Docker compose suite

Services

On a les elements suivants

a) image: Specifies the Docker image to use. b) build: Specifies the path to the Dockerfile if you're building a custom image. c) ports: Maps container ports to host ports. d) volumes: Mounts paths or named volumes. e) environment: Sets environment variables. f) depends_on: Expresses dependency between services. g) restart: Defines the restart policy.

Services build

Dans l'exemple precedent on est partie des images disponibles sur docker hub.

On peut aussi construire une des image nécessaires a l'application avec un Dockerfile

Dans ce cas on utilise `build` a la place de `image` pour le service concerné

YAML

```yaml
version: '3'
services:
  web:
    build:
      context: ./app
      dockerfile: Dockerfile
    ports:
      - "8000:8000"
    volumes:
      - ./app:/app

  database:
    image: postgres:13
    environment:
      - POSTGRES_DB=myapp
      - POSTGRES_USER=user
      - POSTGRES_PASSWORD=password
```

Docker compose workflow

1. Using `docker compose up` with a service that uses `build`:

When you run `docker compose up` and your `compose.yaml` file includes a service that uses

`build` instead of `image`, Docker Compose will automatically build the image if it doesn't already exist. This means you don't necessarily need to run `docker compose build` separately before `docker compose up`.

1. When `docker compose build` is required:

You generally need to run `docker compose build` explicitly in the following scenarios:

a) When you've made changes to your Dockerfile or source code: If you've updated your Dockerfile or the source code that gets built into the image, running `docker compose build` will rebuild the image with the latest changes.

b) To force a fresh build: Even if you haven't made changes, you might want to ensure you have the latest versions of base images or dependencies.

c) When using build arguments: If you're passing build-time variables using `--build-arg`.

d) To build images without starting containers: If you want to pre-build images without starting the services.

e) When using the `--no-cache` option: To build images from scratch, ignoring any cached layers.

1. Best practices:

- During development, it's often convenient to use `docker compose up --build`. This flag tells Docker Compose to build (or rebuild) images before starting the containers.
- In a CI/CD pipeline, it's common to run `docker compose build` explicitly before `docker compose up` to have more control over the build process and to separate building from running.
- If you're not sure whether you need to rebuild, it's generally safe to run `docker compose build`. If no changes are detected, it will use the cached layers and complete quickly.

Remember, Docker uses a caching mechanism during builds. If there are no changes in the Dockerfile or the files it copies into the image, `docker compose build` will be very fast as it will use the cached layers.

---

restart

The `restart` option in a Docker Compose file's services section is used to specify the restart policy for a container. It determines how the container should behave if it stops or crashes.

Role:

- Ensures service reliability by automatically restarting containers under certain conditions.
- Helps maintain the desired state of your application, especially in production environments.

Usage: The `restart` option can take several values:

1. `no` (default): The container will not restart automatically under any circumstance.

2. `always`: The container will always restart, regardless of the exit status.
3. `on-failure`: The container will restart only if it exits with a non-zero status code.
4. `unless-stopped`: Similar to `always`, but won't restart if the container was stopped manually.

Example usage in a Docker Compose file:

```yaml
services:
  web:
    image: nginx
    restart: always

  database:
    image: postgres
    restart: on-failure
```

In this example, the `web` service will always restart, while the `database` service will only restart if it fails.

The `restart` policy in a Docker Compose file does not override the `docker compose down` command.

When you run `docker compose down`, it explicitly stops and removes all containers defined in your Docker Compose file, regardless of their restart policies. The `restart` policy only applies when:

1. The Docker daemon is running
2. The containers are supposed to be up (i.e., after `docker compose up` and before `docker compose down`)

Here's a breakdown of what happens:

1. During normal operation:
    - The restart policy manages container restarts if they crash or stop unexpectedly.
2. When you run `docker compose down`:
    - All containers are stopped and removed.
    - The restart policy is not applied.
    - Resources defined in the Compose file (like networks and volumes) are also removed by default.
3. After `docker compose down`:
    - Even if a container had `restart: always`, it won't restart because the container itself has been removed.

If you want to stop containers temporarily without removing them, you can use `docker compose stop` instead. In this case, containers with `restart: always` will attempt to restart, but the Docker

daemon will prevent this while they're in the stopped state.

To summarize, `docker compose down` takes precedence over the restart policy, allowing you to fully shut down your application stack when needed.

Network

You're correct that Docker Compose sets up a default network automatically. Let's explore when you might need to manually specify networks:

Reasons to manually specify networks:

1. Multiple isolated network groups: If you need to create separate network segments for different parts of your application (e.g., frontend, backend, database).

   ```yaml
   networks:
    frontend:
    backend:

   services:
    web:
      networks: [frontend]
    api:
      networks: [frontend, backend]
    db:
      networks: [backend]
   ```

2. Connecting to external networks: When you need to connect to pre-existing networks or networks managed outside of the Compose file.

   ```yaml
   networks:
    existing_network:
      external: true
   ```

3. Custom network configurations: When you need specific network settings like custom subnets, IP ranges, or network drivers.

   ```yaml
   networks:
    custom_net:
      driver: bridge
      ipam:
        config:
          - subnet: 172.28.0.0/16
   ```

4. Service discovery and DNS: To set custom aliases for services or control how they're

discovered within the network.

```yaml
services:
 web:
   networks:
     frontend:
       aliases:
         - webapp
```

5. Network-level security: To isolate services that don't need to communicate with each other.
6. Multi-host networking: When deploying across multiple Docker hosts (e.g., in a swarm), you might need to use overlay networks.
7. Performance tuning: In some cases, you might want to use different network drivers or configurations for performance reasons.
8. Connecting to services in other Compose projects: If you need to link services across different Compose files.

The default network is often sufficient for simple applications where all services need to communicate with each other. However, as your application architecture becomes more complex or you have specific networking requirements, manually specifying networks gives you finer control over your container communication and isolation.

---

setting your own IP address for a service

Certainly! The IPAM (IP Address Management) configuration in Docker allows you to specify IP addressing details for your networks, including assigning specific IP addresses to containers. Here's how you can use IPAM to give your container a specific IP:

1. First, you define the network with IPAM configuration in the `networks` section of your Docker Compose file:

```yaml
networks:
  my_network:
    driver: bridge
    ipam:
      config:
        - subnet: 172.20.0.0/16
          gateway: 172.20.0.1
```

1. Then, in your service definition, you can specify the exact IP address you want to assign:

```yaml
services:
  my_service:
    image: my_image
    networks:
      my_network:
        ipv4_address: 172.20.0.5
```

Here's a complete example that puts it all together:

```yaml
version: '3'

networks:
  my_network:
    driver: bridge
    ipam:
      driver: default
      config:
        - subnet: 172.20.0.0/16
          gateway: 172.20.0.1

services:
  service1:
    image: nginx
    networks:
      my_network:
        ipv4_address: 172.20.0.5

  service2:
    image: redis
    networks:
      my_network:
        ipv4_address: 172.20.0.6
```

Key points to remember:

1. The subnet should be large enough to accommodate all your containers.
2. The IP addresses you assign must be within the specified subnet.
3. Avoid conflicts by not assigning the same IP to multiple containers.
4. The gateway IP is typically the first usable IP in the subnet.

You can also use IPAM with other network drivers, like Macvlan:

```yaml
networks:
  macvlan_network:
    driver: macvlan
    driver_opts:
      parent: eth0
    ipam:
      config:
        - subnet: 192.168.1.0/24
          gateway: 192.168.1.1

services:
  my_service:
    image: my_image
    networks:
      macvlan_network:
        ipv4_address: 192.168.1.10
```

Remember that when using Macvlan, the subnet and IPs should match your physical network configuration.

Some additional tips:

1. You can specify IP ranges to restrict which IPs can be assigned:

```yaml
ipam:
  config:
    - subnet: 172.20.0.0/16
      ip_range: 172.20.5.0/24
      gateway: 172.20.0.1
```

1. For IPv6, use the `ipv6_address` key instead of `ipv4_address`.
2. If you're using Docker Swarm, remember that each node needs to have a unique subnet to avoid conflicts.

Using IPAM and specifying IPs can be very useful for scenarios where you need predictable addressing, but it does require more manual management. In many cases, letting Docker handle IP assignment automatically is simpler and sufficient.

Pros and Cons of Named Volumes vs. Bind Mounts

Named Volumes: Pros:

- Managed by Docker, making them more portable
- Can be easily shared between containers
- Support for volume drivers, allowing for more advanced storage solutions

- Better performance in some cases, especially on macOS and Windows

Cons:

- Less visibility into the stored data from the host system
- Requires explicit commands to inspect or manipulate data

Bind Mounts: Pros:

- Direct access to files from the host system
- Ideal for development environments (quick edits, live reloading)
- No need to copy data into a volume

Cons:

- Less portable, as they depend on the host's file structure
- Can pose security risks if not managed carefully
- Performance can be slower, especially on non-Linux systems

1. Managing and Maintaining Named Volumes

```bash
# Create a named volume
docker volume create myvolume

# List all volumes
docker volume ls

# Inspect a volume
docker volume inspect myvolume

# Remove a volume
docker volume rm myvolume

# Remove all unused volumes
docker volume prune

# Backup a volume
docker run --rm -v myvolume:/source -v $(pwd):/backup alpine tar czf /backup/myv

# Restore a volume
docker run --rm -v myvolume:/target -v $(pwd):/backup alpine sh -c "tar xzf /bac
```

These commands help you manage Docker volumes effectively. Remember to use them cautiously, especially when removing volumes.

1. Best Practices for Using Volumes in Docker Compose

a) Use named volumes for persistent data:

```yaml
volumes:
  dbdata:

services:
  db:
    image: postgres
    volumes:
      - dbdata:/var/lib/postgresql/data
```

b) Use bind mounts for development:

```yaml
services:
  web:
    build: .
    volumes:
      - ./src:/app/src
```

c) Use temporary volumes for disposable data:

```yaml
services:
  app:
    image: myapp
    volumes:
      - /tmp/app_cache
```

d) Use volume labels for better organization:

```yaml
volumes:
  dbdata:
    labels:
      - "com.example.description=Database data"
      - "com.example.department=IT/Ops"
```

e) Consider using volume drivers for specific needs (e.g., shared storage, backups)

1. Configuring Volume Drivers

Volume drivers allow you to use different storage backends. Here's an example using the `local` driver with options:

```yaml
volumes:
  dbdata:
    driver: local
    driver_opts:
      type: "nfs"
      o: "addr=10.40.0.199,nolock,soft,rw"
      device: ":/docker/example"
```

For cloud deployments, you might use cloud-specific drivers:

```yaml
volumes:
  dbdata:
    driver: rexray/ebs
    driver_opts:
      size: "20"
```

1. Common Volume Configurations for Different Applications

a) Database (e.g., PostgreSQL):

```yaml
services:
  db:
    image: postgres:13
    volumes:
      - pgdata:/var/lib/postgresql/data
volumes:
  pgdata:
```

b) Web server (e.g., Nginx) with static files:

```yaml
services:
  web:
    image: nginx
    volumes:
      - ./static:/usr/share/nginx/html
      - ./nginx.conf:/etc/nginx/nginx.conf:ro
```

c) Application with config and data volumes:

```yaml
services:
  app:
    image: myapp
    volumes:
      - app_config:/app/config
      - app_data:/app/data
volumes:
  app_config:
  app_data:
```

d) Development environment with hot reloading:

```yaml
services:
  frontend:
    build: ./frontend
    volumes:
      - ./frontend/src:/app/src
      - /app/node_modules
```

These examples cover various scenarios you might encounter when working with volumes in Docker Compose. Remember to adjust the configurations based on your specific needs and always consider security implications when using volumes, especially bind mounts.

Docker Compose distinguishes between named volumes and bind mounts based on the syntax you use in your docker-compose.yml file. Let me explain the key differences:

```yaml
version: '3'

services:
  app:
    image: myapp
    volumes:
      # Named volume
      - mydata:/app/data

      # Bind mount (full path)
      - /host/path:/container/path

      # Bind mount (relative path)
      - ./host/path:/container/path

      # Anonymous volume
      - /container/path

volumes:
  mydata:
```

Here's how Docker Compose interprets these different volume definitions:

1. Named Volumes:
   - Syntax: `<volume_name>:<container_path>`
   - Example: `mydata:/app/data`
   - The volume name (e.g., `mydata`) must be declared in the top-level `volumes` section.
2. Bind Mounts:
   - Syntax: `<host_path>:<container_path>`
   - Examples:
     - Absolute path: `/host/path:/container/path`
     - Relative path: `./host/path:/container/path`
   - The host path starts with a `/` (absolute) or `./` (relative to the compose file).
3. Anonymous Volumes:
   - Syntax: `<container_path>`
   - Example: `/container/path`
   - These are managed by Docker but not easily reusable.

Docker Compose uses these rules to determine the type of mount:

1. If the left side of the colon (`:`) matches a named volume in the `volumes` section, it's a named volume.
2. If the left side starts with a `.` or `/`, it's a bind mount.
3. If there's only a right side (no colon), it's an anonymous volume.

It's worth noting a few additional points:

- You can use long syntax for more explicit configuration:

```yaml
volumes:
  - type: volume
    source: mydata
    target: /app/data

  - type: bind
    source: ./host/path
    target: /container/path
```

- Relative paths in bind mounts are relative to the location of the docker-compose.yml file.
- Named volumes defined in the `volumes` section can have additional configuration options, like drivers or labels.

Understanding these distinctions helps in creating clear and maintainable Docker Compose configurations.