه 01. اگر تغییرات دیگری در همان روز اعمال کنید، شماره نسخه را افزایش می‌دهید: **2024120802**.

نحوه به‌روزرسانی

1. فایل زون را با یک ویرایشگر متنی باز کنید (مثلاً `vim /var/named/data/example.com.db`).
2. مقدار Serial را پیدا کنید.
3. مقدار فعلی را یک عدد افزایش دهید.
4. فایل را ذخیره کرده و از دستور زیر برای بارگذاری مجدد زون استفاده کنید:

```
rndc reload
```

Forwarders:

برای ارسال درخواست‌های DNS به سرورهای دیگر (مثلاً Google):

```
forwarders {
    1.1.1.1;
    8.8.8.8;
};
forward only;
```

فورواردینگ در DNS به این معنی است که وقتی سرور DNS نتواند به سوال شما جواب بدهد، این سوال را به یک سرور DNS دیگر که مشخص کرده‌ایم، ارسال می‌کند تا پاسخ را از آنجا دریافت کند.

نمونه فایل کانفیگ DNS

```
options {

    listen-on port 53 { 192.168.69.54; };

    listen-on-v6 port 53 { ::1; };

    directory       "/var/named";
```

```
        dump-file       "/var/named/data/cache_dump.db";
        statistics-file "/var/named/data/named_stats.txt";
        memstatistics-file "/var/named/data/named_mem_stats.txt";
        secroots-file   "/var/named/data/named.secroots";
        recursing-file  "/var/named/data/named.recursing";
        allow-query     { any; };
        recursion no;
        dnssec-enable yes;
        dnssec-validation yes;

    managed-keys-directory "/var/named/dynamic";
        pid-file "/run/named/named.pid";
        session-keyfile "/run/named/session.key";
};
logging {
        channel default_debug {
                file "data/named.run";
                severity dynamic;
        };
};
zone "jamshidi.ir" IN {
        type master;
        file "/var/named/data/jamshidi.db";
        allow-update { none ;};
        allow-transfer { 192.168.69.55; };
};
zone "." IN {
```

```
15        type hint;

        file "named.ca";

};
```

توضیحات فایل کانفیگ DNS

1. بخش options

بخش `options` در فایل تنظیمات DNS شامل تعداد زیادی جزئیات است، اما مهم‌ترین تنظیمات آن عبارت‌اند از:

1. `listen-on port 53 { 192.168.69.54; };`
   مشخص می‌کند سرور فقط به درخواست‌های ارسالی به آدرس **192.168.69.54** روی پورت 53 پاسخ می‌دهد.)پورت پیش‌فرض (DNS
2. `allow-query { any; };`
   تعیین می‌کند که تمام کلاینت‌ها مجاز به ارسال درخواست به سرور هستند. این گزینه برای دسترسی عمومی مهم است.
3. `recursion no;`
   غیرفعال کردن **recursion**، به این معنی که سرور فقط به سوالاتی که مسئولیت آن‌ها را بر عهده دارد (authoritative) پاسخ می‌دهد و سوالات دیگر را حل نمی‌کند.
4. `dnssec-enable yes;`
   فعال کردن **DNSSEC** برای اطمینان از امنیت در پاسخ‌های DNS.
5. `dnssec-validation yes;`

   فعال کردن اعتبارسنجی **DNSSEC** برای بررسی امضاهای دیجیتالی و جلوگیری از جعل DNS.
6. `directory "/var/named";`
   مسیر اصلی ذخیره فایل‌های پیکربندی و زون‌های DNS.

این موارد، مهم‌ترین بخش‌های تنظیم `options` هستند که بر عملکرد سرور تأثیر مستقیم دارند. تنظیمات دیگر برای اهداف خاص یا جزئیات اضافی استفاده می‌شوند.

2. بخش Logging

```
logging {
        channel default_debug {
                file "data/named.run";
                severity dynamic;
        };
};
```

این بخش مربوط به تنظیمات لاگ‌ها (ثبت وقایع) است : logging

یک کانال به نام default_debug تعریف شده که لاگ‌ها را در فایل مشخصی ذخیره : channel default_debug می‌کند.

16 مسیر فایل لاگ که اطلاعات لاگ به آن نوشته می‌شود. این فایل معمولاً برای خطایابی و دیباگ : file "data/named.run" استفاده می‌شود.

سطح شدت لاگ‌ها را مشخص می‌کند. مقدار dynamic باعث می‌شود شدت لاگ‌ها به صورت : severity dynamic خودکار بسته به نیاز تغییر کند.

3. تعریف زون Master

```
zone "jamshidi.ir" IN {
        type master;
        file "/var/named/data/jamshidi.db";
        allow-update { none; };
        allow-transfer { 192.168.69.55; };
};
```

این بخش مربوط به تعریف زون دامنه jamshidi.ir است. دامنه تعریف‌شده نوع Master است. : zone "jamshidi.ir" IN

مشخص می‌کند این زون نوع Master (سرور اصلی) است که اطلاعات معتبر دامنه را ذخیره و مدیریت : type master می‌کند.

مسیر فایل مربوط به این زون است. این فایل شامل رکوردهای DNS (مانند : file "/var/named/data/jamshidi.db" A، NS و غیره) برای دامنه jamshidi.ir است.

مشخص می‌کند که چه آدرس‌هایی اجازه دارند رکوردهای DNS را در یک زون به‌صورت : allow-update { none; } دینامیک تغییر دهند.

```
allow-update { none; };
```

یعنی هیچ‌کس اجازه تغییر ندارد. این حالت معمولاً برای امنیت بیشتر استفاده می‌شود.

مشخص می‌کند فقط آدرس IP مشخص‌شده (در اینجا: 192.168.69.55) اجازه : allow-transfer { 192.168.69.55; } دریافت اطلاعات زون را دارد. این معمولاً برای سرورهای Slave تنظیم می‌شود.

از ابزارهایی مانند **dig** یا **host** می‌توان برای Zone Transfer استفاده کرد. نمونه‌ای از دستور با **dig**:

```
dig axfr @<DNS_Server_IP> <domain_name>
```

چگونه از این مشکل جلوگیری کنیم؟

برای جلوگیری از Zone Transfer به افراد غیرمجاز:

1. در فایل تنظیمات Zone، فقط به سرورهای معتبر اجازه Zone Transfer بدهید:

```
zone "jamshidi.ir" {
    type master;
    file "/var/named/data/jamshidi.db";
    allow-transfer { 192.168.69.55; };  # فقط به سرور Slave اجازه داده شده
};
```

اگر Zone Transfer به‌درستی محدود شود، تلاش برای استخراج اطلاعات دامنه با خطا مواجه خواهد شد.
این کار برای جلوگیری از لو رفتن رکورد های dns استفاده می شود .

4. زون Root یا Hint

```
zone "." IN {
        type hint;
        file "named.ca";
};
```

zone "." :

زون `.` به دامنه Root مربوط است و برای راه‌اندازی اولیه سرور DNS استفاده می‌شود.

- **type hint**: به سرور DNS می‌گوید که به سرورهای Root مراجعه کند تا اطلاعات مربوط به دامنه‌های دیگر را پیدا کند.
- **file "named.ca"**: این فایل لیستی از آدرس‌های IP سرورهای Root DNS را دارد و به سرور کمک می‌کند درخواست‌ها را به این سرورها ارسال کند.

ساده‌تر:

این بخش به سرور DNS یاد می‌دهد از کجا شروع کند و اگر چیزی نداند، سراغ سرورهای اصلی (Root) برود.
فایل `named.ca` هم آدرس سرورهای Root را به سرور می‌دهد.

ACL

در تنظیمات DNS، از (Access Control List) **ACL** برای مدیریت دسترسی‌ها استفاده می‌کنیم. یک نمونه ساده از **ACL** برای سرور BIND می‌تواند به این شکل باشد:

```
acl "trusted" {
    192.168.1.0/24;    // Internal network
    10.0.0.0/16;       // Another private network
    localhost;         // The server itself
    127.0.0.1;         // Loopback
};
```

```
options {

allow-query { trusted; };

allow-recursion { trusted; };

allow-transfer { none; };

};
```

این بخش از پیکربندی DNS فقط به کلاینت‌های موجود در "ACL "trusted اجازه می‌دهد که پرس‌وجو (query) و درخواست‌های بازگشتی (recursion) انجام دهند و انتقال زون‌ها (zone transfers) را غیرفعال می‌کند.

DNS Zone File

$TTL 604800

@ IN SOA **ns1.jamshidi.ir. admin.jamshidi.ir.** (

5 ; Serial

604800 ; Refresh

86400 ; Retry

2419200 ; Expire

604800) ; Negative Cache TTL

jamshidi.ir. IN NS **ns1.jamshidi.ir.**
jamshidi.ir. IN NS **ns2.jamshidi.ir.**

ns1 IN A 192.168.69.54
ns2 IN A 192.168.69.54

@ IN A 192.168.5.129
www IN A 192.168.5.131

این فایل مربوط به تنظیمات **DNS Zone File** برای دامنه `jamshidi.ir` است که ساختار و اطلاعات اصلی دامنه را مشخص می‌کند. هر بخش را به‌صورت جداگانه توضیح می‌دهم:

1. $TTL 604800

این خط مقدار **Time-To-Live (TTL)** پیش‌فرض رکوردهای DNS را تنظیم می‌کند. مقدار `604800` برحسب ثانیه است (معادل 7 روز). این عدد مشخص می‌کند که اطلاعات DNS چه مدت در کش (Cache) کلاینت‌ها یا سرورهای DNS ذخیره شود.

2. SOA Record (Start of Authority)

رکورد **SOA** اطلاعات اصلی در مورد دامنه و سرور اصلی DNS را ارائه می‌دهد:

- **ns1.jamshidi.ir.**: نام سرور اصلی که مدیریت این دامنه را برعهده دارد
- **admin.jamshidi.ir.**: آدرس ایمیل مدیر دامنه (نقطه‌ها به‌جای @ هستند)
- **Serial (5)**: شماره سریال که هر بار تغییر در فایل اعمال می‌شود، باید افزایش یابد. سرورهای ثانویه از این مقدار برای تشخیص تغییرات استفاده می‌کنند
- **Refresh (604800)**: (7 روز) مدت‌زمانی که سرور ثانویه صبر می‌کند تا تنظیمات سرور اصلی را بررسی کند.
- **Retry (86400)**: (1 روز) اگر ارتباط با سرور اصلی قطع شد، بعد از این مدت تلاش مجدد انجام می‌شود.
- **Expire (2419200)**: (28 روز) مدت‌زمانی که پس از قطع ارتباط، سرور ثانویه اطلاعات را معتبر نگه می‌دارد.
- **Negative Cache TTL (604800)**: (7 روز) مدت‌زمان ذخیره پاسخ‌های منفی (یعنی عدم وجود رکورد) در کش

3. NS Records (Name Server Records)

- این بخش مشخص می‌کند که کدام سرورها به‌عنوان **Name Server (NS)** برای این دامنه عمل می‌کنند:
- `jamshidi.ir. IN NS ns1.jamshidi.ir.`
- `jamshidi.ir. IN NS ns2.jamshidi.ir.`

این دو رکورد به کاربران نشان می‌دهند که برای جستجوی اطلاعات دامنه، باید به سرور `ns1` یا `ns2` مراجعه کنند.

4. A Records (Address Records)

- **ns1 IN A 192.168.69.54**: این رکورد مشخص می‌کند که آدرس IP سرور ns1 برابر با 192.168.69.54 است.
- **ns2 IN A 192.168.69.54**: آدرس سرور ns2 هم با ns1 یکسان است.
- **@ IN A 192.168.5.129**: این رکورد نشان می‌دهد که آدرس IP اصلی دامنه jamshidi.ir برابر با 192.168.5.129 است.
- **www IN A 192.168.5.131**: زیر دامنه www.jamshidi.ir به آدرس 192.168.5.131 اشاره می‌کند.

کاربرد هر بخش به‌طور خلاصه

1. **$TTL**: مدت‌زمان کش شدن اطلاعات
2. **SOA**: مشخصات اصلی سرور و اطلاعات به‌روزرسانی DNS
3. **NS**: سرورهای DNS اصلی و ثانویه برای دامنه
4. **A Records**: تبدیل نام دامنه و زیر دامنه‌ها به آدرس IP

Jailing BIND

یک تکنیک امنیتی که به منظور محدود کردن دسترسی و عملکرد نرم‌افزار BIND به استفاده می‌شود . هدف از این کار این است که حتی اگر مهاجم توانست به سیستم شما حمله کند و سرور DNS را نفوذ کند، توانایی دسترسی به سایر بخش‌های سیستم یا سرور را نداشته باشد. به عبارت دیگر، این کار با ایجاد محدودیت‌های بیشتر در فضای کاری BIND، از آسیب‌های احتمالی جلوگیری می‌کند.

چگونه Jailing BIND کار می‌کند؟
زمانی که BIND در حالت Jail قرار دارد، برنامه BIND به یک دایرکتوری یا محیط محدود دسترسی پیدا می‌کند. در این محیط محدود، BIND تنها به فایل‌ها و منابعی که در این دایرکتوری قرار دارند دسترسی خواهد داشت و از دسترسی به سایر بخش‌های سیستم جلوگیری می‌شود.

اینکار مشابه به یک sandbox است که در آن برنامه نمی‌تواند خارج از محیطی که برایش تعریف شده است، عمل کند.

برای **Jailing BIND** یا محدود کردن آن به یک محیط ایزوله، چندین روش وجود دارد. یکی از رایج‌ترین روش‌ها استفاده از ابزارهایی مانند `chroot` برای قرار دادن BIND در یک دایرکتوری خاص است. در اینجا، به مراحل انجام این کار خواهیم پرداخت.

1. استفاده از chroot برای Jail کردن BIND

chroot (change root) یک دستور در لینوکس است که به شما این امکان را می‌دهد که محیط کاری برنامه‌ها را به یک دایرکتوری خاص محدود کنید. این به این معناست که BIND نمی‌تواند به منابع خارج از آن دایرکتوری دسترسی پیدا کند.

مرحله 1: نصب BIND و پیکربندی اولیه

اولین قدم این است که BIND را نصب و پیکربندی کنید. اگر هنوز نصب نکردید، از دستور زیر استفاده کنید:

```
sudo apt-get update
sudo apt-get install bind9
```

مرحله 2: ایجاد دایرکتوری Jail برای BIND

در این مرحله، یک دایرکتوری برای Jail کردن BIND ایجاد می‌کنید. فرض کنید می‌خواهیم آن را در `var/named/chroot/` قرار دهیم.

```
sudo mkdir -p /var/named/chroot
sudo mkdir -p /var/named/chroot/var/named
```

مرحله 3: کپی کردن فایل‌های مورد نیاز به دایرکتوری Jail

حالا باید فایل‌های مورد نیاز BIND را به داخل Jail منتقل کنید. این فایل‌ها معمولاً شامل پیکربندی‌های BIND و فایل‌های دامنه هستند.

برای مثال:

```
sudo cp -r /etc/bind /var/named/chroot/etc
sudo cp -r /var/cache/bind /var/named/chroot/var/cache
```

```
sudo cp -r /var/named /var/named/chroot/var/named
```

مرحله 4: تغییر فایل پیکربندی BIND برای استفاده از chroot

حالا باید فایل پیکربندی BIND را طوری تغییر دهید که از محیط `chroot` استفاده کند.

1. فایل پیکربندی (`/etc/bind/named.conf`) BIND را ویرایش کنید و اطمینان حاصل کنید که مسیرهای مربوط به فایل‌ها و دایرکتوری‌ها به درستی تنظیم شده‌اند. مثلا باید مسیرهایی که به `/var/named/` اشاره دارند، به `/var/named/chroot/var/named/` تغییر کنند.
2. همچنین، اگر از `systemd` برای مدیریت BIND استفاده می‌کنید، باید تنظیمات مربوط به آن را برای `chroot` در فایل سرویس BIND تغییر دهید. به طور پیش‌فرض، ممکن است این فایل در `/etc/systemd/system/bind9.service` باشد.

```
[Service]
ExecStartPre=/usr/sbin/chroot /var/named/chroot /bin/bash -c
"/usr/sbin/named -g"
ExecStart=/usr/sbin/named -g
```

مرحله 5: تغییر مالکیت و مجوزهای فایل‌ها

اطمینان حاصل کنید که BIND دسترسی لازم برای فایل‌ها و دایرکتوری‌ها در داخل Jail را دارد. به عنوان مثال:

```
sudo chown -R bind:bind /var/named/chroot
```

مرحله 6: راه‌اندازی مجدد سرویس BIND

بعد از انجام تنظیمات، باید سرویس BIND را دوباره راه‌اندازی کنید تا تغییرات اعمال شوند:

```
sudo systemctl restart bind9
```

LVM

LVM یک سیستم مدیریت دیسک در لینوکس است که به شما اجازه می‌دهد تا دیسک‌های سخت را به‌صورت منطقی مدیریت کنید و حجم‌های منطقی (Logical Volumes) را از فضای دیسک‌های فیزیکی ایجاد کنید. در واقع، LVM به شما انعطاف‌پذیری بیشتری در تخصیص، تغییر اندازه، و مدیریت فضای ذخیره‌سازی می‌دهد.

LVM شامل ۳ جزء اصلی است:

--
Physical Volume (PV)
Volume Group (VG)
Logical Volume (LV)

1. Physical Volume (PV)

Physical Volume به هر دیسک یا پارتیشنی گفته می‌شود که در LVM برای ذخیره داده‌ها استفاده می‌شود.

یک PV می‌تواند یک دیسک سخت فیزیکی (مانند `/dev/sda`) یا یک پارتیشن از دیسک (مثل `/dev/sda1`) باشد.

ایجاد PV:
برای استفاده از یک دیسک یا پارتیشن در LVM، ابتدا باید آن را به یک Physical Volume تبدیل کنیم. این کار با دستور `pvcreate` انجام می‌شود.

چرا از PV استفاده می‌شود؟
PVها به‌عنوان اجزای اصلی فضای ذخیره‌سازی در LVM عمل می‌کنند. پس از تبدیل یک دیسک یا پارتیشن به PV، می‌توان آن‌ها را به Volume Group اضافه کرد.

2. Volume Group (VG)

Volume Group مجموعه‌ای از Physical Volumeهاست که به‌طور کلی فضای ذخیره‌سازی را برای Logical Volumeها فراهم می‌کند. یک Volume Group می‌تواند شامل یک یا چند PV باشد و این امکان را فراهم می‌کند که فضای ذخیره‌سازی به‌طور انعطاف‌پذیر مدیریت شود.

ایجاد VG:

بعد از اینکه PVها را آماده کردید، باید آن‌ها را به یک Volume Group اضافه کنید. این کار با دستور `vgcreate` انجام می‌شود.

چرا از VG استفاده می‌شود؟

Volume Group به شما این امکان را می‌دهد که فضای ذخیره‌سازی را به‌طور منطقی و بدون وابستگی به دیسک‌های فیزیکی مدیریت کنید. با افزودن چند PV به یک VG، می‌توانید فضای ذخیره‌سازی بیشتری به‌صورت یکپارچه داشته باشید.

3. Logical Volume (LV)

Logical Volume بخشی از فضای ذخیره‌سازی است که از یک یا چند **Volume Group** استخراج می‌شود. هر Logical Volume به‌طور منطقی مانند یک پارتیشن یا دیسک رفتار می‌کند، اما برخلاف پارتیشن‌های سنتی، می‌تواند تغییر اندازه دهد و فضای بیشتری به آن افزوده شود.

ایجاد LV:
بعد از اینکه یک VG ساخته شد، می‌توانید Logical Volumeها را از آن ایجاد کنید. این کار با دستور `lvcreate` انجام می‌شود.

چرا از LV استفاده می‌شود؟

LVها امکان ایجاد و مدیریت پارتیشن‌های منطقی را با انعطاف‌پذیری بیشتر فراهم می‌کنند. می‌توان آن‌ها را در هر زمانی گسترش داد، کوچک کرد یا حتی حذف کرد. همچنین، می‌توان از آن‌ها برای نصب سیستم‌عامل‌ها، داده‌ها یا فایل‌سیستم‌ها استفاده کرد

+

Physical Extents (PE)

Physical Extent (PE) واحدهای کوچک و ثابت اندازه‌ای هستند که در سطح Physical Volume (PV) به‌کار می‌روند. این‌ها بخش‌هایی از فضای ذخیره‌سازی یک PV هستند که توسط Volume Group (VG) برای تخصیص فضای ذخیره‌سازی به Logical Volumes (LV) استفاده می‌شوند.

هر PE معمولاً به اندازه ۴ مگابایت یا ۸ مگابایت است (می‌توانید این اندازه را در هنگام ایجاد VG تعیین کنید، ولی معمولاً پیش‌فرض ۴ مگابایت است).

PE به‌عنوان واحد تخصیص فضای ذخیره‌سازی عمل می‌کند. زمانی که شما Logical Volume (LV) ایجاد می‌کنید، فضای آن از PEهای موجود در Volume Group گرفته می‌شود.

دستورات کلیدی LVM:

- `pvcreate` : برای تبدیل دیسک یا پارتیشن به یک Physical Volume
- `vgcreate` : برای ایجاد یک Volume Group جدید از یک یا چند PV
- `lvcreate` : برای ایجاد یک Volume Group جدید از یک یا چند PV
- `lvextend` : برای گسترش یک Logical Volume
- `lvreduce` : برای کوچک کردن حجم یک Logical Volume
- `vgextend` : برای افزودن PV جدید به Volume Group.
- `vgreduce` : برای حذف PV از یک VG
- `pvremove` : برای حذف یک PV از LVM
- `lvremove` : برای حذف یک LV
- `vgremove` : برای حذف یک VG

نمونه‌ای از نحوه کار LVM:

فرض کنید یک سرور با دو دیسک داریم:

1. `/dev/sdb`
2. `/dev/sdc`

1. شناسایی دیسک‌های جدید

ابتدا دیسک‌ها و پارتیشن‌های موجود را شناسایی کنید:

```
lsblk
lsblk -f
```

اگر دیسک‌ها شناسایی نشده‌اند، دستور زیر را برای اسکن مجدد اجرا کنید:

`

```
echo "- - -" > /sys/class/scsi_host/host0/scan
echo "- - -" > /sys/class/scsi_host/host1/scan
echo "- - -" > /sys/class/scsi_host/host2/scan
```

بعد از اسکن، دوباره دیسک‌ها را بررسی کنید:

```
lsblk -f
```

2. تبدیل دیسک‌ها به Physical Volume

دیسک‌های جدید را به Physical Volume (PV) تبدیل کنید. فرض کنیم دیسک‌های /dev/sdb و /dev/sdc داریم:

```
pvcreate /dev/sdb /dev/sdc
```

3. ایجاد Volume Group

حالا این Physical Volumeها را به یک **Volume Group (VG)** اضافه کنید. مثلاً نام VG را **data** می‌گذاریم:

```
vgcreate data /dev/sdb /dev/sdc
```

4. ایجاد Logical Volume

در این مرحله، یک **Logical Volume (LV)** می‌سازیم. فرض کنیم می‌خواهیم یک LV به نام **lvdata** با حجم 10 گیگابایت ایجاد کنیم:

```
lvcreate -L 10G -n lvdata data
```

5. فرمت کردن Logical Volume

باید LV جدید را فرمت کنید تا قابل استفاده باشد. برای این کار، از فایل سیستم مورد نظر استفاده می‌کنیم:

یا برای سیستم فایل **ext4**:

```
mkfs.ext4 /dev/data/lvdata
```

یا برای سیستم فایل **xfs**:

```
mkfs.xfs /dev/data/lvdata
```

6. مونت (mount) کردن Logical Volume

ابتدا یک دایرکتوری برای mount کردن LV می‌سازیم:

```
mkdir /mnt/lvdata
```

سپس LV را به این مسیر mount می‌کنیم:

```
mount /dev/data/lvdata /mnt/lvdata
```

بررسی تغییرات:
در پایان، با استفاده از دستور زیر مطمئن شوید فضای جدید اعمال شده است:

```
df –h /mnt/lvdata
```

بیشتر کردن فضا (Extend)

1. افزایش اندازه Logical Volume

فرض کنیم می‌خواهید حجم LV را 5 گیگابایت افزایش دهید:

```
lvextend –L +5G /dev/data/lvdata
```

گسترش (Grow) :

با ساختLV بنظر کار به اتمام می رسد . ما فضای Logical Volume را افزایش دادیم، اما سیستم فایل هنوز از این فضای اضافی استفاده نمی‌کند . مانند یک اتاقی که فضای ان را افزایش دادیم ولی صندلی نداریم که بتوان از فضا اضافه شده استفاده کرد .

برای سیستم فایل‌های **ext4**:

```
resize2fs /dev/data/lvdata
```

برای سیستم فایل‌های **xfs**:

```
xfs_growfs /mnt/lvdata
```

بررسی فضای جدید پس از گسترش
پس از گسترش سیستم فایل، فضای جدید را بررسی کنید تا مطمئن شوید اعمال شده است:

```
df –h
```

کم کردن فضا (Reduce)

برای کاهش فضای یک **Logical Volume** در LVM، باید با دقت عمل کنید. این فرآیند شامل مراحل زیر است:

1. اطمینان از استفاده نشدن از فضای Logical Volume

- ابتدا مطمئن شوید که **Logical Volume** در حال استفاده نیست یا داده‌ای که نمی‌خواهید از بین برود، در آن ذخیره نشده است.
- اگر این LV در حال استفاده است، ابتدا آن را **unmount** کنید:

```
umount /mnt/lvdata
```

2. بررسی سلامت سیستم فایل

قبل از کاهش حجم، سیستم فایل باید بررسی و سالم باشد. برای این کار:

```
e2fsck -f /dev/data/lvdata
```

این دستور برای فایل‌سیستم‌های **ext4** است. اگر از **xfs** استفاده می‌کنید، این مرحله نیاز نیست.

3. کاهش اندازه سیستم فایل

ابتدا اندازه سیستم فایل را به مقدار مورد نظر کاهش دهید:

```
resize2fs /dev/data/lvdata 5G
```

(در اینجا حجم جدید را **5 گیگابایت** در نظر گرفته‌ایم.)

⚠️ توجه: حجم جدید باید کمتر یا مساوی حجم نهایی LV باشد.

4. کاهش اندازه Logical Volume

حالا اندازه LV را کاهش دهید:

```
lvreduce -L 5G /dev/data/lvdata
```

(در اینجا حجم جدید LV را **5 گیگابایت** قرار داده‌ایم.)

5. مونت مجدد Logical Volume

بعد از کاهش حجم، LV را دوباره به مسیر مشخص **mount** کنید:

```
mount /dev/data/lvdata /mnt/lvdata
```

6. بررسی فضای نهایی

برای اطمینان از کاهش موفقیت‌آمیز:

```
df -h /mnt/lvdata
```

پشتیبان‌گیری از داده‌ها: کاهش اندازه می‌تواند باعث از دست رفتن داده‌های خارج از محدوده جدید شود. حتماً قبل از انجام این مراحل، از داده‌های خود پشتیبان بگیرید.

NFS

برای راه‌اندازی سرویس **NFS** (Network File System) بر روی یک سیستم لینوکس، ابتدا باید چند مرحله را انجام دهیم که شامل نصب بسته‌های مورد نیاز، تنظیمات لازم برای سرور و کلاینت و سپس تست کردن اتصال بین آن‌ها می‌شود.

1. نصب بسته‌های NFS

برای نصب NFS سرور، دستور زیر را وارد کنید:

```
sudo yum install nfs-utils -y
```

این دستور بسته‌های لازم برای NFS را نصب می‌کند.

2. تنظیمات فایل‌های پیکربندی

برای اینکه دایرکتوری‌ها را به اشتراک بگذارید، باید فایل پیکربندی `/etc/exports` را ویرایش کنید.

ابتدا با دستور `vi` یا `vim` فایل `/etc/exports` را باز کنید:

```
sudo vi /etc/exports
```

سپس دایرکتوری‌ای که می‌خواهید به اشتراک بگذارید را اضافه کنید. برای مثال:

```
/opt/tomcat 192.168.69.0/24(rw,sync,no_root_squash)
```

در فایل پیکربندی (`/etc/exports`) NFS، وقتی یک دایرکتوری را به اشتراک می‌گذارید، می‌توانید پارامترهای مختلفی را برای کنترل دسترسی و نحوه رفتار NFS تنظیم کنید. پارامترهایی مانند `rw` , `sync` , `no_root_squash` که در مثال آمده است، نقش مهمی در تنظیمات امنیتی و دسترسی دارند. در اینجا توضیح می‌دهیم که هرکدام چه کاری انجام می‌دهند:

1. rw (Read-Write)

این گزینه به کلاینت‌ها اجازه می‌دهد که به دایرکتوری اشتراکی هم دسترسی خواندن (Read) و هم نوشتن (Write) داشته باشند.

- **rw**: کلاینت می‌تواند فایل‌ها را هم بخواند و هم تغییر دهد.
- **ro**: اگر از ro استفاده کنید، فقط دسترسی خواندن به کلاینت‌ها داده می‌شود و نمی‌توانند فایل‌ها را تغییر دهند.

2. sync(Synchronous)

پارامتر `sync` مشخص می‌کند که عملیات ورودی/خروجی (I/O) به صورت همزمان انجام شود. این به این معنی است که هر عملیات نوشتن باید تکمیل شود و داده‌ها به دیسک نوشته شوند قبل از اینکه به درخواست بعدی پرداخته شود.

- **sync**: هر عملیات نوشتن منتظر می‌ماند تا داده‌ها به دیسک نوشته شوند و سپس پاسخ به کلاینت داده می‌شود. این باعث اطمینان از پایداری داده‌ها و کاهش ریسک از دست دادن داده‌ها در صورت خاموش شدن غیرمنتظره سیستم می‌شود.
- **async**: اگر از async استفاده کنید، داده‌ها به صورت ناهمزمان نوشته می‌شوند و در نتیجه سرعت دسترسی به فایل‌ها ممکن است بالاتر باشد، اما خطر از دست دادن داده‌ها در صورت وقوع خرابی افزایش می‌یابد.

3. no_root_squash

این گزینه تاثیر زیادی بر رفتار سیستم در زمان استفاده از کاربر root دارد. در NFS، وقتی کلاینت‌ها به سرور متصل می‌شوند، NFS به طور پیش‌فرض کاربران root را به کاربری معمولی تبدیل می‌کند تا از خطرات امنیتی جلوگیری کند. این کار به نام "root squashing" شناخته می‌شود.

- **no_root_squash**: این گزینه باعث می‌شود که کاربر root در کلاینت‌ها همچنان دسترسی‌های root خود را داشته باشد، به این معنی که هیچ تبدیل یا "squash"ای برای کاربر root انجام نمی‌شود.
- **root_squash**: اگر از این گزینه استفاده کنید، کاربر root در کلاینت‌ها به یک کاربر معمولی تبدیل می‌شود (معمولاً nfsnobody)، و بنابراین دسترسی‌های ریشه‌ای از بین می‌روند.

مهم: استفاده از no_root_squash می‌تواند خطر امنیتی داشته باشد، زیرا ممکن است یک کلاینت با دسترسی `root` توانایی انجام عملیات‌های خطرناک روی سرور را داشته باشد. این گزینه معمولاً در محیط‌هایی استفاده می‌شود که کنترل امنیتی بالایی دارند یا نیاز به دسترسی‌های خاصی از سوی root دارند.

جمع‌بندی:

- **rw**: اجازه خواندن و نوشتن به کلاینت‌ها.
- **sync**: اطمینان از نوشتن داده‌ها به دیسک قبل از پاسخ به کلاینت.
- **no_root_squash**: حفظ دسترسی‌های root از طرف کلاینت‌ها (باید با احتیاط استفاده شود).

به طور کلی، شما باید این پارامترها را بسته به نیاز و سطح امنیتی شبکه و سیستم خود تنظیم کنید.

3. راه‌اندازی و فعال‌سازی سرویس NFS

بعد از تنظیمات فایل `etc/exports/`، باید سرویس NFS را راه‌اندازی و فعال کنیم:

```
sudo systemctl start nfs-server
sudo systemctl enable nfs-server
```

برای اطمینان از اینکه سرویس در حال اجرا است، از دستور زیر استفاده کنید:

```
sudo systemctl status nfs-server
```

4. تنظیم فایروال

اگر فایروال فعال باشد، باید پورت‌های NFS را باز کنید. می‌توانید دستور زیر را برای باز کردن پورت‌ها وارد کنید:

```
sudo firewall-cmd --permanent --add-service=nfs
sudo firewall-cmd --permanent --add-service=mountd
sudo firewall-cmd --permanent --add-service=rpc-bind
sudo firewall-cmd --reload
```

5. تنظیمات کلاینت

برای دسترسی به دایرکتوری به اشتراک گذاشته شده از سمت کلاینت، ابتدا باید بسته‌های NFS را نصب کنید:

```
sudo yum install nfs-utils -y
```

سپس دایرکتوری مورد نظر را از سرور مونت کنید:

```
sudo mount -t nfs 192.168.69.54:/opt/tomcat /mnt
```

برای اینکه دایرکتوری به صورت دائمی مانت شود، باید فایل `etc/fstab/` را ویرایش کنید و خط زیر را اضافه کنید:

```
192.168.69.54:/opt/tomcat /mnt nfs defaults 0 0
```

6. تست اتصال

برای اطمینان از عملکرد درست، می‌توانید دستور زیر را برای مشاهده دایرکتوری‌های مانت شده وارد کنید:

```
df -h
```

همچنین می‌توانید از دستور `ls` برای بررسی محتویات دایرکتوری مانت شده استفاده کنید:

```
ls /mnt
```

این مراحل باید سرویس NFS را بر روی سیستم شما راه‌اندازی کرده و اتصال کلاینت به سرور NFS را برقرار کند.

TTFB

TTFB (Time to First Byte) یک معیار کلیدی در عملکرد وب‌سایت‌ها و سرورهاست که نشان می‌دهد چه مدت طول می‌کشد تا اولین بایت از داده از سرور به مرورگر کاربر برسد

این زمان شامل ۳ مرحله اصلی است:

1. **DNS Lookup** : مدت‌زمان لازم برای یافتن آدرس سرور و برقراری اتصال TCP یا TLS (در صورت استفاده از HTTPS).
2. **ارسال درخواست:** زمانی که مرورگر درخواست HTTP/HTTPS را به سرور ارسال می‌کند.
3. **دریافت اولین پاسخ سرور:** زمانی که سرور شروع به ارسال اولین داده به مرورگر می‌کند.

مثال :

TTFB = 200ms : به این معنی است که از لحظه‌ای که مرورگر درخواست را به سرور ارسال می‌کند تا لحظه‌ای که اولین بایت داده دریافت می‌شود 200 میلی‌ثانیه طول کشیده است.

چرا مهم است؟

- زمان بالای TTFB می‌تواند نشان‌دهنده مشکلاتی مانند:
  - کندی سرور
  - مشکلات شبکه
  - زمان پردازش طولانی برای درخواست‌ها
- زمان پایین TTFB به بهبود تجربه کاربری و سرعت بارگذاری وب‌سایت کمک می‌کند.

چطور TTFB را بهبود دهیم؟

1. **استفاده از CDN** برای کاهش تأخیر شبکه
2. **بهینه‌سازی کدهای سرور و پایگاه داده**
3. **فعال کردن کشینگ (Caching)** در سمت سرور
4. **بهبود کانفیگ وب‌سرور (مثل NGINX یا Apache)**

FIREWALL

مفاهیم کلی : این مفاهیم متعلق به iptables یا فایروال لینوکس هستند و برای مدیریت ترافیک شبکه استفاده می‌شوند. به طور کلی، iptables ترافیک شبکه را بر اساس زنجیره‌ها (chains) و جداول (tables) مدیریت می‌کند. در ادامه، هر یک از موارد ذکرشده توضیح داده شده‌اند:

Chains (زنجیره‌ها):

1. PREROUTING

- این زنجیره قبل از اینکه ترافیک به شبکه داخلی یا پردازش‌های سیستم برسد، اعمال می‌شود.
- کاربرد: معمولاً برای تغییر مسیر (DNAT) یا تغییر بسته‌ها قبل از رسیدن به مقصد استفاده می‌شود.

2. INPUT

- ترافیکی که وارد سیستم یا سرور می‌شود و مقصد آن خود سرور است، از این زنجیره عبور می‌کند.
- کاربرد: برای کنترل دسترسی به سرویس‌ها یا پورت‌های سرور (مثلاً بستن یا باز کردن پورت‌ها).

3. OUTPUT

- ترافیکی که از سیستم خارج می‌شود و توسط خود سیستم تولید شده است، از این زنجیره عبور می‌کند.
- کاربرد: کنترل ترافیکی که از سرور به مقصدهای دیگر ارسال می‌شود.

4. POSTROUTING

- این زنجیره بعد از مسیریابی ترافیک و درست قبل از خروج از دستگاه اعمال می‌شود.
- کاربرد: معمولاً برای تغییر آدرس مبدأ (SNAT) استفاده می‌شود.

5. FORWARD

- ترافیکی که از سیستم عبور می‌کند ولی مقصد آن سیستم نیست، از این زنجیره عبور می‌کند.
- کاربرد: در سیستم‌های مسیریاب (router) برای انتقال ترافیک بین شبکه‌ها.

Tables (جداول):

1. Filter Table

- این جدول پیش‌فرض برای فیلتر کردن و مدیریت ترافیک استفاده می‌شود.
- زنجیره‌های مرتبط: `INPUT`, `OUTPUT`, `FORWARD`.
- کاربرد:
  - اجازه یا جلوگیری از دسترسی به ترافیک خاص.
  - بستن یا بازکردن پورت‌ها.

2. NAT Table

- این جدول برای ترجمه آدرس شبکه (Network Address Translation) استفاده می‌شود.
- زنجیره‌های مرتبط: `PREROUTING`, `POSTROUTING`, `OUTPUT`.
- کاربرد:

- **DNAT**: تغییر آدرس مقصد بسته‌ها (مثلاً برای Port Forwarding).
- **SNAT**: تغییر آدرس مبدأ بسته‌ها (مثلاً برای اینترنت به اشتراک گذاشته شده).

3. Mangle Table

- این جدول برای تغییر ویژگی‌های بسته‌های شبکه استفاده می‌شود.
- زنجیره‌های مرتبط: همه زنجیره‌ها (`PREROUTING`, `INPUT`, `FORWARD`, `OUTPUT`, `POSTROUTING`).
- کاربرد:
  - تغییر TTL (Time to Live).
  - علامت‌گذاری (marking) بسته‌ها برای پردازش‌های بعدی.

مثال کاربردی:

1. **PREROUTING**: اگر بخواهید ترافیک ورودی به یک آدرس IP مشخص را به آدرس دیگری بفرستید (DNAT)

2. **INPUT**: اگر بخواهید فقط به درخواست‌های SSH از یک محدوده IP اجازه دسترسی بدهید
3. **OUTPUT**: محدود کردن دسترسی سرور به اینترنت برای پروتکل یا مقصد خاص
4. **POSTROUTING**: استفاده از SNAT برای تغییر آدرس مبدأ بسته‌ها در اینترنت
5. **FORWARD**: تنظیم مسیریابی بین دو شبکه داخلی

خلاصه:

- **Chains** مشخص می‌کنند که ترافیک در کدام مرحله بررسی می‌شود.
- **Tables** (فیلتر، NAT، یا تغییرات پیشرفته) تعیین می‌کنند که چه کاری باید روی بسته انجام شود.

1. iptables

ابزاری قدیمی‌تر برای مدیریت فایروال در لینوکس است که مبتنی بر Netfilter کار می‌کند.

از iptables برای تعریف قوانینی (Rules) استفاده می‌شود که ترافیک شبکه را در سطح بسته (Packet) کنترل می‌کنند.

می‌توان قوانین را برای فیلتر کردن بسته‌ها، تغییر مسیر بسته‌ها، یا تنظیم NAT تعیین کرد.

نقاط قوت:

- بسیار قدرتمند و قابل تنظیم.
- امکان اعمال کنترل دقیق روی بسته‌های شبکه.

نقاط ضعف:

- پیچیدگی در مدیریت قوانین زیاد.
- مناسب نبودن برای محیط‌های پویا که تغییرات سریع نیاز دارند.

معماری:

- از **Chain**ها و **Table**ها برای کنترل بسته‌ها استفاده می‌کند.
- جدول‌ها شامل:

- **Filter Table**: برای فیلتر کردن بسته‌ها
- **NAT Table**: برای ترجمه آدرس شبکه
- **Mangle Table**: برای تغییر بسته‌های شبکه

Chain در iptables چیست؟

Chainها مانند لیست‌هایی از دستورات هستند که مشخص می‌کنند بسته‌های شبکه چگونه پردازش شوند. اگر بسته‌ای وارد یک Chain شود:

1. قوانین به ترتیب بررسی می‌شوند.
2. اولین قانونی که با بسته مطابقت داشته باشد اجرا می‌شود.

3. اگر هیچ قانونی مطابقت نداشته باشد، تصمیم نهایی به **Policy پیش‌فرض** Chain (مانند ACCEPT یا DROP) واگذار می‌شود.

iptables: های پیش‌فرض در Chain

1. **INPUT**: برای پردازش بسته‌های ورودی (Inbound Traffic)
2. **OUTPUT**: برای پردازش بسته‌های خروجی (Outbound Traffic)
3. **FORWARD**: برای بسته‌هایی که از سیستم عبور می‌کنند (مانند روتر)

زیر هم بودن قوانین در Chain:

زمانی که قوانین زیادی در یک Chain دارید:

1. **به ترتیب پردازش می‌شوند**:
   - هر بسته از ابتدای لیست شروع شده و تک‌تک قوانین را بررسی می‌کند.
   - اگر قانون شماره 50 با بسته مطابقت داشته باشد، بسته باید از 49 قانون قبلی عبور کند.
2. **تأثیر بر عملکرد**:
   - هرچه تعداد قوانین بیشتر باشد، زمان پردازش هر بسته افزایش می‌یابد.
   - این موضوع می‌تواند به تأخیر در پردازش بسته‌ها و کاهش کارایی شبکه منجر شود.
3. **قابلیت مدیریت پایین**:
   - با افزایش تعداد قوانین، مدیریت، عیب‌یابی، و تغییر قوانین دشوارتر می‌شود.
   - اشتباهات انسانی بیشتر رخ می‌دهد، مثلاً قرار دادن یک قانون در جای نادرست می‌تواند کل سیستم را مختل کند.

مشکلات زیاد بودن قوانین:

1. **کاهش عملکرد (Performance)**:
   - هر بسته باید تمام قوانین را یک‌به‌یک بررسی کند تا مطابقت پیدا کند.
   - این تأخیر در سیستم‌هایی با حجم بالای ترافیک (High Traffic) مشهودتر است.
2. **پیچیدگی در عیب‌یابی**:
   - پیدا کردن قانونی که یک مشکل خاص را ایجاد می‌کند دشوار است.
   - تغییر یک قانون ممکن است به قوانین دیگر آسیب برساند.
3. **خطاهای منطقی**:
   - اگر قوانین درست به ترتیب قرار نگیرند، ممکن است قوانین بعدی هرگز اجرا نشوند.
   - مثلاً اگر در ابتدای Chain قانونی نوشته شود که همه ترافیک را بپذیرد (ACCEPT)، قوانین بعدی بی‌اثر خواهند شد.
4. **مشکل در مقیاس‌پذیری**:
   - وقتی تعداد قوانین افزایش یابد، نگهداری و گسترش سیستم سخت‌تر می‌شود.
   - در محیط‌های بزرگ، مثل Data Centerها، این مشکل باعث محدودیت در طراحی شبکه می‌شود.

همچنین این امکان وجود دارد که تمام **rule** ها را به‌صورت یکجا در یک فایل ذخیره کرد و سپس این فایل را به iptables اعمال کرد. این روش به شما اجازه می‌دهد قوانین را بهتر مدیریت و تغییر دهید. همچنین باعث می‌شود که نیازی نباشد هر قانون را به‌صورت جداگانه اجرا کنید. این کار به‌خصوص در محیط‌های بزرگ و پیچیده بسیار مفید است.

ذخیره تمام Ruleها در یک فایل

برای ذخیره تمام قوانین فعلی iptables در یک فایل، می‌توانید از دستور زیر استفاده کنید:

```
iptables-save > /etc/iptables/rules.v4
```

بارگذاری قوانین از فایل

برای اعمال مجدد قوانین ذخیره‌شده، از دستور زیر استفاده کنید:

```
iptables-restore < /etc/iptables/rules.v4
```

2. firewalld

1. **ابزاری مدرن‌تر** برای مدیریت فایروال است که روی iptables ساخته شده و کار با فایروال را ساده‌تر می‌کند.
2. **دایمون (Daemon)** که توسط firewalld استفاده می‌شود، امکان مدیریت قوانین فایروال را به‌صورت پویا و بدون نیاز به ری‌استارت فراهم می‌کند.

مزایا

1. **رابط کاربری ساده‌تر** نسبت به iptables ارائه می‌دهد.
2. **ناحیه‌های فایروال (Zones)** را برای دسته‌بندی قوانین بر اساس نوع شبکه (مانند Home، Public، Work و...) فراهم می‌کند.
3. **امکان تغییر قوانین** به‌صورت پویا وجود دارد، بدون اینکه اتصال شبکه مختل شود.

معایب

1. **برای محیط‌های پیچیده** یا بسیار حساس، ممکن است ویژگی‌های محدودی داشته باشد.

ویژگی‌ها

1. **به‌صورت پیش‌فرض** در توزیع‌های مدرن لینوکس مانند RHEL و Fedora استفاده می‌شود.
2. **سازگاری با iptables** را نیز ارائه می‌دهد.

3. nftables

3. **توضیحات:**

nftables به‌عنوان جایگزین مدرن برای iptables و ip6tables معرفی شده و به‌طور مستقیم توسط Netfilter پشتیبانی می‌شود. هدف اصلی nftables، ساده‌سازی قوانین شبکه و افزایش کارایی است.

مزایا

1. **استفاده از یک هسته واحد** برای مدیریت ipv4 و ipv6.
2. **کاهش پیچیدگی قوانین:** در مقایسه با iptables، قوانین ساده‌تر و واضح‌تر نوشته می‌شوند.
3. **عملکرد بالا:** نیاز به بررسی مکرر جدول‌ها کاهش یافته و سرعت عملکرد بهبود می‌یابد.
4. **پشتیبانی از حالت‌های پویا** و انعطاف‌پذیری بیشتر.

معایب

1. **هنوز به اندازه iptables** در برخی محیط‌های سنتی محبوب نیست.
2. **یادگیری آن زمان‌بر** ممکن است برای افرادی که به iptables عادت دارند، چالش‌برانگیز باشد.

ویژگی‌ها

1. **در سیستم‌های جدید لینوکسی** (مانند Ubuntu 20.04 یا RHEL 8) به‌عنوان جایگزین پیش‌فرض iptables استفاده می‌شود.
2. **امکان تعریف قوانین** با Syntax ساده‌تر و خوانا تر فراهم شده است.

"Configuring Firewall Rules with iptables"

```
# Flush previous rules
iptables -F

# Set default policies to DROP (default action for inbound, outbound, and
forwarded packets)

iptables -P INPUT DROP / ACCEPT
iptables -P OUTPUT DROP / ACCEPT
iptables -P FORWARD DROP / ACCEPT

# Allow incoming SSH on port 22
iptables -A INPUT -p tcp --dport 22 -j ACCEPT

# Allow incoming HTTP traffic on port 80
iptables -A INPUT -p tcp --dport 80 -j ACCEPT

# Allow incoming HTTPS traffic on port 443
iptables -A INPUT -p tcp --dport 443 -j ACCEPT

# Allow incoming ping (ICMP) requests
iptables -A INPUT -p icmp --icmp-type echo-request -j ACCEPT

# Drop all incoming traffic from a specific IP address (e.g.,
192.168.1.100)
```

```
iptables -A INPUT -s 192.168.1.100 -j DROP
```

- `-A` **(Append)**: اضافه کردن قانون به انتهای زنجیره.
  - مثال: `iptables -A INPUT -p tcp --dport 80 -j ACCEPT`
- `-I` **(Insert)**: وارد کردن قانون در ابتدای زنجیره (یا در موقعیت مشخص).
  - مثال: `iptables -I INPUT 1 -p tcp --dport 80 -j ACCEPT`
  - `1` مشخص‌کننده موقعیت در زنجیره است.
- `-D` **(Delete)**: حذف یک قانون از زنجیره.
  - مثال: `iptables -D INPUT -p tcp --dport 80 -j ACCEPT`
- `-F` **(Flush)**: پاک کردن تمام قوانین از زنجیره.
  - مثال: `iptables -F`
- `-P` **(Policy)**: تنظیم سیاست پیش‌فرض برای زنجیره.
  - مثال: `iptables -P INPUT DROP`
- `-L` **(List)**: نمایش قوانین موجود در زنجیره.
  - مثال: `iptables -L`
- `-T` **(Table)**: مشخص کردن جدول (برای مثال `filter` یا `nat`).
  - مثال: `iptables -t nat -L`
- `-N` **(New Chain)**: ایجاد یک زنجیره جدید.
  - مثال: `iptables -N MYCHAIN`
- `-X` **(Delete Chain)**: حذف یک زنجیره تعریف‌شده توسط کاربر.
  - مثال: `iptables -X MYCHAIN`
- `-E` **(Rename Chain)**: تغییر نام یک زنجیره.
  - مثال: `iptables -E MYCHAIN NEWCHAIN`
- `-s` **(Source)**: تعیین آدرس IP مبدا برای قانون.
  - مثال: `iptables -A INPUT -s 192.168.1.100 -j DROP`
- `-d` **(Destination)**: تعیین آدرس IP مقصد برای قانون.
  - مثال: `iptables -A OUTPUT -d 192.168.1.200 -j ACCEPT`
- `-p` **(Protocol)**: تعیین پروتکل (TCP، UDP، ICMP و غیره).
  - مثال: `iptables -A INPUT -p tcp --dport 22 -j ACCEPT`
- `--dport` / `--sport` **(Destination Port / Source Port)**: تعیین پورت مقصد یا مبدا.
  - مثال: `iptables -A INPUT -p tcp --dport 80 -j ACCEPT`
- `-j` **(Jump)**: تعریف اکشن برای بسته‌ها (مانند ACCEPT، DROP، REJECT).
  - مثال: `iptables -A INPUT -p tcp --dport 80 -j ACCEPT`
- `-m` **(Match)**: استفاده از ماژول‌های اضافی برای تطابق دقیق‌تر.
  - مثال: `iptables -A INPUT -m state --state NEW -j ACCEPT`
- `--state` : تنظیم وضعیت اتصال (مثلاً NEW، ESTABLISHED).
  - مثال: `iptables -A INPUT -m state --state NEW -j ACCEPT`
- `-v` **(Verbose)**: نمایش اطلاعات بیشتر در مورد قوانین.

- مثال: `iptables -L -v`

- **`-h` (Help)**: نمایش صفحه راهنما.

- مثال: `iptables -h`

- `INPUT DROP` : تمام ترافیک ورودی به‌طور پیش‌فرض مسدود می‌شود
- `INPUT ACCEPT` : تمام ترافیک ورودی به‌طور پیش‌فرض مجاز می‌شود
- `OUTPUT ACCEPT` : تمام ترافیک خروجی به‌طور پیش‌فرض مجاز است
- `OUTPUT DROP` : تمام ترافیک خروجی به‌طور پیش‌فرض مسدود می شود
- `FORWARD DROP` : تمام ترافیک عبوری از سیستم مسدود می‌شود

```
iptables -F
```

این دستور تمام قوانین قبلی فایروال را از جدول `iptables` پاک می‌کند.

```
iptables -P INPUT DROP / ACCEPT
iptables -P OUTPUT DROP / ACCEPT
iptables -P FORWARD DROP / ACCEPT
```

این دستورات سیاست‌های پیش‌فرض فایروال را تنظیم می‌کنند:

```
iptables -A INPUT -p tcp --dport 22 -j ACCEPT
```

این دستور به ترافیک ورودی با پروتکل TCP روی پورت 22 (SSH) اجازه عبور می‌دهد. پورت 22 برای اتصال به سرور از طریق پروتکل SSH استفاده می‌شود.

```
iptables -A INPUT -p tcp --dport 80 -j DROP
```

این دستور ترافیک ورودی برای پروتکل TCP روی پورت 80 (HTTP) را مسدود می کند . این پورت برای دسترسی به وب‌سایت‌ها از طریق HTTP استفاده می‌شود.

```
iptables -A INPUT -p tcp --dport 443 -j ACCEPT
```

این دستور به ترافیک ورودی برای پروتکل TCP روی پورت 443 (HTTPS) اجازه می‌دهد. پورت 443 برای اتصال امن به وب‌سایت‌ها استفاده می‌شود.

```
iptables -A INPUT -p icmp --icmp-type echo-request -j ACCEPT
```

این دستور اجازه ارسال درخواست‌های پینگ را می دهد .

```
iptables -A OUTPUT -p icmp --icmp-type echo-reply -j DROP
```

این دستور تمام درخواست‌های **پاسخ پینگ (Echo Reply)** که به سرور شما می‌آیند را مسدود کند .

```
iptables -A INPUT -s 192.168.1.100 -j DROP
```

این دستور ترافیک ورودی از یک آدرس IP خاص (در اینجا 192.168.1.100) را مسدود می‌کند.

در فایروال‌ها و به‌ویژه در **iptables**، دستورات **REJECT** و **DROP** هر دو برای مسدود کردن ترافیک استفاده می‌شوند، اما تفاوت‌های مهمی در نحوه رفتار آن‌ها وجود دارد. این تفاوت‌ها بر نحوه مدیریت و پاسخ‌دهی به بسته‌ها تأثیر می‌گذارد.

1. DROP:

- زمانی که یک بسته با دستور **DROP** مسدود می‌شود، هیچ پاسخی به فرستنده ارسال نمی‌شود.
- بسته به‌سادگی از بین می‌رود و هیچگونه اطلاع‌رسانی به فرستنده مبنی بر مسدود شدن بسته وجود ندارد.
- این رفتار باعث می‌شود که فرستنده نمی‌تواند متوجه شود که بسته‌ای که ارسال کرده به‌طور عمدی مسدود شده یا به‌دلیل سایر مشکلات (مانند خطا در مسیر) از دست رفته است.

مزایا:

- باعث کاهش بار سرور یا شبکه می‌شود، زیرا هیچ اطلاعات اضافی (برای مسدود کردن بسته) به فرستنده ارسال نمی‌شود.
- این معمولاً برای امنیت بیشتر به‌کار می‌رود، زیرا مهاجم ممکن است نتواند تشخیص دهد که بسته‌ها عمداً مسدود می‌شوند یا به‌دلیل مشکلات دیگر از دست رفته‌اند.

2. REJECT:

- زمانی که یک بسته با دستور **REJECT** مسدود می‌شود، سیستم یک پاسخ به فرستنده ارسال می‌کند که به آن اطلاع می‌دهد که بسته مسدود شده است.
- بسته‌ها به‌طور مستقیم رد می‌شوند و یک پیام خطا (مثلاً "ICMP "Destination Unreachable یا TCP "Connection Refused"") به فرستنده ارسال می‌شود.
- این پیام به فرستنده اطلاع می‌دهد که مقصد دسترس‌پذیر نیست یا اتصال رد شده است.

مزایا:

- این رفتار برای جلوگیری از سردرگمی فرستندگان مفید است، زیرا فرستنده می‌تواند متوجه شود که بسته به‌طور عمدی مسدود شده است.
- در برخی موارد، این می‌تواند برای شبکه‌های مدیریتی مفید باشد تا مطمئن شوند که دیگر کاربران یا سیستم‌ها خطای مسیریابی دریافت نمی‌کنند و علت مسدود شدن بسته‌ها مشخص است.

ICMP (ECHO)

در پروتکل ICMP (Internet Control Message Protocol)، پیام‌های **Echo** و **Echo Reply** برای تست اتصال شبکه و بررسی تاخیر استفاده می‌شوند. این پیام‌ها معمولاً در دستور `ping` مشاهده می‌شوند.

ساختار یک پیام ICMP :

نوع (Type): مشخص‌کننده نوع پیام (مثلاً Echo Reply، Echo Request و غیره).
کد (Code): اطلاعات اضافی در مورد نوع پیام را ارائه می‌دهد.
مجموع بررسی (Checksum): برای اطمینان از صحت داده‌ها استفاده می‌شود.
بقیه هدر (Rest of Header): بسته به نوع و کد متغیر است (مثلاً شامل شناسه و ترتیب برای پیام‌های اکو).

انواع پیام‌های Echo در ICMP:

1. **Echo Request** (نوع 8):
   - زمانی که شما از دستور `ping` برای ارسال پینگ به یک دستگاه دیگر استفاده می‌کنید، در واقع شما یک **Echo Request** ارسال کرده‌اید.
   - این پیام به دستگاه مقصد ارسال می‌شود تا از آن خواسته شود که آیا به شبکه متصل است یا خیر.
   - این پیام به صورت یک درخواست به سیستم مقصد ارسال می‌شود تا نشان دهد که سیستم ارسال‌کننده (مبدا) منتظر دریافت پاسخ است.
2. **Echo Reply** (نوع 0):
   - زمانی که دستگاه مقصد درخواست پینگ را دریافت می‌کند، به آن پاسخ می‌دهد. پاسخ دریافتی از دستگاه مقصد **Echo Reply** است.
   - این پاسخ نشان می‌دهد که دستگاه مقصد آماده است و به شبکه متصل است.
   - در واقع، پاسخ به یک **Echo Request** که به دستگاه مقصد ارسال شده، پیام **Echo Reply** است.

روند عملکرد:

- زمانی که شما از دستور `ping` استفاده می‌کنید، یک **Echo Request** به سرور هدف ارسال می‌شود.
- اگر سرور هدف در دسترس باشد، یک **Echo Reply** به مبدا ارسال می‌شود.
- تاخیر زمانی بین ارسال **Echo Request** و دریافت **Echo Reply** به شما اطلاعاتی در مورد زمان تاخیر و وضعیت اتصال شبکه می‌دهد.

مثال:

- **Echo Request**: `ping 192.168.1.1` یا `ping example.com`
- **Echo Reply**: شما در پاسخ به این دستور پیامی مانند این دریافت خواهید کرد:

```
bytes from 192.168.1.1: icmp_seq=1 ttl=64 time=0.040 ms
```

سایر انواع ICMP:

- **Destination Unreachable**: زمانی که دستگاه مقصد نتواند به هدف برسد
- **Time Exceeded**: زمانی که بسته داده به مقصد نمی‌رسد و در راه از بین می‌رود
- **Redirect**: میزبان را هدایت می‌کند تا از یک روتر دیگر برای بهبود مسیریابی استفاده کند

NameSpaces

Namespaces در لینوکس تکنولوژی‌ای هستند که برای ایزوله کردن منابع سیستم استفاده می‌شوند. هر namespace به یک نوع خاص از منابع مربوط می‌شود و به فرآیندها اجازه می‌دهد تا تصور کنند در یک محیط جداگانه و مخصوص به

خودشان هستند. در ادامه، توضیح ساده‌ای از انواع namespaces ارائه می‌شود:

1. mnt (Mount Namespace)

- **کاربرد:** ایزوله کردن **سیستم فایل**.
- **توضیح ساده:** هر فرآیند می‌تواند سیستم فایل مخصوص خودش را داشته باشد. مثلاً فرآیند A می‌تواند یک درایو خاص را mount کند و فرآیند B آن را نبیند.
- **مثال:** در Docker، کانتینرها سیستم فایل جداگانه‌ای دارند و تغییرات در آن‌ها روی سیستم اصلی تأثیری ندارد.

2. pid (Process ID Namespace)

- **کاربرد:** ایزوله کردن **شناسه فرآیندها (PIDs)**.
- **توضیح ساده:** هر namespace می‌تواند مجموعه‌ای از فرآیندها با PIDs مختص به خودش داشته باشد. فرآیندها در namespaceهای مختلف نمی‌توانند فرآیندهای یکدیگر را ببینند.
- **مثال:** در کانتینرها، فرآیند "1" مربوط به همان کانتینر است و فرآیندهای سیستم اصلی را نمی‌بیند.

3. net (Network Namespace)

- **کاربرد:** ایزوله کردن **تنظیمات شبکه**.
- **توضیح ساده:** هر namespace شبکه خودش را دارد، شامل **آدرس‌های IP، routing tables، و پورت‌ها**.
- **مثال:** در Docker، هر کانتینر می‌تواند آدرس IP خودش را داشته باشد، بدون اینکه با شبکه سیستم اصلی تداخل پیدا کند.

4. user (User Namespace)

- **کاربرد:** ایزوله کردن **شناسه‌های کاربری (UID و GID)**.
- **توضیح ساده:** یک فرآیند می‌تواند در namespace خودش کاربر root باشد، اما در سیستم اصلی همچنان به‌عنوان یک کاربر معمولی دیده شود.
- **مثال:** کانتینرها برای امنیت بیشتر از این ویژگی استفاده می‌کنند تا فرآیندهای داخل کانتینر دسترسی مستقیم به سیستم اصلی نداشته باشند.

5. cgroup (Control Groups)

• **کاربرد:** مدیریت و محدود کردن **منابع سیستم** برای فرآیندها.

• **توضیح ساده:** cgroup به شما اجازه می‌دهد که میزان استفاده از CPU، حافظه، دیسک، و شبکه را برای گروه خاصی از فرآیندها محدود کنید.

• **مثال:** اگر یک کانتینر بیش از حد از حافظه استفاده کند، cgroup می‌تواند آن را متوقف کند یا محدودیت اعمال کند تا سایر فرآیندها تحت تأثیر قرار نگیرند.

DOCKER

قبل از داکر :

در دنیای لینوکس، مفهوم **ایزوله‌سازی فرآیندها** همیشه یک چالش بزرگ بوده است. یکی از ابزارهای اولیه و قدرتمند در این زمینه، فرمان `unshare` بود.
این ابزار به توسعه‌دهندگان و مدیران سیستم اجازه می‌داد که یک فضای نام (Namespace) مجزا برای فرآیندها ایجاد کنند.
حالا ببینیم `unshare` دقیقاً چی بود و چرا Docker جای اون رو گرفت.

فرمانی برای ایجاد فضای نام (Namespace)

در لینوکس unshare به شما امکان می‌دهد که یک فرآیند را در یک یا چند فضای نام مجزا اجرا کنید.
فضاهای نام در لینوکس به شما اجازه می‌دهند که منابع مختلف سیستم مثل شبکه، فایل‌سیستم، فرآیندها و ... را از دید فرآیندها ایزوله کنید.

```
unshare --mount --uts --ipc --net --pid --fork -- sh
```

چالش‌های `unshare` :

1. **پیچیدگی زیاد:** استفاده از `unshare` نیازمند دانش عمیقی از فضاهای نام و سیستم لینوکس بود.
2. **بدون ابزارهای مدیریت:** هیچ رابط کاربری یا ابزار ساده‌ای برای مدیریت این فضاهای نام وجود نداشت.
3. **عدم پایداری:** پیاده‌سازی‌های دستی با `unshare` ممکن بود باعث ناسازگاری یا مشکلات امنیتی شوند.
4. **ضعف در خودکارسازی:** `unshare` به تنهایی نمی‌توانست فرآیندها و وابستگی‌ها را به صورت خودکار مدیریت کند.

با امدن docker همه این مشکلات حل شد ! به جای اینکه شما بخواهید برای هر فرآیند فضاهای نام را دستی تنظیم کنید، Docker این کار را به صورت خودکار و با یک رابط کاربری ساده انجام داد.

نصب Docker در سیستم‌های مختلف

Ubuntu :

```
sudo apt update
sudo apt install -y apt-transport-https ca-certificates curl software-
properties-common
curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo gpg --
dearmor -o /usr/share/keyrings/docker-archive-keyring.gpg
echo "deb [arch=$(dpkg --print-architecture) signed-
by=/usr/share/keyrings/docker-archive-keyring.gpg]
https://download.docker.com/linux/ubuntu $(lsb_release -cs) stable" | sudo
tee /etc/apt/sources.list.d/docker.list > /dev/null
sudo apt update
sudo apt install -y docker-ce
sudo systemctl enable --now docker
```

Debian :

```
sudo apt update
sudo apt install -y apt-transport-https ca-certificates curl software-
properties-common
curl -fsSL https://download.docker.com/linux/debian/gpg | sudo gpg --
dearmor -o /usr/share/keyrings/docker-archive-keyring.gpg
echo "deb [arch=$(dpkg --print-architecture) signed-
by=/usr/share/keyrings/docker-archive-keyring.gpg]
https://download.docker.com/linux/debian $(lsb_release -cs) stable" | sudo
tee /etc/apt/sources.list.d/docker.list > /dev/null
sudo apt update
sudo apt install -y docker-ce
sudo systemctl enable --now docker
```

CentOS :

```
sudo yum install -y yum-utils
sudo yum-config-manager --add-repo
https://download.docker.com/linux/centos/docker-ce.repo
sudo yum install -y docker-ce docker-ce-cli containerd.io
sudo systemctl enable --now docker
```

بدون معطلی اولین کانتینر خودمون رو میسازیم :

1. دستور اجرای یک کانتینر:

```
docker container run -d --name app1 busybox sleep 1000
```

- **تحلیل دستور:**
- `docker container run` : برای اجرای یک کانتینر جدید.
- `-d` : اجرای کانتینر در حالت **Detached** (پس‌زمینه).
- `--name app1` : نام کانتینر به صورت `app1`.
- `busybox` : ایمیج مورد استفاده (یک ایمیج سبک و کاربردی).
- `sleep 1000` : اجرای فرمان `sleep` برای 1000 ثانیه داخل کانتینر.
- **کاربرد:** ایجاد یک کانتینر ساده و در حال اجرا برای تست یا استفاده‌های کوتاه‌مدت.

- d

فلگ `-d` در دستور `docker container run` مخفف **Detached mode** است. وقتی این فلگ را استفاده می‌کنید، کانتینر به صورت **پس‌زمینه** (Background) اجرا می‌شود و ترمینال شما آزاد باقی می‌ماند. بیایید این موضوع را دقیق‌تر بررسی کنیم:

چرا از `-d` استفاده کنیم؟

1. **اجرای فرآیندها در پس‌زمینه:** اگر می‌خواهید یک سرویس یا برنامه بدون مسدود کردن ترمینال اجرا شود، از این فلگ استفاده می‌کنید. (مثلاً یک وب‌سرور یا یک اسکریپت طولانی‌مدت)
2. **دسترسی به ترمینال:** بعد از اجرای کانتینر، می‌توانید به راحتی به ترمینال خود برگردید و دستورات دیگری را اجرا کنید.
3. **مدیریت ساده‌تر کانتینرها:** با اجرای کانتینر در حالت Detached، می‌توانید از دستورات مدیریتی مثل `docker` `logs` یا `docker exec` برای تعامل با کانتینر استفاده کنید.

در نتیجه: اگر از d - استفاده نشود برنامه `sleep` مستقیماً خروجی خود را در ترمینال نشان می‌دهد و تا زمانی که فرآیند تمام نشود، ترمینال شما اشغال خواهد بود.

BusyBox

قبل از BusyBox باید با مفهومی به نام Base Image اشنا بشیم :

Base Image

Base Image در Docker یک سیستم‌عامل کوچک یا مجموعه‌ای از ابزارهای پایه است که برای اجرای برنامه‌ها در کانتینر استفاده می‌شود. برخلاف یک سیستم‌عامل کامل در ماشین‌های مجازی (VM)، Base Image شامل فقط بخش‌های ضروری است و لایه‌ای سبک از نرم‌افزارها را ارائه می‌دهد. این موضوع باعث کاهش حجم و افزایش سرعت کانتینرها می‌شود و تمرکز را روی نیازمندی‌های خاص برنامه قرار می‌دهد.

مدیریت کانتینرها در Docker: دستورات کاربردی

توضیحات دستورات Docker:

1. **`docker container ls`**
   - نمایش لیست کانتینرهای در حال اجرا.
2. **`docker container ls -a`**
   - نمایش لیست تمام کانتینرها (در حال اجرا، متوقف‌شده، یا حذف‌شده).
3. **`docker container rm --force 7470a53eea81`**
   - حذف اجباری یک کانتینر با استفاده از ID یا نام آن، حتی اگر در حال اجرا باشد.
4. **`docker container stop app3`**
   - متوقف کردن کانتینر با نام `app3` .
5. **`docker container start app3`**
   - شروع به کار دوباره کانتینر متوقف‌شده با نام `app3` .
6. **`docker container rm --force $(docker container ls -qa)`**
   - حذف اجباری تمام کانتینرها (در حال اجرا یا متوقف).
7. **`docker container stop -t 5`**
   - متوقف کردن تمام کانتینرها با دادن 5 ثانیه زمان برای پایان عملیات.
8. **`docker container kill app1`**
   - متوقف کردن آنی (بدون مهلت) کانتینر با نام `app1` .

9. `docker container inspect app1`
   - نمایش اطلاعات دقیق درباره کانتینر `app1` (مثل شبکه، تنظیمات محیطی، و منابع).
10. `docker container logs --tail 200 -f app1`
    - مشاهده آخرین 200 خط لاگ کانتینر `app1` به صورت زنده.
11. `docker container exec -it app1 sh`
    - اجرای یک شل تعاملی داخل کانتینر `app1`.
12. `docker container exec -it --env=class=devops app1 sh`
    - اجرای یک شل تعاملی داخل کانتینر `app1` همراه با تنظیم متغیر محیطی `class` برابر `devops`.
13. `docker container exec -it app1 top`
    - اجرای دستور `top` برای مشاهده فرآیندهای فعال در کانتینر `app1`.
14. `docker container restart app1`
    - ری‌استارت کردن کانتینر `app1`.
15. `docker container cp webapp:/usr/share/nginx/html/index.html .`
    - کپی فایل `index.html` از داخل کانتینر `webapp` به مسیر فعلی سیستم میزبان.
16. `docker container cp default.conf webapp:/etc/nginx/conf.d/`
    - کپی فایل `default.conf` از سیستم میزبان به مسیر مشخص‌شده داخل کانتینر `webapp`.
17. `docker container top webapp`
    - نمایش فرآیندهای در حال اجرا داخل کانتینر `webapp`.
18. `docker container run -d --name app2 --restart always busybox:latest ping 1.1.1.1`
    - اجرای کانتینر جدید با نام `app2` با استفاده از ایمیج `busybox:latest` به صورت پس‌زمینه (`-d`)، که دستور `ping 1.1.1.1` را اجرا می‌کند و در صورت توقف، همیشه ری‌استارت می‌شود (`--restart always`).

بررسی دقیق کانتینرها با Docker Inspect


```
"State": {
"Status": "running",
"Running": true,
"Paused": false,
"Restarting": false,
"OOMKilled": false,
"Dead": false,
"Pid": 12345,
"ExitCode": 0,
"StartedAt": "2025-01-08T12:00:00Z",
"FinishedAt": "0001-01-01T00:00:00Z"
}
```


1. `Status`
   - وضعیت فعلی کانتینر را نشان می‌دهد. مقدار آن می‌تواند یکی از موارد زیر باشد:
     - `running`: کانتینر در حال اجرا است

- `exited` : کانتینر متوقف شده است
- `paused` : کانتینر متوقف (Pause) شده است
- `restarting` : کانتینر در حال ری‌استارت است
- `dead` : کانتینر به حالت غیرقابل استفاده رفته است

---

2. **`Running`**
   - مقدار `true` یا `false` نشان می‌دهد که کانتینر در حال اجرا است یا خیر.

---

3. **`Paused`**
   - مقدار `true` یا `false` نشان می‌دهد که کانتینر متوقف (Pause) شده است یا خیر.

---

4. **`Restarting`**
   - مقدار `true` یا `false` نشان می‌دهد که کانتینر در حال ری‌استارت است یا خیر.

---

5. **`OOMKilled`**
   - مقدار `true` یا `false` نشان می‌دهد که کانتینر به دلیل استفاده بیش از حد از حافظه توسط سیستم عامل متوقف شده است.

---

6. **`Dead`**
   - مقدار `true` یا `false` نشان می‌دهد که کانتینر به حالت غیرقابل استفاده رسیده است.

---

7. **`Pid`**
   - شناسه پردازش اصلی (Main Process ID) که داخل کانتینر اجرا می‌شود.

---

8. **`ExitCode`**
   - کد خروج کانتینر پس از اجرای فرآیند. مقادیر معمول:
     - `0` : اجرای موفقیت‌آمیز.
     - غیر از `0` : خطا در اجرای فرآیند.

---

9. `StartedAt`
   - زمان دقیق شروع به کار کانتینر در قالب زمان UTC.

---

10. `FinishedAt`
    - زمان دقیق توقف یا پایان فرآیند کانتینر در قالب زمان UTC. اگر کانتینر هنوز در حال اجرا باشد، مقدار `01T00:00:00Z-01-0001` نمایش داده می‌شود.

لایه‌های Image در Docker: RW و RO

داکر از یک ساختار لایه‌ای برای مدیریت (Images) و کانتینرها استفاده می‌کند. این طراحی لایه‌ای به کارآمدی و انعطاف‌پذیری بالای Docker کمک می‌کند. در اینجا، نقش لایه‌های RO و RW و مفهومی مثل Copy-on-Write توضیح داده می‌شود.

1. لایه‌های Read-Only (RO)

- **تعریف:**
  ایمیج های Docker شامل چندین لایه Read-Only (فقط قابل خواندن) هستند. این لایه‌ها تغییرناپذیرند و اساساً شامل داده‌هایی هستند که هنگام ساخت (Image) با استفاده از `Dockerfile` ایجاد شده‌اند.
- **ویژگی‌ها:**
  - لایه‌های Read-Only بین کانتینرهای مختلفی که از یک ایمیج ساخته شده‌اند، به اشتراک گذاشته می‌شوند.
  - این اشتراک‌گذاری باعث صرفه‌جویی در فضای ذخیره‌سازی می‌شود.

2. لایه RW (Read-Write)

- **تعریف:**
  هنگامی که یک کانتینر از یک ایمیج Docker ایجاد می‌شود، یک لایه جدید RW به صورت موقت (Ephemeral) به بالای لایه‌های RO اضافه می‌شود.
- **ویژگی‌ها:**
  - تغییرات یا داده‌های جدید که در کانتینر ایجاد می‌شوند (نوشتن فایل، تغییرات، نصب برنامه‌ها)، فقط در این لایه RW ذخیره می‌شوند.
  - لایه RW مختص همان کانتینر است و با دیگر کانتینرها به اشتراک گذاشته نمی‌شود.

Copy-On-Write (کپی هنگام نوشتن)

مفهوم **Copy-On-Write** یکی از اصول کلیدی در Docker است که نحوه مدیریت تغییرات در لایه‌های Image و کانتینرها را توضیح می‌دهد.

1. **مفهوم:**
   - اگر کانتینر بخواهد داده‌ای را که در یکی از لایه‌های RO قرار دارد تغییر دهد، داده موردنظر به لایه RW کپی می‌شود.
   - سپس تغییرات روی نسخه کپی‌شده اعمال می‌شود، بدون این که لایه RO اصلی تحت تأثیر قرار گیرد.
2. **مزایا:**

- **صرفه‌جویی در حافظه:** داده‌ها فقط در صورت نیاز کپی می‌شوند، که باعث استفاده بهینه از فضای ذخیره‌سازی می‌شود.
- **امنیت لایه‌ها:** لایه‌های RO همیشه دست‌نخورده باقی می‌مانند و می‌توانند مجدداً توسط کانتینرهای دیگر استفاده شوند.

3. **مثال عملی:**
   - فرض کنید فایل `config.txt` در یکی از لایه‌های RO قرار دارد.
   - اگر کانتینر بخواهد محتوای این فایل را تغییر دهد:
     - فایل ابتدا به لایه RW کپی می‌شود.
     - تغییرات فقط روی این نسخه کپی‌شده اعمال می‌شوند.

چرا این ساختار مهم است؟

- **بازدهی بالا:** لایه‌های مشترک بین کانتینرها باعث کاهش استفاده از حافظه می‌شوند.
- **انعطاف‌پذیری:** هر کانتینر می‌تواند به صورت مستقل تغییرات خود را اعمال کند، بدون این که بر دیگر کانتینرها تأثیر بگذارد.
- **پشتیبانی از نسخه‌بندی:** لایه‌های RO به حفظ تاریخچه تغییرات در تصاویر Docker کمک می‌کنند.

داکر فایل (Dockerfile)

یک فایل متنی ساده است که شامل مجموعه‌ای از دستورات است که به داکر می‌گوید چگونه یک ایمیج بسازد. این فایل، نقشه‌ی راهی برای ساخت ایمیج است و به شما امکان می‌دهد تمام مراحل، تنظیمات و وابستگی‌های لازم برای اجرای یک اپلیکیشن را تعریف کنید.

ویژگی‌های Dockerfile:

1. **دستورات مشخص:** هر خط یک دستور خاص را اجرا می‌کند (مثل نصب پکیج‌ها یا کپی فایل‌ها).
2. **قابل بازتولید:** هر بار که از آن استفاده کنید، ایمیج دقیقاً به همان شکل قبلی ساخته می‌شود.
3. **چندمرحله‌ای:** می‌توانید ایمیج‌هایی سبک‌تر و بهینه‌تر بسازید.

ساختار کلی Dockerfile:

1. **FROM:** مشخص کردن Base Image (مثل python:3.13).
2. **RUN:** اجرای دستورات خط فرمان (مثل نصب ابزارها).
3. **ADD یا COPY:** کپی کردن فایل‌ها به ایمیج.
4. **WORKDIR:** تعیین دایرکتوری کاری در ایمیج.
5. **ENTRYPOINT یا CMD:** مشخص کردن دستوراتی که هنگام اجرای کانتینر اجرا می‌شوند.

آموزش ساخت یک ایمیج و اجرای کانتینر از ابتدا

مرحله ۱: ساخت دایرکتوری پروژه

ابتدا یک دایرکتوری برای پروژه خود ایجاد کنید و به آن وارد شوید.

```
mkdir my-python-app
cd my-python-app
```

مرحله ۲: ساخت فایل های مورد نیاز

ساخت فایل پایتونی :

```
touch main.py
vim main.py
```

سپس کد برنامه خود را به این فایل انتقال می دهیم :

```
import time
time.sleep(100)
print("Hello Docker Class")
time.sleep(1000)
```

ساخت داکرفایل :

```
touch Dockerfile
vim Dockerfile
```

سپس محتوای زیر را در آن اضافه کنید:

```
FROM python:3.13.1-alpine3.20
RUN mkdir /app
WORKDIR /app
COPY main.py .
CMD ["python", "main.py"]
```

```
FROM python:3.13.1-alpine3.20
```

- **FROM:** این دستور برای تعریف Base Image استفاده می‌شود.
- ایمیج پایه `python:3.13.1-alpine3.20` نسخه‌ی سبک و بهینه‌ای از پایتون است که بر اساس Alpine Linux ساخته شده. این ترکیب برای ایجاد ایمیج‌هایی با حجم کمتر مناسب است.

```
RUN mkdir /app
```

- **RUN:** برای اجرای دستورات خط فرمان در مرحله‌ی ساخت ایمیج استفاده می‌شود.
  - این دستور یک دایرکتوری به نام `app/` داخل ایمیج ایجاد می‌کند تا فایل‌ها و پروژه‌ها را سازمان‌دهی کند.

```
WORKDIR /app
```

- **WORKDIR:** این دستور مشخص می‌کند که دستورات بعدی (مثل COPY و CMD) در کدام دایرکتوری اجرا شوند.
  - دایرکتوری `app/` به عنوان دایرکتوری کاری تعیین می‌شود.

```
COPY main.py .
```

- **COPY:** فایل‌ها را از سیستم میزبان به داخل ایمیج کپی می‌کند.
  - این دستور فایل `main.py` را از دایرکتوری میزبان به دایرکتوری `app/` داخل ایمیج منتقل می‌کند.

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

- **CMD:** مشخص می‌کند که کانتینر هنگام اجرا چه دستوری را اجرا کند.
  - این دستور فایل `main.py` را با استفاده از پایتون اجرا می‌کند.

مرحله 3: ساخت ایمیج Docker

پس از آماده‌سازی فایل‌ها، به دایرکتوری پروژه بروید و ایمیج خود را با دستور زیر بسازید:

```
docker image build . -t mypythonapp:v1.0.0
```

توضیحات:

- `.` - dot محل فایل‌های پروژه (دایرکتوری جاری)
- `-t mypythonapp:v1.0.0` تگ کردن ایمیج با نام mypythonapp و نسخه v1.0.0.

مرحله 4: اجرای کانتینر

ایمیج ساخته شده را به یک کانتینر تبدیل کنید و آن را اجرا کنید:

```
docker container run --name mypythonapp_container mypythonapp:v1.0.0
```

توضیحات:

- `--name mypythonapp_container` : نام دلخواه برای کانتینر
- `mypythonapp:v1.0.0` : نام و نسخه ایمیج

مرحله ۵: مشاهده لاگ کانتینر

برای مشاهده خروجی برنامه داخل کانتینر، از دستور زیر استفاده کنید:

```
docker container logs –f mypythonapp_container
```

مرحله 6: حذف و متوقف کردن کانتینر و ایمیج

متوقف کردن کانتینر:

```
docker container stop mypythonapp_container
```

حذف کانتینر:

```
docker container rm mypythonapp_container
```

چرا Dockerfile با حجم زیاد مشکل‌ساز است؟

اگر Dockerfile به درستی بهینه‌سازی نشود و منجر به ساخت ایمیج‌های حجیم شود، مشکلات متعددی به وجود می‌آید. در زیر برخی از این مشکلات و دلایل اجتناب از Dockerfile با حجم زیاد آورده شده است:

مشکلات ناشی از حجم بالای ایمیج:

1. **کاهش سرعت انتقال و اجرا:**
   - ایمیج‌های حجیم زمان بیشتری برای انتقال از یک سیستم به سیستم دیگر، یا از یک ریجستری به سرور، نیاز دارند.
   - اجرای کانتینر از ایمیج‌های حجیم زمان بیشتری می‌گیرد.
2. **افزایش مصرف منابع:**
   - فضای دیسک بیشتری روی سیستم میزبان اشغال می‌شود.
   - منابع شبکه برای انتقال ایمیج‌ها بیشتر مصرف می‌شود.
3. **افزایش پیچیدگی مدیریت:**
   - نگهداری ایمیج‌های بزرگ و چندلایه پیچیده‌تر می‌شود.
   - اشکال‌زدایی ایمیج‌های حجیم دشوارتر است.
4. **مشکلات امنیتی:**

- ایمیج‌های حجیم معمولاً شامل پکیج‌ها و فایل‌های غیرضروری هستند که ممکن است حاوی آسیب‌پذیری‌های امنیتی باشند.
- حذف فایل‌ها یا ابزارهای غیرضروری در ایمیج، سطح حمله را کاهش می‌دهد.

راه‌حل‌ها:

1. **استفاده از Base Image مناسب:**
   - انتخاب ایمیج‌های کوچک‌تر مانند `alpine` به جای ایمیج‌های عمومی و بزرگ.

2. **ترکیب دستورات با `&&` :**
   - کاهش تعداد لایه‌ها با ترکیب دستورات متعدد در یک لایه.
3. **پاک کردن فایل‌های موقت:**
   - فایل‌های موقتی که در فرآیند نصب ایجاد می‌شوند، باید در همان دستور `RUN` حذف شوند.

بهینه‌سازی Dockerfile

نسخه اولیه (بدون بهینه‌سازی):

```
FROM python:3.13.1-alpine3.20

RUN apk update
RUN apk add --no-cache bash
RUN mkdir /app
WORKDIR /app
COPY main.py .
CMD ["python", "main.py"]
```

مشکل: این Dockerfile سه دستور `RUN` دارد که باعث ایجاد سه لایه مجزا می‌شود.

نسخه بهینه شده :

```
FROM python:3.13.1-alpine3.20

# ترکیب دستورات نصب و ایجاد دایرکتوری در یک دستور
RUN apk update && \
    apk add --no-cache bash && \
    mkdir -p /app

# تنظیم دایرکتوری کاری و کپی فایل‌ها
WORKDIR /app
COPY main.py .

# اجرای برنامه
CMD ["python", "main.py"]
```

توضیحات:

1. **`apk update && apk add --no-cache bash`:**

دستورات `update` و نصب `bash` در یک خط ترکیب شدند.

2. **`mkdir -p /app`:**

دستور ایجاد دایرکتوری `app/` نیز در همان دستور `RUN` اضافه شده است.

3. **`&& and \`:**

استفاده از `&&` برای زنجیره کردن دستورات و `\` برای خوانایی بهتر کد.

مثال: Dockerfile حجیم با استفاده از GCC

ابتدا یک **Dockerfile** غیر بهینه برای برنامه‌ای که نیاز به کامپایل با GCC دارد را ایجاد می‌کنیم. سپس این فایل را بهینه می‌کنیم.

نسخه اولیه (بدون بهینه‌سازی):

```
# استفاده از ایمیج پایه Alpine
FROM alpine:3.21.0

# نصب ابزارهای مورد نیاز برای کامپایل
RUN apk update && apk add alpine-sdk

# کپی فایل سورس کد به داخل کانتینر
COPY main.c /app/main.c

# تنظیم دایرکتوری کاری
WORKDIR /app

# کامپایل برنامه
RUN gcc -Wall main.c -o main

# اجرای برنامه
CMD ["./main"]
```

مشکلات این Dockerfile

1. **حجم بالا:** ابزارهای مورد نیاز برای کامپایل (مانند GCC) بعد از کامپایل مورد نیاز نیستند، اما همچنان در ایمیج باقی می‌مانند.
2. **غیر بهینه بودن:** همه ابزارها در یک مرحله نصب می‌شوند، که حجم نهایی ایمیج را افزایش می‌دهد.

نسخه بهینه شده :

```
# مرحله اول: بیلد (Build Stage)
FROM alpine:3.21.0 AS build

# نصب ابزارهای مورد نیاز برای بیلد
RUN apk update && apk add alpine-sdk
```

```
# تنظیم دایرکتوری کاری و کپی سورس کد
WORKDIR /app
COPY main.c .

# کامپایل سورس کد
RUN gcc -Wall main.c -o main

# مرحله دوم: ایمیج نهایی (Final Stage)
FROM alpine:3.21.0

# تنظیم دایرکتوری کاری
WORKDIR /app

# کپی فایل کامپایل‌شده از مرحله بیلد
COPY --from=build /app/main .

# اجرای برنامه
CMD ["./main"]
```

ویژگی‌های نسخه بهینه:

1. **Multi-stage build:** ابزارهای کامپایل فقط در مرحله اول (بیلد) نصب می‌شوند و به ایمیج نهایی منتقل نمی‌شوند.
2. **حجم کمتر:** ایمیج نهایی فقط شامل فایل اجرایی (binary) است.
3. **سازماندهی بهتر:** مراحل جداگانه باعث خوانایی و مدیریت بهتر Dockerfile می‌شوند.

جلوگیری از نصب دوباره پکیج‌ها

یکی از مشکلات رایج در Dockerfileهای غیر بهینه این است که هر بار که کد تغییر می‌کند، پکیج‌ها و وابستگی‌ها دوباره نصب می‌شوند. برای جلوگیری از این مشکل، می‌توانیم با استفاده از ترتیب مناسب دستورات در Dockerfile، فقط در صورت تغییر فایل وابستگی‌ها (مثل `requirements.txt`) پکیج‌ها را نصب کنیم.

نسخه اولیه (بدون بهینه‌سازی):

```
# استفاده از ایمیج پایه پایتون
FROM python:3.10-slim

# ایجاد دایرکتوری کاری
WORKDIR /app

# کپی کردن تمام فایل‌های پروژه به کانتینر
COPY . .
```

```
# نصب وابستگی‌ها از فایل requirements.txt
RUN pip install --no-cache-dir -r requirements.txt

# دستور اجرا برای اجرای فایل اصلی
CMD ["python", "main.py"]
```

مشکلات این Dockerfile:

1. **کپی کردن زودهنگام فایل‌ها:**
   - دستور `COPY . .` تمام فایل‌های پروژه (از جمله `main.py`) را قبل از نصب پکیج‌ها کپی می‌کند.
   - این باعث می‌شود که هر تغییری در فایل‌های کد (مثل `main.py`) باعث شود لایه نصب وابستگی‌ها (`RUN ... pip install`) دوباره اجرا شود.
2. **نصب دوباره پکیج‌ها:**
   - حتی اگر فایل `requirements.txt` تغییری نکرده باشد، داکر به دلیل تغییر در فایل‌های کپی شده، تمام پکیج‌ها را دوباره نصب می‌کند. این عملکرد زمان‌بر و غیر بهینه است.

نسخه بهینه شده :

```
# استفاده از ایمیج پایه پایتون
FROM python:3.10-slim

# تعریف متغیر محیطی برای جلوگیری از تولید فایل‌های کش
ENV PYTHONDONTWRITEBYTECODE=1
ENV PYTHONUNBUFFERED=1

# ایجاد دایرکتوری کاری
WORKDIR /app

# کپی کردن فایل وابستگی‌ها به کانتینر
COPY requirements.txt .

# نصب وابستگی‌ها از فایل requirements.txt
RUN pip install --no-cache-dir -r requirements.txt

# کپی کردن فایل‌های پروژه به دایرکتوری کاری
COPY . .

# دستور اجرا برای اجرای فایل اصلی
CMD ["python", "main.py"]
```

مفهوم و کاربرد دستور ADD در Dockerfile

دستور **ADD** یکی از دستورات موجود در Dockerfile است که برای اضافه کردن فایل‌ها یا پوشه‌ها به یک ایمیج داکر استفاده می‌شود. این دستور برخلاف دستور **COPY** قابلیت‌هایی اضافی دارد که آن را در شرایط خاص بسیار مفید می‌کند.

- **کپی (COPY)**: فایل‌ها یا پوشه‌ها را بدون هیچ تغییری از سیستم میزبان به ایمیج اضافه می‌کند.
- **اد (ADD)**: علاوه بر کپی کردن، می‌تواند:
  1. فایل‌های آرشیو (مانند `tar.`) را به‌طور خودکار اکسترکت کند.
  2. فایل‌ها را از منابع شبکه (URL) دانلود و اضافه کند.

مثال: اضافه کردن فایل آرشیو با ADD

فرض کنید یک فایل آرشیو به نام `website.tar` دارید که شامل دو فایل زیر است:

- `index.html`
- `default.conf`

شما می‌خواهید این فایل‌ها را به صورت اکسترکت‌شده داخل ایمیج قرار دهید. برای این کار از دستور **ADD** استفاده می‌کنیم.

```
FROM nginx:latest

# اضافه کردن و اکسترکت کردن فایل آرشیو
ADD website.tar /usr/share/nginx/html
```

نتیجه:

وقتی این دستور اجرا شود:

- فایل `website.tar` در مسیر `usr/share/nginx/html/` اکسترکت می‌شود.
- فایل‌های `index.html` و `default.conf` به این مسیر اضافه می‌شوند.

کار با Bash و Shell در کانتینرهای Docker

گاهی اوقات نیاز داریم وارد محیط داخلی یک کانتینر شویم تا دستورات را اجرا یا مشکلات را بررسی کنیم. Docker این امکان را با دستور **`docker exec`** فراهم می‌کند که به شما اجازه می‌دهد به‌صورت تعاملی (interactive) یا غیرتعاملی دستورات را داخل کانتینر اجرا کنید.

1. اجرای دستورات درون کانتینر با Bash/Shell

برای وارد شدن به محیط Shell (یا Bash) یک کانتینر:

```
docker container exec -it <CONTAINER_NAME_OR_ID> bash
```

- **`-i`** : حالت تعاملی (interactive) را فعال می‌کند
- **`-t`** : حالت تعاملی (interactive) را فعال می‌کند

- اجرای شل Bash داخل کانتینر : `bash`

مثال:

فرض کنید یک کانتینر به نام `myapp` دارید:

```
docker container exec –it myapp bash
```

2. اگر Bash در کانتینر نصب نباشد؟

بعضی از ایمیج‌های Docker مانند Alpine به‌صورت پیش‌فرض Bash ندارند. در این مواقع، می‌توانید از **sh** (Shell ساده‌تر) استفاده کنید:

```
docker container exec –it <CONTAINER_NAME_OR_ID> sh
```

3. اجرای دستورات خاص بدون وارد شدن به محیط کانتینر

اگر فقط می‌خواهید یک دستور خاص را داخل کانتینر اجرا کنید، نیازی به وارد شدن به محیط Bash یا Shell ندارید.
می‌توانید دستور موردنظر را مستقیم اجرا کنید:

```
docker container exec <CONTAINER_NAME_OR_ID>
```

تفاوت و کاربردهای ENTRYPOINT و CMD در Docker

مفاهیم ENTRYPOINT و CMD در فایل Dockerfile مشخص می‌کنند که کانتینر در زمان اجرا چه کاری انجام دهد.
هرچند هر دو برای تعریف فرمان‌های اجرایی استفاده می‌شوند، اما کاربردها و رفتارهای متفاوتی دارند.

1. CMD چیست؟

- **** برای تعریف فرمان پیش‌فرض استفاده می‌شود که در زمان اجرای کانتینر اجرا می‌شود.
- اگر هنگام اجرای کانتینر دستور دیگری ارائه شود، دستور **CMD** نادیده گرفته می‌شود.
- **** تنها یک بار در فایل Dockerfile تعریف می‌شود (اگر چند بار تعریف شود، آخرین مقدار استفاده می‌شود).

مثال:

```
FROM ubuntu:latest
CMD ["echo", "Hello from CMD"]
```

2. ENTRYPOINT چیست؟

- **** برای تعریف فرمانی استفاده می‌شود که همیشه اجرا می‌شود و نمی‌توان آن را هنگام اجرای کانتینر تغییر داد.
- اگر هنگام اجرای کانتینر دستور جدیدی ارائه شود، به عنوان آرگومان به **ENTRYPOINT** اضافه می‌شود.

مثال:

```
FROM ubuntu:latest
ENTRYPOINT ["echo"]
```

5. چه زمانی از ENTRYPOINT و CMD استفاده کنیم؟

- از **CMD** برای تنظیم مقادیر پیش‌فرض یا دستورات قابل تغییر استفاده کنید.
- از **ENTRYPOINT** برای وظایف ثابت و غیرقابل تغییر استفاده کنید (مانند اجرای یک سرور، ابزار خاص، یا اسکریپت اصلی).

متغیرها در Docker: کاربرد و اهمیت

متغیرها در Docker نقش مهمی در پویایی و انعطاف‌پذیری کانتینرها دارند. شما می‌توانید با استفاده از متغیرها مقادیر مختلفی را به کانتینرها بدهید، بدون این که نیاز به تغییر Dockerfile باشد.

چرا از متغیرها استفاده کنیم؟

1. **پیکربندی آسان‌تر:** می‌توانید بدون تغییر کد، تنظیمات مختلفی را برای محیط‌های توسعه، تست و تولید اعمال کنید.
2. **کاهش سخت‌کدینگ (Hardcoding):** به جای این که مقادیر را مستقیم در کد بنویسید، از متغیرها استفاده می‌کنید.
3. **پویایی:** رفتار کانتینر را می‌توان با تغییر متغیرها تغییر داد.

آموزش Push و Pull کردن یک Image روی Docker Hub

گام 1: لاگین به Docker Hub

ابتدا باید به حساب Docker Hub خود وارد شوید. برای این کار، از دستور زیر استفاده کنید:

```
docker login
```

گام 2: ساخت یک ایمیج و تگ کردن آن

ابتدا یک ایمیج بسازید و برای شناسایی بهتر، آن را تگ کنید. برای مثال:

```
docker build -t <username>/<image-name>:<tag> .

docker build -t kasrajamshidi/myapp:v1.0.0 .
```

گام 3: Push کردن ایمیج به Docker Hub

برای ارسال ایمیج به Docker Hub، از دستور زیر استفاده کنید:

```
docker push <username>/<image-name>:<tag>

docker push kasrajamshidi/myapp:v1.0.0
```

گام 4: Pull کردن ایمیج از Docker Hub

برای دانلود ایمیج از Docker Hub به سیستم خود، از دستور زیر استفاده کنید:

```
docker pull <username>/<image-name>:<tag>

docker pull kasrajamshidi/myapp:v1.0.0
```

Harbor

نصب Harbor

1. از سایت رسمی Harbor، آخرین نسخه را دانلود کنید:

```
wget https://github.com/goharbor/harbor/releases/download/v2.9.0/harbor-offline-installer-v2.12.2.tgz
```

2. فایل دانلودشده را استخراج کنید:

```
tar -xzvf harbor-online-installer-v2.9.0.tgz
```

3. وارد پوشه Harbor شوید:

```
cd harbor
```

4. فایل پیش‌فرض harbor.yml.tmpl را به harbor.yml کپی کنید:

```
cp harbor.yml.tmpl harbor.yml
```

5. فایل harbor.yml را ویرایش کنید:

```
nano / vim harbor.yml
```

تغییرات زیر را اعمال کنید:

• **hostname**: سرور خود را وارد کنید IP نام دامنه یا آدرس

```
hostname: <your-server-ip>
```

• **HTTPS یا HTTP**:

• اگر از HTTPS استفاده نمی‌کنید، بخش https را کامنت کنید و بخش http را فعال کنید:

```
http:
  port: 80
```

6. اسکریپت نصب را اجرا کنید:

```
./install.sh
```

• نام کاربری و رمز عبور پیش‌فرض:

• **Username:** admin

• **Password:** Harbor12345 (پیش‌فرض) تنظیم شده در فایل harbor.yml.

وارد شدن به Harbor:

ابتدا باید به رجیستری Harbor متصل شوید:

```
docker login <harbor-url or ip >
```

```
docker login harbor.kasrajamshidi.com or ip
```

پس از اجرای دستور، نام کاربری و رمز عبور خود را وارد کنید.

تگ کردن ایمیج:

برای ارسال یک ایمیج، باید آن را با نام کامل رجیستری و پروژه در Harbor تگ کنید:

```
docker tag <local-image> <harbor-url>/<project-name>/<image-name>:<tag>
```

- **<local-image>** : (مثلاً `nginx:latest`) نام یا شناسه ایمیج محلی
- **<harbor-url>** : آدرس رجیستری Harbor
- **<project-name>** : نام پروژه‌ای که در Harbor ساخته‌اید
- **<image-name>** : نام دلخواه برای ایمیج
- **<tag>** : (مثلاً `v1.0`). تگ دلخواه

مثال:

```
docker tag nginx:latest harbor.kasrajamshidi.com/myproject/nginx:v1.0
```

PUSH

برای ارسال ایمیج به Harbor:

```
docker push <harbor-url>/<project-name>/<image-name>:<tag>
```

مثال:

```
docker push harbor.kasrajamshidi.com/myproject/nginx:v1.0
```

PULL

برای دریافت ایمیج از Harbor:

```
docker pull <harbor-url>/<project-name>/<image-name>:<tag>
```

مثال:

```
docker pull harbor.mycompany.com/myproject/nginx:v1.0
```

بررسی ایمیج‌ها:

```
docker images
```

Docker Volume

داکر ابزارهایی برای مدیریت داده‌های داخل کانتینرها ارائه می‌دهد. این داده‌ها می‌توانند دائمی یا موقتی باشند. دو روش اصلی برای این کار وجود دارد که به توان به کمک آن اطلاعات و محتوای کانتینر را حفظ کرد حتی در صورت حذف شدن کانتینر .

چرا از Volume استفاده کنیم؟

- **داده‌ها پایدار می‌مانند:** حتی اگر کانتینر حذف شود، داده‌ها باقی می‌مانند.
- **مدیریت توسط Docker:** لازم نیست نگران محل ذخیره‌سازی باشید.
- **امنیت بیشتر:** داده‌ها مستقل از سیستم فایل میزبان نگهداری می‌شوند.

چطور از Volume استفاده کنیم؟

1. **ایجاد Volume:**

```
docker volume create my_volume
```

این دستور یک Volume به نام my_volume می‌سازد.

2. **اتصال Volume به کانتینر:**

```
docker run -d --name my_container -v my_volume:/app busybox
```

• /app: مسیر داخل کانتینر.

• my_volume: نام Volume.

3. **مشاهده لیست Volumeها:**

```
docker volume ls
```

2. Bind Mount

مفهوم **Bind Mounts** به شما این امکان را می‌دهند که یک پوشه یا فایل از سیستم میزبان را مستقیماً به داخل کانتینر متصل کنید.

چرا از Bind Mount استفاده کنیم؟

• **دسترسی مستقیم به فایل‌های میزبان:** برای توسعه برنامه، می‌توانید فایل‌های پروژه را مستقیماً داخل کانتینر استفاده کنید.

• **کنترل بیشتر روی مکان داده‌ها:** شما تصمیم می‌گیرید داده‌ها کجا ذخیره شوند.

• امنیت کمتر نسبت به docker volume به دلیل ایزوله نبودن

چطور از Bind Mount استفاده کنیم؟

1. اتصال یک دایرکتوری از میزبان به کانتینر:

```
docker run -d --name my_container -v /home/user/project:/app busybox
```

• /home/user/project: مسیر روی میزبان.

• /app: مسیر داخل کانتینر.

2. مثال :

فرض کنید یک پروژه Python دارید:

```
docker run -d -v $(pwd):/app python:3.9
```

• $(pwd): مسیر فعلی روی میزبان.

• /app: مسیر داخل کانتینر.

فایل سیستم Docker: OverlayFS

داکر از سیستم فایل **OverlayFS** (یا نسخه‌های مشابه مثل Overlay2) برای مدیریت و ذخیره داده‌های کانتینرها استفاده می‌کند. این سیستم فایل به‌طور خاص برای عملکرد بهتر و کارایی بیشتر در محیط‌های کانتینری طراحی شده است.

فایل سیستم Docker: OverlayFS

برای مدیریت و ذخیره داده‌های کانتینرها (Overlay2 یا نسخه‌های مشابه مثل) **OverlayFS** از سیستم فایل Docker استفاده می‌کند. این سیستم فایل به‌طور خاص برای عملکرد بهتر و کارایی بیشتر در محیط‌های کانتینری طراحی شده است.

OverlayFS چیست؟

این یک سیستم فایل لایه‌ای (Layered Filesystem) است که به شما اجازه می‌دهد فایل‌ها را از چندین لایه ادغام کنید. این سیستم فایل بسیار سریع و کارآمد است و در Docker برای مدیریت داده‌های کانتینرها به کار می‌رود.

چطور کار می‌کند؟

1. **لایه‌های Read-Only (RO):**
   - ایمیج کانتینرها شامل چندین لایه فقط خواندنی (RO) است.
   - این لایه‌ها تغییر نمی‌کنند و ثابت باقی می‌مانند.
2. **لایه Writeable (RW):**
   - زمانی که یک کانتینر اجرا می‌شود، یک لایه بالایی قابل نوشتن (RW) به آن اضافه می‌شود.
   - تمام تغییراتی که در کانتینر انجام می‌دهید (مثل ایجاد فایل یا حذف فایل) فقط در این لایه RW اعمال می‌شود.
3. بخش **Union View:**
   - لایه‌ها را طوری نمایش می‌دهد که شما فقط یک سیستم فایل یکپارچه می‌بینید.

ساختار OverlayFS در Docker

فرض کنید یک کانتینر از یک تصویر (Image) ساخته شده است:

- **Base Image:** لایه پایه (RO).
- **Application Layer:** کدهای اپلیکیشن شما (RO).
- **Writable Layer:** لایه نوشتنی که مخصوص کانتینر است (RW).

هر تغییری که در فایل‌های کانتینر بدهید، فقط در لایه RW ذخیره می‌شود، در حالی که لایه‌های پایین‌تر دست‌نخورده باقی می‌مانند.

مزایای OverlayFS

1. **صرفه‌جویی در فضای دیسک:**
   - لایه‌های مشترک بین کانتینرها (مثل Base Image) تکرار نمی‌شوند.
2. **عملکرد بالا:**
   - نیازی به کپی کردن تمام داده‌ها نیست؛ فقط تغییرات ذخیره می‌شوند.
3. **مدیریت بهتر لایه‌ها:**
   - هر لایه می‌تواند مجزا نگهداری شود و از لایه‌های دیگر استفاده کند.

چرا OverlayFS برای RO و RW محدودیت دارد؟

این مفهوم در برخی موارد **رفتار محدود یا پیچیده‌ای** دارد، به‌ویژه اگر نیاز به دسترسی همزمان **خواندنی (RO)** و **نوشتنی (RW)**داشته باشید.

مشکلات اصلی:

1. **همزمانی RO و RW:**
   - مفهوم OverlayFS به‌طور پیش‌فرض لایه‌های RO و RW را جدا می‌کند.
   - اگر بخواهید فایل‌های یک لایه RO را در لایه RW تغییر دهید، ممکن است رفتار غیرمنتظره‌ای رخ دهد.
2. **عملیات نوشتن زیاد (Heavy Writes):**
   - مفهوم OverlayFS برای حجم زیاد عملیات نوشتن یا تغییر در داده‌ها طراحی نشده است. در این شرایط ممکن است عملکرد پایین بیاید.
3. **عدم پشتیبانی از برخی ویژگی‌ها:**
   - برخی ویژگی‌های پیشرفته سیستم فایل (مثل تغییر دسترسی دقیق RO/RW در سطح فایل) توسط OverlayFS پشتیبانی نمی‌شود.

در نتیجه اگر برنامه شما به شدت به عملیات **نوشتن (write-intensive)** یا **خواندن (read-intensive)** وابسته است، استفاده از **OverlayFS**ممکن است بهینه نباشد.

Docker NetWork

در داکر، شبکه‌ها برای ارتباط بین کانتینرها و یا ارتباط کانتینرها با دنیای خارج استفاده می‌شوند. به طور پیش‌فرض، داکر ابزارهای مختلفی برای شبکه‌بندی فراهم می‌کند و بسته به نیاز پروژه، می‌توانید شبکه‌های مختلفی ایجاد کنید.

انواع شبکه‌های Docker

Bridge

- **کاربرد:** مناسب برای کانتینرهایی که روی یک هاست (host) اجرا می‌شوند و نیاز به ارتباط با یکدیگر دارند.
- **ویژگی‌ها:**
- کانتینرها در این شبکه می‌توانند از طریق آدرس IP یا نام کانتینر همدیگر را پیدا کنند.
- کانتینرها به اینترنت دسترسی دارند، اما از بیرون قابل‌دسترسی نیستند مگر اینکه **پورت‌ها را منتشر (publish)** کنید.

مثال:

```
docker network create my_bridge_network


docker run -d --name app1 --network my_bridge_network busybox sleep 10000


docker run -d --name app2 --network my_bridge_network busybox sleep 10000
```

می‌توانند همدیگر را پینگ کنند `app2` و `app1`

اما روش دیگری هم وجود دارد ؟ بله

روش دوم:

```
docker container run -d --name app1 --network net-01 busybox sleep 100000

docker container run -d --name app2 --network container:app1 busybox
```

ویژگی‌ها:

1. **`--network net-01`**:
   - در اینجا کانتینر `app1` به یک شبکه از پیش‌ساخته شده با نام `net-01` متصل می‌شود. این شبکه یک شبکه Bridge سفارشی است.
2. **`--network container:app1`**:
   - در اینجا کانتینر `app2` به **شبکه کانتینر** **`app1`** متصل می‌شود.
   - این مدل اتصال به این معناست که `app2` به‌جای عضویت در یک شبکه جداگانه، از **network namespace** کانتینر `app1` استفاده می‌کند.

نتیجه:

- کانتینر `app2` و `app1` دقیقاً از یک namespace شبکه استفاده می‌کنند.
- به این ترتیب، `app2` به IP و تنظیمات شبکه کانتینر `app1` دسترسی خواهد داشت و به‌عنوان یک کانتینر مستقل آدرس شبکه جداگانه ندارد.
- این نوع اتصال معمولاً برای مواردی است که دو کانتینر باید **کاملاً یکسان از نظر شبکه** رفتار کنند (مانند اشتراک پورت‌ها).

Host

شبکه **`host`** یکی از گزینه‌های شبکه‌بندی در Docker است که کانتینر را مستقیماً به شبکه میزبان (هاست) متصل می‌کند. در این حالت، کانتینر از همان آدرس IP و پورت‌های شبکه‌ای هاست استفاده می‌کند، انگار که مستقیماً در هاست در حال اجراست.

- **کاربرد:** زمانی که می‌خواهید کانتینر مستقیماً به شبکه هاست متصل شود و از آدرس IP هاست استفاده کند.

ویژگی‌های کلیدی شبکه `host`:

1. **اتصال مستقیم به شبکه هاست:**
   - کانتینر از همان رابط‌های شبکه‌ای (**network interfaces**) هاست استفاده می‌کند.
   - آدرس IP کانتینر با آدرس IP هاست یکی خواهد بود.
2. **سرعت بالا:**
   - چون ارتباط شبکه‌ای از طریق هاست انجام می‌شود، سربار اضافی شبکه‌بندی Docker وجود ندارد.
3. **عدم ایزوله‌سازی شبکه:**
   - برخلاف شبکه‌های دیگر (مثل bridge)، در اینجا ایزوله‌سازی شبکه‌ای بین کانتینر و هاست وجود ندارد.

- **مثال:**

```
docker run --network host nginx
```

توضیح:

1. **`--network host`:**
   - این گزینه مشخص می‌کند که کانتینر به شبکه `host` متصل شود.
2. **`nginx`:**
   - تصویر (image) سرور وب Nginx که اجرا خواهد شد.

None

شبکه **`none`** در Docker یک نوع شبکه بسیار ساده و ایزوله است که اجازه نمی‌دهد کانتینر به هیچ شبکه‌ای متصل شود. این نوع شبکه برای موارد خاصی کاربرد دارد، مانند تست کردن کانتینرها یا اجرای برنامه‌هایی که نیازی به ارتباط شبکه‌ای ندارند.

ویژگی‌های شبکه `none`:

1. **ایزوله بودن کامل:**
   - در این حالت، کانتینر هیچ دسترسی به اینترنت یا سایر کانتینرها ندارد.
   - تنها یک اینترفیس شبکه لوپ‌بک (**loopback interface**) در کانتینر وجود دارد.
2. **کاربرد برای تست:**
   - این شبکه معمولاً برای تست برنامه‌هایی استفاده می‌شود که نیازی به ارتباط شبکه‌ای ندارند یا باید در محیطی کاملاً ایزوله اجرا شوند.
3. **امنیت بالا:**
   - چون هیچ ارتباط شبکه‌ای وجود ندارد، خطر حملات شبکه‌ای به کانتینر نیز عملاً صفر است.

```
docker run --network none alpine
```

توضیح:

1. `--network none`:
   - مشخص می‌کند که کانتینر به شبکه `none` متصل شود.
2. `alpine`:
   - تصویر (image) پایه‌ای که برای اجرای کانتینر استفاده می‌شود.

Overlay

مفهوم **Overlay Network** یکی از انواع درایورهای شبکه در Docker است که برای ارتباط بین کانتینرهایی که روی چند هاست (Node) مختلف قرار دارند، استفاده می‌شود. این نوع شبکه معمولاً در حالت **Docker Swarm** یا **Kubernetes** برای مدیریت ارتباطات بین هاست‌ها و کانتینرها به کار می‌رود.

- **کاربرد:** برای ارتباط بین کانتینرهایی که روی چند هاست مختلف اجرا می‌شوند (در حالت swarm یا multi-host).

ویژگی‌های کلیدی:

1. **شبکه توزیع‌شده:**
   - Overlay Network ترافیک را از طریق تونل‌های رمزنگاری‌شده بین هاست‌های مختلف ارسال می‌کند.
2. **امنیت:**
   - ترافیک بین Nodeها رمزنگاری می‌شود، که امنیت بالایی برای ارتباطات فراهم می‌کند.
3. **جداسازی شبکه:**
   - هر شبکه Overlay یک فضای شبکه جداگانه برای سرویس‌ها ایجاد می‌کند.
4. **انعطاف‌پذیری:**
   - پشتیبانی از اتصال سرویس‌ها و کانتینرها به شبکه‌های مختلف بدون تغییر در تنظیمات.

- **مثال:**

در دنیای Docker و شبکه، **Endpoint** به نقطه‌ای گفته می‌شود که یک کانتینر در شبکه Docker به آن متصل می‌شود. به زبان ساده‌تر، Endpoint مثل درگاهی است که ارتباط شبکه‌ای یک کانتینر را مدیریت می‌کند.

```
docker network create --driver overlay my_overlay_network
```

`--driver overlay`: مشخص کردن نوع درایور شبکه

Macvlan

شبکه `macvlan` در Docker یکی از پیشرفته‌ترین گزینه‌های شبکه‌بندی است که به کانتینرها اجازه می‌دهد مستقیماً به شبکه فیزیکی متصل شوند و یک آدرس MAC منحصربه‌فرد و جداگانه دریافت کنند. این نوع شبکه‌بندی برای سناریوهایی که نیاز به تعامل مستقیم کانتینرها با سایر دستگاه‌های شبکه دارید، بسیار مفید است.

ویژگی‌های کلیدی شبکه `macvlan`:

1. **اتصال مستقیم به شبکه فیزیکی:**
   - هر کانتینر مانند یک دستگاه مستقل در شبکه عمل می‌کند و یک آدرس MAC و IP جداگانه دارد.
2. **تعامل مستقیم با دستگاه‌های دیگر:**

- کانتینرها می‌توانند بدون واسطه به شبکه اصلی متصل شوند و با سایر دستگاه‌های شبکه ارتباط برقرار کنند.

3. **ایزوله‌سازی بالا:**
   - چون کانتینرها به صورت مستقیم به شبکه متصل می‌شوند، از شبکه Docker ایزوله هستند.

مثال :

```
docker network create \
  --driver macvlan \
  --subnet=192.168.1.0/24 \
  --gateway=192.168.1.1 \
  -o parent=eth0 \
  my_macvlan_network
```

توضیح دستور:

1. **`--driver macvlan`:**
   - مشخص می‌کند که شبکه از نوع `macvlan` است.
2. **`--subnet=192.168.1.0/24`:**
   - محدوده آدرس IP شبکه‌ای که کانتینرها از آن استفاده خواهند کرد.
3. **`--gateway=192.168.1.1`:**
   - آدرس Gateway که کانتینرها برای دسترسی به شبکه‌های خارجی استفاده می‌کنند.
4. **`-o parent=eth0`:**
   - رابط شبکه (network interface) هاست که کانتینرها از آن استفاده می‌کنند.
5. **`my_macvlan_network`:**
   - نام شبکه‌ای که ایجاد می‌شود.

اجرای یک کانتینر در شبکه `macvlan`:

```
docker run --rm --network my_macvlan_network alpine ifconfig
```

توضیح:

- کانتینر به شبکه `my_macvlan_network` متصل می‌شود.
- دستور `ifconfig` برای نمایش آدرس‌های IP و MAC کانتینر اجرا می‌شود.

مفهوم Endpoint در Docker

- وقتی یک کانتینر به یک شبکه Docker متصل می‌شود، یک Endpoint برای آن کانتینر در شبکه ایجاد می‌شود.
- این Endpoint شامل اطلاعاتی مانند آدرس IP، تنظیمات شبکه، و ارتباط با دیگر کانتینرها یا میزبان است.

ساختار Endpoint

از سه بخش اصلی تشکیل شده است:

1. **آدرس IP**: آدرسی که به کانتینر اختصاص داده می‌شود.
2. **پورت‌ها**: پورت‌هایی که کانتینر روی آنها گوش می‌دهد.
3. **کانفیگ‌های شبکه**: تنظیماتی مثل Gateway ،DNS و Subnet.

دستور های اصلی Docker Network

1. مشاهده شبکه‌ها

```
docker network ls
```

2. ایجاد یک شبکه

```
docker network create my_custom_network
```

3. بررسی جزئیات یک شبکه

```
docker network inspect my_custom_network
```

4. حذف یک شبکه

```
docker network rm my_custom_network
```

ساختار شبکه و Endpoint در Docker

در Docker، برای برقراری ارتباط بین کانتینرها و مدیریت جریان داده‌ها، از مفاهیمی مانند **شبکه (Network)** و **نقطه انتهایی (Endpoint)**استفاده می‌شود. این دو مفهوم پایه‌های اساسی ساختار شبکه در Docker هستند.

1. Network (شبکه):

- **تعریف:**
  شبکه در Docker جایی است که کانتینرها در آن قرار می‌گیرند تا بتوانند با یکدیگر ارتباط برقرار کنند. این شبکه‌ها می‌توانند انواع مختلفی داشته باشند که هرکدام ویژگی‌ها و کاربردهای خاص خود را دارند (مثل bridge، host، overlay و غیره).
- **وظایف شبکه در Docker:**
  1. **ایزوله‌سازی:**
     شبکه‌ها به تفکیک و ایزوله‌سازی ارتباطات بین کانتینرها کمک می‌کنند.
     برای مثال، کانتینرهایی که در یک شبکه نیستند، نمی‌توانند با یکدیگر ارتباط برقرار کنند.
  2. **مدیریت آدرس‌دهی:**
     Docker به طور خودکار برای کانتینرها آدرس IP اختصاص می‌دهد.
  3. **ارتباط داخلی:**
     امکان برقراری ارتباط بین کانتینرها از طریق نام یا آدرس IP.
- **مثال از ایجاد شبکه:**

```
docker network create my_network
```

2. Endpoint (نقطه انتهایی):

- **تعریف:**
Endpoint به نقطه اتصال یک کانتینر به یک شبکه گفته می‌شود. زمانی که یک کانتینر به یک شبکه متصل می‌شود، Docker برای آن یک Endpoint ایجاد می‌کند که شامل اطلاعات ضروری برای برقراری ارتباط در شبکه است.
- **ویژگی‌ها:**
  1. **اطلاعات Endpoint:**
     - آدرس IP کانتینر در آن شبکه.
     - نام کانتینر برای دسترسی در شبکه.
     - تنظیمات مربوط به DNS و سایر اطلاعات شبکه.
  2. **ارتباط با شبکه:**
  Endpoint واسطه‌ای بین کانتینر و شبکه است و مدیریت جریان داده‌ها را انجام می‌دهد.
- **مثال از اتصال یک کانتینر به شبکه:**

```
docker network connect my_network my_container
```

- **توضیح:**
در این دستور، کانتینر `my_container` به شبکه `my_network` متصل شده و یک Endpoint در این شبکه برای آن ایجاد می‌شود.

نکات مهم درباره Network و Endpoint:

1. **یک کانتینر می‌تواند به چند شبکه متصل باشد:**
   - هر کانتینر می‌تواند چند Endpoint داشته باشد که هرکدام مربوط به یک شبکه است.
2. **ایزوله‌سازی:**
   - اگر دو کانتینر در شبکه‌های مختلف باشند، به صورت پیش‌فرض نمی‌توانند با یکدیگر ارتباط برقرار کنند، مگر اینکه از تنظیمات خاص (مثل network `host`) استفاده شود.
3. **مدیریت خودکار Docker:**
   - داکر به صورت خودکار Endpointها را مدیریت می‌کند، اما شما می‌توانید آن‌ها را به صورت دستی تنظیم یا حذف کنید.

خلاصه:

- **شبکه (Network):**

محیطی است که کانتینرها می‌توانند در آن با یکدیگر ارتباط برقرار کنند.

- **نقطه انتهایی (Endpoint):**

نقطه اتصال یک کانتینر به یک شبکه است که اطلاعات ضروری مانند آدرس IP و نام کانتینر را شامل می‌شود.

- **کاربرد:**

این ساختار امکان ایجاد ارتباطات ایزوله و قابل مدیریت بین کانتینرها را فراهم می‌کند و پایه‌ای برای راه‌اندازی برنامه‌های چندکانتینری و میکروسرویس‌ها است.

VOTING APP

1. اجزای اصلی پروژه:

- **VOTE (Python):** یک اپلیکیشن نوشته‌شده با پایتون که در پورت `8080` اجرا می‌شود
- **REDIS:** یک پایگاه داده در حافظه برای ذخیره‌سازی اطلاعات به صورت موقت
- **WORKER (DotNet Core):** وظیفه پردازش داده‌ها را برعهده دارد. این سرویس به REDIS و پایگاه داده متصل است.
- **DB (Postgres):** یک پایگاه داده پایدار برای ذخیره‌سازی اطلاعات
- **RESULT (Node):** اپلیکیشنی که نتیجه را نشان می‌دهد و در پورت `8081` اجرا می‌شود

2. ارتباط بین اجزا:

- همه اجزا از طریق یک شبکه مجازی داکر (`net-01`) به هم متصل شده‌اند.
- **VOTE** داده‌ها را به **REDIS** ارسال می‌کند
- **WORKER** داده‌ها را از **REDIS** دریافت کرده و در پایگاه داده **Postgres** ذخیره می‌کند
- **RESULT** داده‌ها را از **Postgres** می‌خواند و به کاربران نمایش می‌دهد

1. ساخت ایمیج‌ها و شبکه:

ابتدا ایمیج‌ها ساخته یا دانلود شده‌اند:

```
docker image build . -t worker:v1.0.0    # ساخت ایمیج برای Worker
docker image pull postgres:15-alpine    # دریافت ایمیج Postgres
docker image pull redis                 # دریافت ایمیج Redis
```

سپس یک شبکه برای ارتباط بین کانتینرها ایجاد شده است:

```
docker network create net-01             # ساخت نتورک
```

2. اجرای کانتینرها:

VOTE (Python):

```
docker container run -d --name vote --network net-01 -p 8080:80
vote:v1.0.0
```

- این کانتینر در شبکه `net-01` اجرا می‌شود.
- پورت `8080` را به پورت `80` کانتینر متصل می‌کند.

 REDIS:

```
docker container run -d --name redis --network net-01 redis:latest
```

- این کانتینر فقط در شبکه `net-01` قرار دارد و به عنوان پایگاه داده موقت استفاده می‌شود.

WORKER (DotNet Core):

```
docker container run -d --name worker --network net-01 worker:v1.0.0
```

- این کانتینر داده‌ها را از REDIS خوانده و به پایگاه داده ارسال می‌کند.

DB (Postgres):

برای پایگاه داده، ابتدا یک ولوم ایجاد شده تا داده‌ها به صورت پایدار ذخیره شوند:

```
docker volume create pg-data
```

سپس کانتینر اجرا شده:

```
docker container run -d --name db --network net-01 -v pg-
data:/var/lib/pgsql/data \
  -e POSTGRES_USER=postgres -e POSTGRES_PASSWORD=postgres postgres:15-
alpine
```

- ولوم `pg-data` برای ذخیره داده‌های پایدار به کار می‌رود.
- متغیرهای محیطی برای کاربر و رمز عبور پایگاه داده تنظیم شده‌اند.

RESULT (Node):

```
docker container run -d --name result --network net-01 -p 8081:80
result:v1.0.0
```

این کانتینر در پورت `8081` اجرا می‌شود و نتایج را نمایش می‌دهد.

نحوه ارتباط کانتینرها:

- همه کانتینرها از طریق شبکه `net-01` به هم متصل شده‌اند.
- ارتباط داخلی بین کانتینرها با استفاده از نام کانتینر (به عنوان hostname) انجام می‌شود. به عنوان مثال، **WORKER** می‌تواند به REDIS با آدرس `redis` دسترسی پیدا کند.

خبر خوب ! شما هیچ وقت از این روش در محیط عملیاتی استفاده نمی کنید!

Docker Compose:

داکر کامپوز ابزاری است که به شما این امکان را می‌دهد تا چندین سرویس Docker را به راحتی در یک فایل (YAML) تعریف کرده و همزمان آن‌ها را اجرا کنید. این ابزار بیشتر برای مدیریت پروژه‌های پیچیده و multi-container استفاده می‌شود. Docker Compose به طرز قابل توجهی فرآیندهای راه‌اندازی و مدیریت کانتینرها را ساده‌تر می‌کند.

ویژگی‌ها و مزایای Docker Compose:

1. **تعریف چندین سرویس**: با Docker Compose، شما می‌توانید چندین کانتینر را در یک فایل YAML تعریف کرده و تمام آن‌ها را با یک دستور `docker-compose up` راه‌اندازی کنید. این کار می‌تواند شامل وب سرور، پایگاه داده، کش (cache) و سایر سرویس‌ها باشد. به این ترتیب، نیازی نیست که برای هر کانتینر به طور جداگانه دستورات `docker run` را وارد کنید.
2. **مدیریت شبکه‌ها و وولوم‌ها**: Docker Compose به شما این امکان را می‌دهد که شبکه‌ها و وولوم‌ها (Volumes) را به راحتی در یک فایل تنظیم کنید. شبکه‌ها به کانتینرها این امکان را می‌دهند که با هم ارتباط برقرار کنند، و وولوم‌ها برای ذخیره‌سازی داده‌ها به طور پایدار در نظر گرفته می‌شوند.
3. **مقیاس‌پذیری**: یکی از ویژگی‌های جالب Docker Compose این است که می‌توانید به راحتی تعداد کانتینرهای یک سرویس را مقیاس‌پذیر کنید. به عنوان مثال، شما می‌توانید تعداد کانتینرهای یک اپلیکیشن وب را با دستور `docker-compose scale` افزایش دهید تا بتوانید بار ترافیکی بیشتری را تحمل کنید.
4. **پیکربندی راحت**: در Docker Compose، تمام تنظیمات از جمله متغیرهای محیطی، پورت‌ها، حجم‌ها و شبکه‌ها در یک فایل YAML جمع‌آوری می‌شود. به همین دلیل، مدیریت پیکربندی‌ها و تغییرات در پروژه‌های بزرگ بسیار ساده‌تر می‌شود.
5. **کار با Docker Swarm و Kubernetes:**
   مفهوم Docker Compose می‌تواند برای اجرای سرویس‌ها در محیط‌های بزرگ‌تر مثل Docker Swarm و Kubernetes استفاده شود. این ابزار به شما کمک می‌کند کانتینرها را به‌سادگی مدیریت و مقیاس‌پذیر کنید.

نحوه استفاده از Docker Compose:

برای استفاده از Docker Compose، باید یک فایل به نام `docker-compose.yml` بسازید که شامل تعاریف و تنظیمات مختلف سرویس‌های شما باشد. در این فایل شما سرویس‌های مختلف مانند اپلیکیشن‌ها، پایگاه داده‌ها، کش‌ها و غیره را می‌توانید تعریف کنید. سپس با استفاده از دستور `docker-compose up` این سرویس‌ها را به صورت همزمان راه‌اندازی کنید.

یک مثال ساده:

```
version: '3'
services:
  web:
    image: nginx:latest
    ports:
      - "80:80"
  db:
    image: mysql:5.7
    environment:
      MYSQL_ROOT_PASSWORD: example
```

```
  app:
    image: node:latest
    build: ./app
    depends_on:
      - db
```

توضیح کد:

```
version: '3'
```

این خط نسخه قالب (schema) فایل Docker Compose را مشخص می‌کند. نسخه 3 یکی از نسخه‌های متداول و پرکاربرد است که امکانات و ویژگی‌های بسیاری ارائه می‌دهد.

2. سرویس‌ها (Services)

```
services:
```

این بخش تمام سرویس‌هایی که در پروژه استفاده می‌شوند را تعریف می‌کند. هر سرویس نشان‌دهنده یک کانتینر است که تنظیمات خاص خودش را دارد.

3. سرویس وب (web)

```
  web:
    image: nginx:latest
    ports:
      - "80:80"
```

- **`web:`**
  نام سرویسی است که تعریف شده و می‌توان از آن به‌عنوان شناسه در شبکه استفاده کرد.
- **`image: nginx:latest`**
  مشخص می‌کند که کانتینر بر اساس ایمیج `nginx:latest` ساخته می‌شود. این ایمیج شامل آخرین نسخه Nginx است.
- **`ports: - "80:80"`**
  این تنظیم، پورت 80 روی سیستم میزبان (host) را به پورت 80 داخل کانتینر متصل می‌کند. بنابراین، اگر در مرورگر آدرس `localhost` را وارد کنید، وب‌سرور Nginx اجرا خواهد شد.

4. سرویس پایگاه داده (db)

```
  db:
    image: mysql:5.7
    environment:
      MYSQL_ROOT_PASSWORD: example
```

- **`db:`**

نام سرویس پایگاه داده است.

- **`image: mysql:5.7`**
  مشخص می‌کند که کانتینر بر اساس ایمیج `mysql:5.7` ساخته می‌شود. این ایمیج حاوی نسخه 5.7 از پایگاه داده MySQL است.
- **`environment:`**
  این بخش متغیرهای محیطی (Environment Variables) را تعریف می‌کند. در اینجا:
- **`MYSQL_ROOT_PASSWORD: example:`**
  رمز عبور کاربر root پایگاه داده برابر با مقدار `example` تنظیم شده است. این تنظیم برای امنیت و دسترسی به MySQL ضروری است.

5. سرویس اپلیکیشن (app)

```
app:
   image: node:latest
   build: ./app
   depends_on:
     - db
```

- **`app:`**
  نام سرویسی است که اپلیکیشن شما را اجرا می‌کند.
- **`image: node:latest`**
  مشخص می‌کند که کانتینر بر اساس ایمیج `node:latest` ساخته می‌شود. این ایمیج شامل آخرین نسخه Node.js است.
- **`build: ./app`**
  این تنظیم به Docker می‌گوید که به جای دانلود ایمیج آماده، ایمیج را از دایرکتوری `app/.` بسازد. بنابراین، در این مسیر باید یک Dockerfile وجود داشته باشد.
- **`depends_on:`**
  این تنظیم مشخص می‌کند که سرویس `app` به سرویس `db` وابسته است. یعنی ابتدا سرویس `db` اجرا می‌شود و سپس نوبت به اجرای سرویس `app` می‌رسد.

خیلی اسان تر VOTING APP

```
version: '3'
services:
  vote:
    image: vote:v1.0.0
    networks:
      - net-01
    ports:
      - "8080:80"
```

```
  redis:
    image: redis:latest
    networks:
      - net-01

  worker:
    image: worker:v1.0.0
    networks:
      - net-01

  db:
    image: postgres:15-alpine
    networks:
      - net-01
    volumes:
      - pg-data:/var/lib/pgsql/data
    environment:
      POSTGRES_USER: postgres
      POSTGRES_PASSWORD: postgres

  result:
    image: result:v1.0.0
    networks:
      - net-01
    ports:
      - "8081:80"

networks:
  net-01:
    driver: bridge

volumes:
  pg-data:
```

برای این کار نیاز به یک فایل yml داریم :

ساختار کلی فایل

یک فایل Docker Compose به زبان YAML نوشته می‌شود و شامل سه بخش اصلی است:

1. **نسخه (Version)**: تعیین می‌کند که از چه استانداردی در Compose استفاده می‌کنید.
2. **سرویس‌ها (Services)**: هر سرویس معادل یک کانتینر است. اینجا مشخص می‌کنید که هر سرویس چه کاری انجام می‌دهد، از چه ایمیجی استفاده می‌کند و چه تنظیماتی دارد.

3. **شبکه‌ها و ولوم‌ها (Networks and Volumes)**: برای اینکه سرویس‌ها بتوانند با هم ارتباط داشته باشند یا داده‌ها را ذخیره کنند، شبکه و ولوم تعریف می‌کنید.
4. نسخه:

```
version: '3'
```

این خط فقط می‌گوید که از نسخه 3 استاندارد Docker Compose استفاده می‌کنیم. نسخه 3 یکی از محبوب‌ترین نسخه‌هاست.

2. تعریف سرویس‌ها:

```
services:
```

این بخش اصلی فایل است که تمام سرویس‌ها را تعریف می‌کنیم. هر سرویس یک بلاک جداگانه دارد.

سرویس 1: vote

```
  vote:
    image: vote:v1.0.0
    networks:
      - net-01
    ports:
      - "8080:80"
```

- `vote:` : اسم سرویس است. این اسم باید یونیک باشد
- `image:` : مشخص می‌کند که از ایمیجی به نام `vote:v1.0.0` برای ساخت کانتینر استفاده کنیم
- `networks:` : این سرویس به شبکه‌ای به نام `net-01` متصل می‌شود
- `ports:` : مشخص می‌کند که **پورت 8080** روی سیستم شما (میزبان) به **پورت 80** داخل کانتینر متصل باشد. یعنی اگر در مرورگر آدرس `localhost:8080` را وارد کنید، به این سرویس دسترسی پیدا می‌کنید

سرویس 2: redis

```
  redis:
    image: redis:latest
    networks:
      - net-01
```

- این سرویس از ایمیج رسمی Redis استفاده می‌کند.
- فقط در شبکه `net-01` فعال است و داده‌ها را به صورت موقت ذخیره می‌کند.

سرویس 3: worker

```
  worker:
    image: worker:v1.0.0
    networks:
      - net-01
```

این سرویس مسئول پردازش داده‌هاست و به Redis و پایگاه داده متصل می‌شود.

سرویس 4: db (دیتابیس)

```
  db:
    image: postgres:15-alpine
    networks:
      - net-01
    volumes:
      - pg-data:/var/lib/pgsql/data
    environment:
      POSTGRES_USER: postgres
      POSTGRES_PASSWORD: postgres
```

- **`image:`** : از ایمیج رسمی PostgreSQL استفاده می‌کند
- **`volumes:`** : از یک ولوم به نام `pg-data` برای ذخیره پایدار داده‌ها استفاده می‌کند:
- **`environment:`** : متغیرهای محیطی مثل نام کاربری و رمز عبور پایگاه داده را تنظیم می‌کند

سرویس 5: result

```
  result:
    image: result:v1.0.0
    networks:
      - net-01
    ports:
      - "8081:80"
```

- این سرویس داده‌ها را از پایگاه داده می‌خواند و به کاربران نمایش می‌دهد.
- پورت 8081 سیستم شما به پورت 80 داخل کانتینر متصل است. یعنی با آدرس `localhost:8081` می‌توانید نتیجه را ببینید.

3. تعریف نتورک ها:

```
networks:
  net-01:
    driver: bridge
```

- این بخش یک شبکه مجازی با نام `net-01` ایجاد می‌کند.
- نوع شبکه `bridge` است که به کانتینرها اجازه می‌دهد با هم ارتباط داشته باشند.

4. تعریف ولوم‌ها:

```
volumes:
  pg-data:
```

یک ولوم به نام `pg-data` ایجاد شده است. این ولوم برای ذخیره‌سازی پایدار داده‌های پایگاه داده استفاده می‌شود. حتی اگر کانتینر حذف شود، داده‌ها باقی می‌مانند.

نکات مهم :

1. **فاصله‌ها در YAML مهم هستند**: هر زیرمجموعه باید با 2 فاصله (یا چند برابر آن) مشخص شود.
2. **شبکه‌ها و ولوم‌ها اختیاری نیستند**: برای پروژه‌های واقعی، استفاده از شبکه و ولوم ضروری است.
3. **سازگاری نام‌ها**: اسم سرویس‌ها، شبکه‌ها و ولوم‌ها باید یکتا باشند.
4. **اجرا با یک دستور**: با اجرای دستور زیر، تمام کانتینرها با تنظیمات بالا به طور همزمان اجرا می‌شوند:

```
docker-compose up
```

اگر بخواهید در پس‌زمینه اجرا شود:

```
docker-compose up -d
```

۱. توضیح مفهوم این پروژه

این پروژه شامل دو سرویس است:

1. **Nginx**: انجینکس به عنوان یک **وب سرور** که درخواست‌های HTTP را مدیریت می‌کند. فایل‌های استاتیک مثل `index.html` و `main.js` را مستقیماً تحویل می‌دهد و درخواست‌های مربوط به فایل‌های PHP را به سرویس PHP-FPM می‌فرستد.
2. **PHP-FPM**: **پردازشگر** **PHP** که درخواست‌های فایل‌های PHP را پردازش کرده و خروجی را به Nginx برمی‌گرداند.

هدف پروژه:

- وقتی آدرس‌هایی مثل `http://example.com/index.html` یا `http://example.com/main.js` را وارد می‌کنی، Nginx مستقیم فایل‌ها را سرو کند.
- وقتی آدرس‌هایی مثل `http://example.com/info.php` را وارد می‌کنی، Nginx درخواست را به PHP-FPM ارسال کند تا فایل پردازش شود.

۲. ساختار پروژه

پروژه از چند بخش اصلی تشکیل شده:

1. **فایل‌های محتوا (Content)**:
   - این فایل‌ها در دایرکتوری‌ای به نام `nginx-content` ذخیره می‌شوند و شامل:
   - `index.html`
   - `info.php`
2. **فایل تنظیمات Nginx**:
   - تنظیمات سرور در دایرکتوری‌ای مثل `nginx-config` ذخیره می‌شود.
3. **Docker Compose**:
   - برای مدیریت سرویس‌های Nginx و PHP-FPM از یک فایل `docker-compose.yml` استفاده می‌کنیم.

۳. فایل‌های مورد نیاز

۱. محتوا (Content)

فایل `index.html` :

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>My Nginx Server</title>
</head>
<body>
    <h1>Welcome to My Nginx Server!</h1>
</body>
</html>
```

فایل `info.php` :

```
<?php
phpinfo();
?>
```

۲. فایل تنظیمات (`default.conf`) Nginx

این فایل مشخص می‌کند Nginx چگونه درخواست‌ها را مدیریت کند.

```
server {
    listen 80;
    server_name localhost;
```

```
    root /usr/share/nginx/html;
    index index.php index.html;

    location / {
        try_files $uri $uri/ =404;
    }

    location ~ \.php$ {
        try_files $uri =404;
        fastcgi_split_path_info ^(.+\.php)(/.+)$;
        include fastcgi_params;
        fastcgi_pass php-fpm:9000;
        fastcgi_index index.php;
        fastcgi_param SCRIPT_FILENAME $document_root$fastcgi_script_name;
    }
}
```

این تنظیمات:

- درخواست‌های فایل‌های استاتیک (مانند `index.html`) را مستقیماً پاسخ می‌دهد.
- درخواست‌های PHP (مانند `info.php`) را به PHP-FPM می‌فرستد.

۳. فایل Docker Compose

این فایل برای مدیریت سرویس‌های Nginx و PHP-FPM به کار می‌رود:

```
version: '3.8'

services:
  nginx:
    container_name: nginx
    image: nginx:latest
    restart: always
    ports:
      - "80:80"
    networks:
      - myapp
    volumes:
      - nginx-content:/usr/share/nginx/html
      - nginx-config:/etc/nginx/conf.d
      - nginx-log:/var/log/nginx
```

```
  php-fpm:
    container_name: php-fpm
    image: php:8.3-fpm
    restart: always
    networks:
      - myapp
    volumes:
      - nginx-content:/usr/share/nginx/html

networks:
  myapp:

volumes:
  nginx-content:
  nginx-config:
  nginx-log:
```

توضیحات :

سرویس `nginx`

```
  nginx:
    container_name: nginx
    image: nginx:latest
    restart: always
    ports:
      - "80:80"
    networks:
      - myapp
    volumes:
      - nginx-content:/usr/share/nginx/html
      - nginx-config:/etc/nginx/conf.d
      - nginx-log:/var/log/nginx
```

توضیح بخش‌ها:

سرویس `nginx`

```
  nginx:
    container_name: nginx
    image: nginx:latest
    restart: always
    ports:
```

```
    - "80:80"
  networks:
    - myapp
  volumes:
    - nginx-content:/usr/share/nginx/html
    - nginx-config:/etc/nginx/conf.d
    - nginx-log:/var/log/nginx
```

1. **`container_name: nginx`**
   - نامی که برای کانتینر Nginx انتخاب شده است. این نام در هنگام اجرای کانتینر در `docker ps` نمایش داده می‌شود.
2. **`image: nginx:latest`**
   - تصویر Docker که برای این سرویس استفاده می‌شود. اینجا از آخرین نسخه Nginx استفاده شده است.
3. **`restart: always`**
   - اگر کانتینر به هر دلیلی متوقف شود، به صورت خودکار دوباره اجرا می‌شود.
4. **`ports:`**
   - پورت 80 روی سیستم میزبان به پورت 80 داخل کانتینر متصل شده است. به این معنی که وقتی مرورگر به `http://localhost` متصل شود، درخواست به کانتینر Nginx فرستاده می‌شود.
5. **`networks:`**
   - این سرویس روی شبکه‌ای به نام `myapp` قرار می‌گیرد.
6. **`volumes:`**
   - **`nginx-content:/usr/share/nginx/html`** : فایل‌های استاتیک و PHP که Nginx ارائه می‌دهد از این دایرکتوری بارگذاری می‌شوند.
   - **`nginx-config:/etc/nginx/conf.d`** : در اینجا فایل تنظیمات Nginx (مثل `default.conf`) قرار دارد.
   - **`nginx-log:/var/log/nginx`** : لاگ‌های تولیدشده توسط Nginx در اینجا ذخیره می‌شوند.

سرویس `php-fpm`

```
php-fpm:
  container_name: php-fpm
  image: php:8.3-fpm
  restart: always
  networks:
    - myapp
  volumes:
    - nginx-content:/usr/share/nginx/html
```

توضیح بخش‌ها:

1. **`container_name: php-fpm`**

- نام کانتینر پردازشگر PHP.

2. `image: php:8.3-fpm`
   - ایمیج Docker که برای پردازشگر PHP استفاده می‌شود. اینجا از نسخه PHP 8.3 با حالت FPM استفاده شده است.
3. `restart: always`
   - این سرویس نیز مانند Nginx در صورت توقف، دوباره راه‌اندازی می‌شود.
4. `networks:`
   - این سرویس هم روی شبکه `myapp` قرار دارد و می‌تواند با Nginx ارتباط برقرار کند.
5. `volumes:`

   `nginx-content:/usr/share/nginx/html` : Nginx اینجا فایل‌های PHP را از همان دایرکتوری که استفاده می‌کند بارگذاری می‌کند.

Networks:

```
networks:
  myapp:
```

شبکه‌ای به نام myapp تعریف شده است. این شبکه امکان ارتباط داخلی (بدون نیاز به اینترنت) بین سرویس‌های nginx و php-fpm را فراهم می‌کند.

Volumes:

```
volumes:
  nginx-content:
  nginx-config:
  nginx-log:
```

این بخش حجم‌های مشترک (Volumes) بین سرویس‌ها را تعریف می‌کند:

1. `nginx-content` :
   - برای ذخیره فایل‌های پروژه (HTML، PHP).
2. `nginx-config` :
   - برای ذخیره تنظیمات Nginx (مثل `default.conf`).
3. `nginx-log` :
   - برای ذخیره لاگ‌های تولیدشده توسط Nginx.

۳. نکات مهم

- هر دویه Nginx و PHP-FPM از طریق شبکه `myapp` با هم ارتباط برقرار می‌کنند.
- فایل‌های HTML و PHP در حجم `nginx-content` ذخیره می‌شوند، بنابراین هر تغییری در فایل‌های محلی به کانتینرها منتقل می‌شود.

- فایل تنظیمات Nginx از طریق حجم `nginx-config` به کانتینر Nginx متصل است.

Ansible

انسیبل یه ابزار خیلی ساده و کاربردیه که کمک می‌کنه سرورها و سیستم‌هات رو راحت‌تر مدیریت کنی. به جای اینکه مجبور باشی یه سری کارهای تکراری مثل نصب برنامه‌ها، پیکربندی سرورها یا راه‌اندازی سرویس‌ها رو به صورت دستی انجام بدی، می‌تونی این کارها رو با Ansible به صورت خودکار انجام بدی.

خیلی ساده بگم:

- **چرا استفاده می‌کنیم؟**
  فرض کن باید روی ۱۰ تا سرور یه نرم‌افزار رو نصب کنی. به جای اینکه بری دونه‌دونه این کارو انجام بدی، با Ansible یه دستور می‌نویسی و همه کارها رو خودش انجام می‌ده.
- **چطور کار می‌کنه؟**
  نیازی نداری روی سرورهات چیزی نصب کنی، فقط باید دسترسی SSH داشته باشی. بعدش با فایل‌های ساده‌ای که به زبان YAML نوشته می‌شن، هر کاری که بخوای رو بهش می‌گی.
- **به درد چی می‌خوره؟**
  هر جا که بخوای چیزی رو خودکار کنی؛ مثل نصب برنامه، کانفیگ سرورها، هماهنگ کردن سرویس‌ها، یا حتی آپدیت کردن سیستم‌ها.

Ansible چگونه کار می‌کند؟

1. **Control Node:**
   - سیستمی که Ansible روی آن نصب شده و وظیفه مدیریت بقیه سرورها را دارد.
2. **ماشین‌های مقصد (Managed Nodes):**
   - سرورهایی که قرار است توسط Ansible مدیریت شوند.
3. **Playbook:**
   - فایل‌های YAML که دستورات و تنظیمات در آن نوشته می‌شوند.
4. **(Modules):**
   - واحدهای از پیش‌ساخته‌ای که وظایف مختلفی مثل نصب برنامه، مدیریت فایل‌ها و سرویس‌ها را انجام می‌دهند.

Inventory (فهرست سیستم‌ها)

- **چی هست؟**
  اینونتوری یه فایل یا آدرس هست که توش لیست تمام سرورهایی که می‌خواید با Ansible مدیریت کنید، نگه‌داری می‌شه.
  - مثلا:

```
[web]
server1.example.com
server2.example.com
```

```
[db]
db1.example.com
```

Task (وظیفه)

- **چی هست؟**

یه Task یه کارِ مشخص و کوچیکه که Ansible باید انجام بده. هر Task از یه **Module** استفاده می‌کنه.

- مثلا:

```
- name: کپی کردن فایل به سرور
  copy:
    src: /local/path/file.txt
    dest: /remote/path/file.txt
```

Playbook

- **چی هست؟**

پلی بوک یه لیسته از Taskها که می‌خواید Ansible روی سرورها انجام بده. مثلا می‌خواید یه وب‌سرور نصب بشه، فایل‌ها کپی بشن، و سرویس ری‌استارت بشه.

مثال ساده:

```
- name: نصب وب‌سرور
  hosts: web
  tasks:
    - name: install nginx
      apt:
        name: nginx
        state: present

    - name: ری‌استارت سرویس nginx
      service:
        name: nginx
        state: restarted
```

جمع‌بندی:

- **Inventory:** لیست سرورها
- **Module:** ابزار کوچیک برای انجام یه کار خاص
- **Task:** یه کار مشخص که از یه ماژول استفاده می‌کنه
- **Playbook:** نوشته شده YAML ها که توی Taskیه لیست از
- **Desired State == Current State:** هدف نهایی انسیبل اینه که سیستم‌ها همیشه توی وضعیت دلخواهتون باشن

نصب Ansible و یک تمرین

آموزش نصب و استفاده از Ansible با Virtualenv

در این آموزش، به‌صورت مرحله‌به‌مرحله یاد می‌گیریم که چطور Ansible را داخل یک محیط ایزوله (Virtualenv) نصب و تنظیم کنیم و اولین Playbook را اجرا کنیم.

نصب Virtualenv:

```
sudo yum install -y virtualenv
```

ایجاد محیط ایزوله (Virtualenv)

1. ایجاد یک محیط ایزوله با نسخه مشخصی از پایتون:

```
virtualenv -p python3.6 venv-01
```

وارد شدن به دایرکتوری محیط ایزوله:

```
cd venv-01
```

فعال‌سازی محیط ایزوله:

```
source bin/activate
```

نصب Ansible

نصب Ansible در محیط ایزوله:

```
pip install ansible
```

بررسی نسخه نصب‌شده:

```
ansible --version
```

تنظیم فایل پیکربندی Ansible

1. ایجاد یا ویرایش فایل `ansible.cfg` :

```
vim ansible.cfg
```

افزودن تنظیمات پایه:

```
[defaults]
inventory = ./hosts.yaml
host_key_checking = False
```

تنظیم فایل Inventory

1. ایجاد فایل `hosts.yaml` :

```
vim hosts.yaml
```

افزودن سرورهای هدف:

`

```
all:
  hosts:
    server1:
      ansible_host: 192.168.69.54
      ansible_user: root
      ansible_ssh_private_key_file: ~/.ssh/id_rsa
    server2:
      ansible_host: 192.168.69.55
      ansible_user: root
      ansible_ssh_private_key_file: ~/.ssh/id_rsa
```

ساختار کلی فایل

1. بخش `all:`

- این بخش نشان‌دهنده گروهی از سرورها است که در دسته‌بندی خاصی قرار دارند.
- در این مثال، همه سرورها در گروه **`all`** قرار گرفته‌اند.
- گروه‌بندی به ما امکان می‌دهد دستورات یا Playbookها را برای گروهی خاص از سرورها اجرا کنیم.

2. بخش `hosts:`

- زیرمجموعه‌ای از گروه **all** است و شامل لیستی از سرورها (هاست‌ها) است.
- هر سرور با یک نام مشخص (مثل `server1` و `server2`) تعریف شده است.
- این نام‌ها صرفاً برای شناسایی سرورها در Playbookها و دستورات استفاده می‌شوند.

3. مشخصات هر سرور

- هر سرور دارای ویژگی‌های خاصی است که Ansible از آن‌ها برای اتصال و مدیریت استفاده می‌کند.

ویژگی‌های سرور:

1. **`ansible_host:`**
   - آدرس IP یا نام دامنه سرور.
   - در اینجا، `192.168.69.54` و `192.168.69.55` آدرس‌های IP سرورها هستند.
2. **`ansible_user:`**
   - نام کاربری که Ansible برای اتصال به سرور استفاده می‌کند.
   - در اینجا از کاربر `root` استفاده شده است.
3. **`ansible_ssh_private_key_file:`**
   - مسیر فایل کلید خصوصی SSH که برای احراز هویت استفاده می‌شود.
   - در اینجا، کلید خصوصی در مسیر `ssh/id_rsa./~` قرار دارد.

کلیدهای SSH یک روش امن و ساده برای احراز هویت بین سیستم‌ها هستند. این کلیدها شامل یک کلید عمومی (که روی سرور ذخیره می‌شود) و یک کلید خصوصی (که روی سیستم شما می‌ماند) هستند. با استفاده از این روش، دیگر نیازی به وارد کردن مکرر رمز عبور در زمان اتصال به سرور نیست. در Ansible، کلید خصوصی در فایل Inventory مشخص می‌شود تا بتواند به صورت خودکار و بدون توقف به سرورها متصل شود. کاربرد اصلی این روش، ساده‌تر و سریع‌تر کردن مدیریت سرورهاست.

تست اتصال به سرورها

پینگ کردن سرورها:

```
ansible -i hosts.yaml all -m ping
```

ایجاد اولین Playbook

ایجاد فایل Playbook:

```
vim install_nginx.yml
```

دستورات:

```
---
name: install and configure webservers
hosts: all
become: True
tasks:
  - name: add nginx repo
    copy:
      src: files/nginx.repo
      dest: /etc/yum.repos.d/nginx.repo
```

```
  - name: install nginx
    yum:
      name: nginx
      state: present

  - name: start nginx service
    service:
      name: nginx
      state: restarted
```

نکته:

مطمئن بشید که فایل `nginx.repo` در مسیر `/files` موجود باشد.

```
name: install and configure webservers
hosts: all
become: True
```

- **`name`** : توضیح کلی درباره Playbook. اینجا هدف، نصب و پیکربندی سرورهای وب است
- **`hosts`** : گروهی از سرورها که این Playbook روی آنها اجرا خواهد شد. مقدار `all` به این معنی است که روی تمام سرورهای موجود در فایل Inventory اجرا می‌شود
- **`become`** : اجازه استفاده از دسترسی ریشه (root) را فعال می‌کند. این برای اجرای دستورات سیستمی ضروری است

۲. تعریف وظایف (Tasks):

این بخش شامل فهرستی از کارهایی است که باید انجام شود. هر کار به ترتیب اجرا می‌شود.

افزودن مخزن (Repository) NGINX :

```
- name: add nginx repo
  copy:
    src: files/nginx.repo
    dest: /etc/yum.repos.d/nginx.repo
```

- **`name`** : توضیح مختصری درباره این وظیفه. در اینجا، افزودن فایل مخزن NGINX
- **`copy`** : توضیح مختصری درباره این وظیفه. در اینجا، افزودن فایل مخزن NGINX
  - **`src`** : مسیر فایل مخزن NGINX که روی سیستم قرار دارد (مسیر نسبی)
  - **`dest`** : مسیر مقصد که فایل باید روی سرور هدف کپی شود

اجرای Playbook:

```
ansible-playbook -i hosts.yaml install_nginx.yml
```

نصب NGINX:

```
- name: install nginx
  yum:
    name: nginx
    state: present
```

- `name` : توضیح وظیفه. اینجا هدف نصب نرم‌افزار NGINX است
- `yum` : ماژولی برای مدیریت بسته‌ها در سیستم‌های مبتنی بر RHEL/CentOS
  - `name` : نام بسته‌ای که باید نصب شود. اینجا nginx
  - `state`: وضعیت هدف. مقدار present به این معنی است که اگر NGINX نصب نشده باشد، آن را نصب کند.

راه‌اندازی سرویس NGINX:

```
- name: start nginx service
  service:
    name: nginx
    state: restarted
```

- `name` : توضیح وظیفه. اینجا هدف، راه‌اندازی مجدد سرویس NGINX است
- `service` : ماژولی برای مدیریت سرویس‌ها
  - `name` : نام سرویس که باید کنترل شود. اینجا nginx
  - `state` : وضعیت هدف. مقدار restarted به این معنی است که سرویس باید در صورت اجرا یا عدم state: اجرا، دوباره شروع شود.