제 목	05장 Docker 컨테이너 관리하기	
상세내용	Docker Compose를 활용해 컨테이너 관리하기	

1. Docker Compose를 사용하는 이유

✓ Docker Copmose란?

여러 개의 Docker 컨테이너들을 하나의 서비스로 정의하고 구성해 하나의 묶음으로 관리할 수 있게 도와주는 툴이다.

✓ Docker Compose를 사용하는 이유

1. 여러 개의 컨테이너를 관리하는 데 용이

   여러 개의 컨테이너로 이루어진 복잡한 애플리케이션을 한 번에 관리할 수 있게 해준다. 여러 컨테이너를 하나의 환경에서 실행하고 관리하는 데 도움이 된다.

2. 복잡한 명령어로 실행시키던 걸 간소화 시킬 수 있음

   이전에 MySQL 이미지를 컨테이너로 실행시킬 때 아래와 같은 명령어를 실행시켰다.

```
$ docker run -e MYSQL_ROOT_PASSWORD=password123 -p 3306:3306 -v c:/Users/jaeseong/Documents/Develop/docker-mysql/mysql_data:/var/lib/mysql -d mysql
```

너무 복잡하지 않은가? Docker Compose를 사용하면 위와 같이 컨테이너를 실행시킬 때마다 복잡한 명령어를 입력하지 않아도 된다. 단순히 **docker compose up** 명령어만 실행시키면 된다

[실습] Docker Compose 전체 흐름 느껴보기 (Nginx 설치 및 실행)

✓ Docker CLI로 컨테이너를 실행시킬 때

```
$ docker run --name webserver -d -p 80:80 nginx
```

✓ Docker Compose로 컨테이너를 실행시킬 때

1. compose.yml 파일 작성하기

compose.yml

```
services:
  my-web-server:
    container_name: webserver
    image: nginx
    ports:
      - 80:80
```

- `services: my-web-server` : Docekr Compose에서 하나의 컨테이너를 **서비스(service)**라고 부른다. 이 옵션은 **서비스에 이름**을 붙이는 기능이다.
- `container_name: web-server` : 컨테이너를 띄울 때 붙이는 별칭이다. CLI에서 `--name web-server` 역할과 동일하다.
- `image: nginx` : 컨테이너를 실행시킬 때 어떤 이미지를 사용할 지 정의하는 명령어이다. `$ docker run [이미지명]`와 동일한 역할이다.
- `ports` : 포트 매핑은 어떻게 할 지를 설정하는 옵션이다. CLI에서`-p 80:80` 역할과 동일하다.

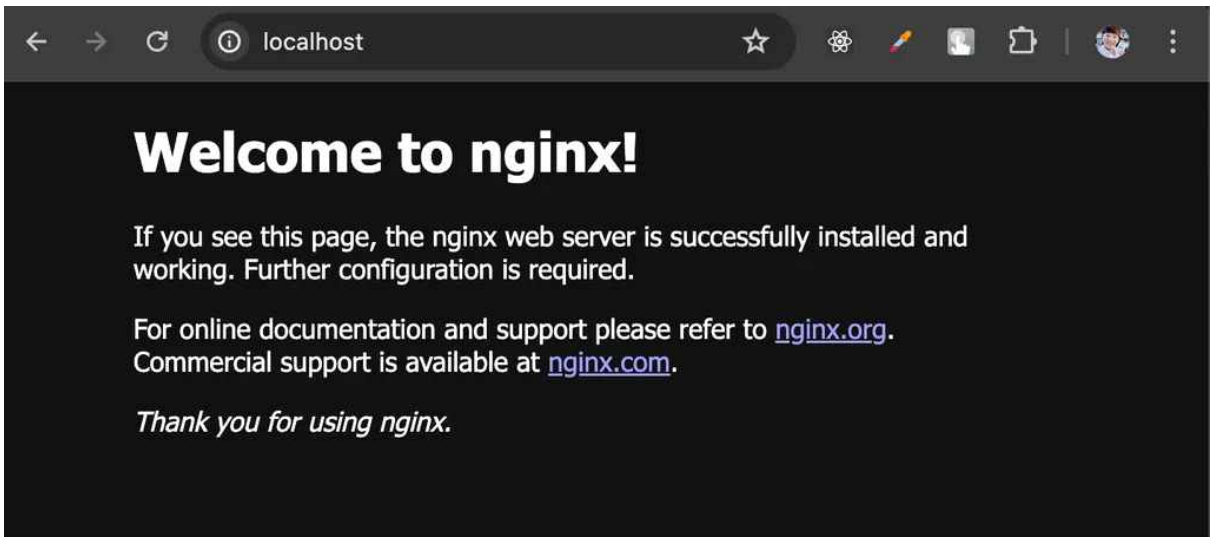
2. compose 파일 실행시키기

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

3. compose 실행 현황 보기

```
$ docker compose ps
$ docker ps
```

4. localhost:80 들어가보기



5. compose 실행 현황 보기

```
$ docker compose down
```

3. 자주 사용하는 Docker Compose CLI 명령어

> docker-compose로 시작하는 명령어는 더 이상 업데이트를 지원하지 않는 Docker Compose의 v1 명령어이므로 되도록이면 사용하지 말자. v2부터는 docker compose로 시작하는 명령어를 사용한다.

▶ 아래 명령어들은 **compose.yml**이 존재하는 디렉토리에서 실행시켜야 한다

✓ Docker CLI로 컨테이너를 실행시킬 때

compose.yml

```
services:
  websever:
    container_name: webserver
        image: nginx
        ports:
            - 80:80
```

✓ compose.yml에서 정의한 컨테이너 실행

```
$ docker compose up     # 포그라운드에서 실행
$ docker compose up -d # 백그라운드에서 실행
```

▶ -d : 백그라운드에서 실행

✓ Docker Compose로 실행시킨 컨테이너 확인하기

```
# compose.yml에 정의된 컨테이너 중 실행 중인 컨테이너만 보여준다.
$ docker compose ps

# compose.yml에 정의된 모든 컨테이너를 보여준다.
$ docker compose ps -a
```

✓ Docker Compose 로그 확인하기

```
# compose.yml에 정의된 모든 컨테이너의 로그를 모아서 출력한다.
$ docker compose logs
```

✓ 컨테이너를 실행하기 전에 이미지 재빌드하기

```
$ docker compose up --build # 포그라운드에서 실행
$ docker compose up --build -d # 백그라운드에서 실행
```

- compose.yml에서 정의한 이미지 파일에서 코드가 변경 됐을 경우, 이미지를 다시 빌드해서 컨테이너를 실행시켜야 코드 변경된 부분이 적용된다. 그러므로 이럴 때에는 --build 옵션을 추가해서 사용해야 한다.

참고 : `docker compose up` vs `docker compose up --build`

- `docker compose up` : 이미지가 없을 때만 빌드해서 컨테이너를 실행시킨다. 이미지가 이미 존재하는 경우 이미지를 빌드하지 않고 컨테이너를 실행시킨다.
- `docker compose up --build` : 이미지가 있건 없건 무조건 빌드를 다시해서 컨테이너를 실행시킨다.

✓ 이미지 다운받기 / 업데이트하기

```
$ docker compose pull
```

- `compose.yml`에서 정의된 이미지를 다운 받거나 업데이트 한다.
  - 로컬 환경에 이미지가 없다면 이미지를 다운 받는다.
  - 로컬 환경에 이미 이미지가 있는데, Dockerhub의 이미지와 다른 이미지일 경우 이미지를 업데이트 한다.

✓ Docker Compose에서 이용한 컨테이너 종료하기

```
$ docker compose down
```

[실습] Docker Compose로 MySQL 실행시키기

✓ Docker CLI로 컨테이너를 실행시킬 때

```
$ docker run -e MYSQL_ROOT_PASSWORD=pwd1234 -p 3306:3306 -v c:/Users/jaeseong/Documents/Develop/docker-mysql/mysql_data:/var/lib/mysql -d mysql
```

✓ Docker CLI로 컨테이너를 실행시킬 때

1. compose 파일 작성하기

compose.yml

```
services:
  my-db:
    image: mysql
    environment:
      MYSQL_ROOT_PASSWORD: pwd1234
    volumes:
      - ./mysql_data:/var/lib/mysql
    ports:
      - 3306:3306
```

- `environment: ...` : CLI에서 `-e MYSQL_ROOT_PASSWORD=password` 역할과 동일하다.
- `volumes: ...` : CLI에서 `-v {호스트 경로}:/var/lib/mysql` 역할과 동일하다.

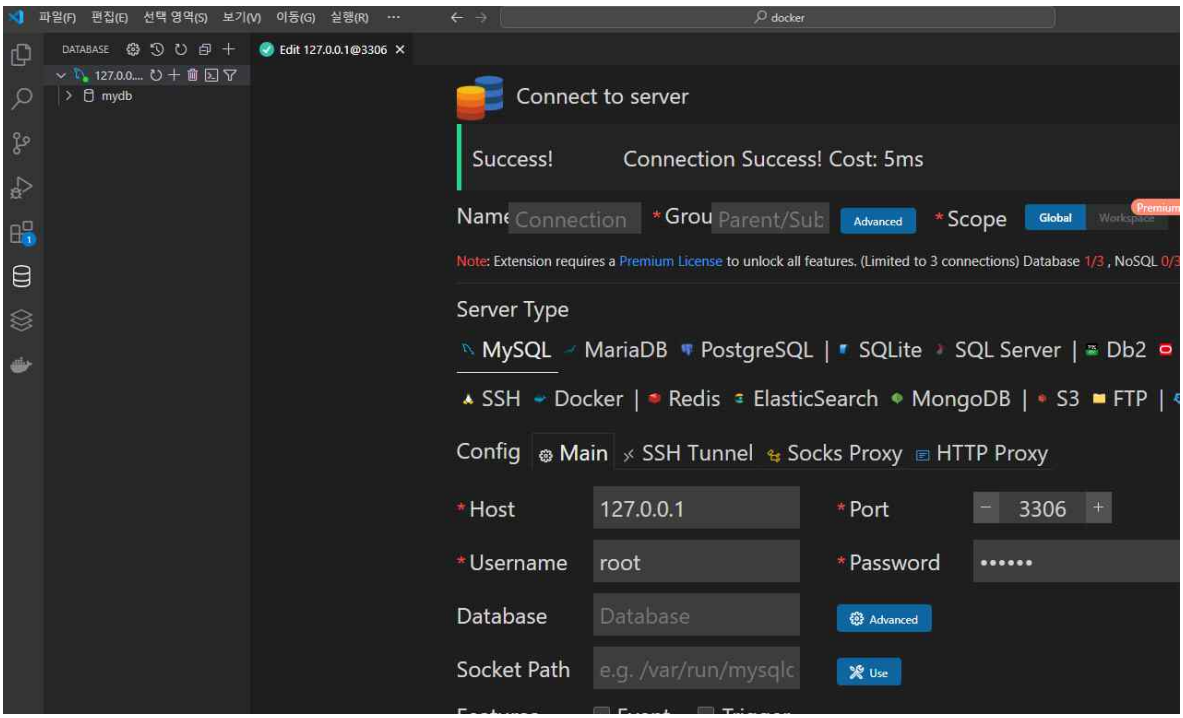
2. compose 파일 작성하기

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

3. compose 실행 현황 보기

```
$ docker compose ps
$ docker ps
```

4. 잘 작동하는 지 DBeaver에 연결시켜보기



5. volume의 경로에 데이터가 저장되고 있는 지 확인하기

6. compose로 실행된 컨테이너 삭제

```
$ docker compose down
```

[실습] Docker Compose로 프론트엔드(HTML, CSS, Nginx) 실행시키기

✓ Docker Compose로 프론트엔드(HTML, CSS, Nginx) 실행시키기

1. HTML, CSS 파일 만들기

index.html

```
<!DOCTYPE html>
<head>
    <meta charset="UTF-8">
    <link rel="stylesheet" href="style.css">
</head>
<body>
    <h1>My Web Page</h1>
</body>
</html>
```

주의) Nginx의 기본 설정에 의하면 메인 페이지(첫 페이지)의 파일명을 index.html 이라고 지어야 한다.

index.html

```
* {
  color: blue;
}
```

2. Dockerfile 작성하기

Dockerfile

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

3. compose 파일 작성하기

▶ **참고) compose를 작성하지 않고 Docker CLI로 실행시킬 때**

```
$ docker build -t my-web-server .
$ docker run -d -p 80:80 my-web-server
```

compose.yml

```
services:
  my-web-server:
    build: .
    ports:
      - 80:80
```

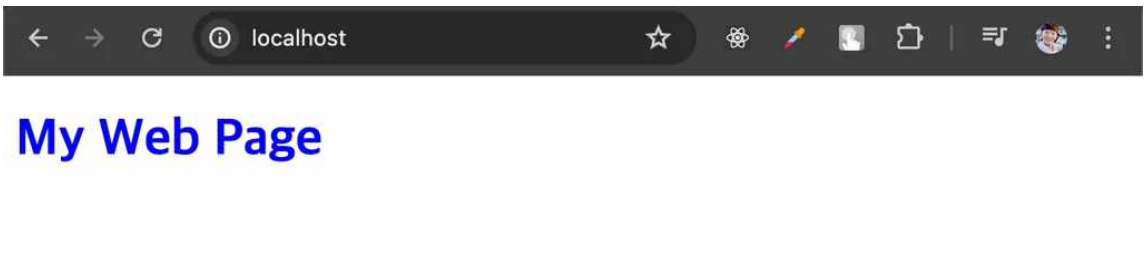
4. compose 파일 실행시키기

```
$ docker compose up -d --build
```

5. compose 실행 현황 보기

```
$ docker compose ps
$ docker ps
```

6. compose 실행 현황 보기



7. compose로 실행된 컨테이너 삭제

```
$ docker compose down
```

4. Docker CLI ↔ Docker Compose 쉽게 작성하기

지금까지의 예제를 보면 Docker CLI로 작성할 수 있는 명령어는 전부 compose.yml 파일로 옮길 수 있다. 반대로 compose.yml에 작성한 모든 값은 Docker CLI로 나타낼 수 있다. 이를 편하게 변환해주는 사이트가 존재한다.

✓ Docker CLI → compose.yml로 변환

참조: https://www.composerize.com/

✓ compose.yml → Docker CLI로 변환

참조: https://www.decomposerize.com/

[실습] MySQL, Redis 컨테이너 동시에 띄워보기

✓ Docker Compose로 MySQL, Redis 실행시키기

1. compose 파일 작성하기

compose.yml

```
services:
  my-db:
    image: mysql
    environment:
      MYSQL_ROOT_PASSWORD: pwd1234
    volumes:
      - ./mysql_data:/var/lib/mysql
    ports:
      - 3306:3306

  my-cache-server:
    image: redis
    ports:
      - 6379:6379
```

주의) YAML 문법에서는 들여쓰기가 중요하다.

2. compose 파일 실행시키기

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

3. compose 실행 현황 보기

```
$ docker compose ps
$ docker ps
```

4. compose 실행 현황 보기

```
$ docker compose ps
$ docker ps
```

4. compose로 실행된 컨테이너 삭제

```
$ docker compose down
```

[실습] Spring Boot, MySQL, Redis 컨테이너 동시에 띄워보기

✓ 1. Spring Boot 프로젝트에 Redis 연결 코드 추가하기

build.gradle

```
…

dependencies {
        …
        implementation  'org.springframework.boot:spring-boot-starter-data-red
is'
}
```

application.yml

```
spring:
  datasource:
    url: jdbc:mysql://my-db:3306/mydb
    username: root
    password: pwd1234
    driver-class-name: com.mysql.cj.jdbc.Driver
  data:
    redis:
      host: localhost
      port: 6379
```

RedisConfig

```
@Configuration
public class RedisConfig {

  @Bean
  public RedisTemplate<String, Object> redisTemplate(RedisConnectionFactory
connectionFactory) {
    RedisTemplate<String, Object> template = new RedisTemplate<>();
    template.setConnectionFactory(connectionFactory);
    template.setKeySerializer(new StringRedisSerializer());
    template.setValueSerializer(new GenericJackson2JsonRedisSerializer());
    return template;
  }
}
```

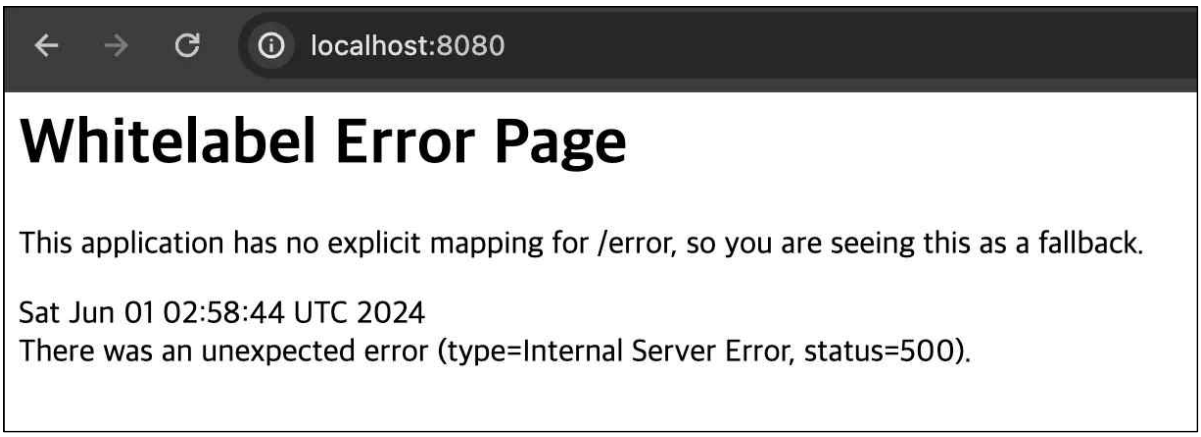
compose.yml

```
services:
  my-server:
    build: .
    ports:
      - 8080:8080
    depends_on:
      my-db:
        condition: service_healthy
      my-cache-server:
        condition: service_healthy
  my-db:
    image: mysql
    environment:
      MYSQL_ROOT_PASSWORD: pwd1234
      MYSQL_DATABASE: mydb
    volumes:
      - c:/docker_data/mysql_data:/var/lib/mysql
    ports:
      - 3306:3306
    healthcheck:
      test: [ "CMD", "mysqladmin", "ping" ]
      interval: 5s
      retries: 10
  my-cache-server:
    image: redis
    ports:
      - 6379:6379
    healthcheck:
      test: [ "CMD", "redis-cli", "ping" ]
      interval: 5s
      retries: 10
```

✓ 2. Docker 컨테이너로 띄워보기

```
$ ./gradlew clean build
$ docker compose down
$ docker compose up --build -d
```

위 명령어를 통해 컨테이너를 띄운 뒤에 localhost:8080으로 요청을 해보면 아래와 같은 에러가 발생한다.



Connection refused 에러가 발생한 이유는 Redis와 연결이 잘 안 됐기 때문이다. 왜 안됐는 지 application.yml 파일을 다시 한 번 살펴보자.

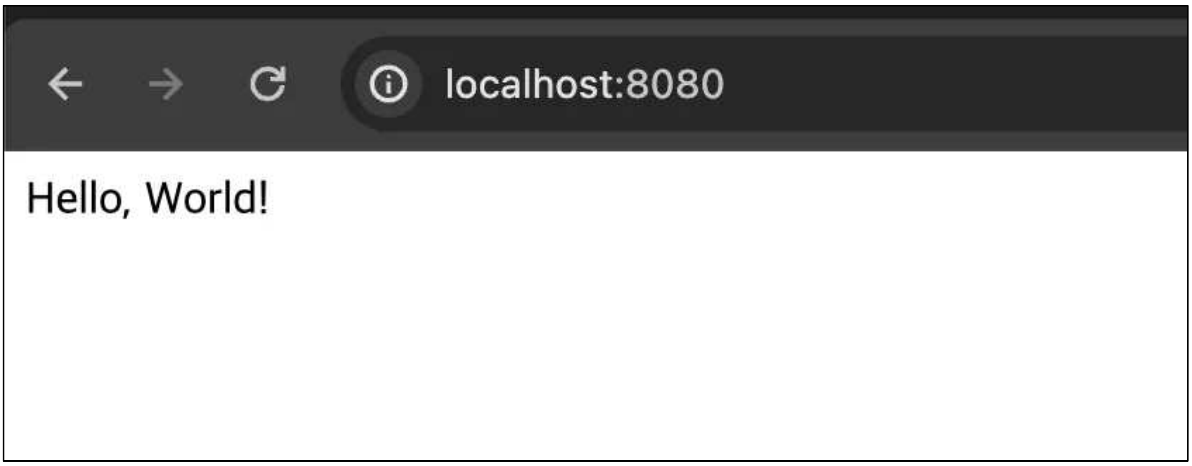
application.yml

```
spring:
  datasource:
    url: jdbc:mysql://my-db:3306/mydb
    username: root
    password: pwd1234
    driver-class-name: com.mysql.cj.jdbc.Driver
  data:
    redis:
      host: localhost
      host: my-cache-server
      port: 6379
```

각 컨테이너는 각자의 네트워크를 가지고 있기 때문에, localhost가 아니라 **Redis가 실행되고 있는 컨테이너로 통신**을 해야 한다. Redis가 실행되고 있는 컨테이너의 주소는 service 이름으로 표현한다고 했다. compose.yml에서 Redis가 실행되고 있는 컨테이너의 service 이름을 my-cache-server라고 이름 붙였다.

위와 같이 코드를 수정한 뒤에 다시 한 번 실행시켜보자.

```
$ ./gradlew clean build
$ docker compose down
$ docker compose up --build -d
```



에러가 발생하지 않고 정상적으로 실행되는 걸 확인할 수 있다.

✓ 그림으로 이해하기