Hands-on Lab

랩용 Docker 컨테이너 설정

Lab 1 Docker 이미지 설정

Lab 환경을 설정하기 위해 3개의 도커 컨테이너를 사용할 것입니다. 3개의 컨테이너는 다음과 같습니다.

- 몽고디비 컨테이너
  - MongoDB 서버가 포함된 컨테이너
  - 빌드의 일부로 설치되는 mongo shell
- 레디스 컨테이너
  - Redis 서버가 포함된 컨테이너
- 애플리케이션 컨테이너 – Node.js 애플리케이션을 실행하기 위한 컨테이너
  - Node.js가 포함된 컨테이너
  - Node.js 패키지 관리자(NPM)가 설치되었습니다.
  - 몽구스 라이브러리가 설치되었습니다.
  - 빠른 웹 액세스를 위해 설치된 Express
  - Redis 클라이언트 라이브러리 설치됨

Docker Compose를 사용하면 공유 네트워크 및 내구성 있는 스토리지로 이러한 3개의 컨테이너를 빠르게 설정할 수 있습니다.

1. 작업 디렉토리 구조 생성

   1.1 다음 디렉토리 구조를 사용하십시오. 다음 섹션에서 파일을 생성합니다. node_modules 디렉토리를 생성할 필요가 없습니다 . 이는 node.js 패키지 관리자인 NPM에 의해 자동으로 생성됩니다. 이 디렉토리는 설치된 NPM 라이브러리를 추적합니다.

2. Docker Compose용 설정 파일

2.1 다음 내용으로 myapp 디렉토리 에 docker- compose.yml을 생성합니다.

YAML 파일은 들여쓰기에 민감합니다. YAML 스타일 콘텐츠에 도움이 되는 편집기를 사용해야 합니다.

```yaml
version: '3'  # Docker Compose file version

services:  # Defines all the services (applications) that make up your project

  app:  # The first service, your Node.js application
    build:  # Instructions for Docker to build your app's Docker image
      context: ./app  # The build context directory, Docker will look for a Dockerfile in this directory
    volumes:  # Specifies the directories that your app container will share with your host
      - ./app:/app  # Mounts the ./app directory on the host to /app in the container
      - /app/node_modules  # Creates an anonymous volume for /app/node_modules in the container
    ports:  # Exposes ports to the host machine
      - "3000:3000"  # Maps port 3000 of the container to port 3000 of the host
    environment:  # Sets environment variables in the container
      - MONGO_URL=mongodb://admin:password@mongo:27017/product-service?authSource=admin  # Connection string for MongoDB
      - REDIS_HOST=redis  # Hostname of the Redis service
    depends_on:  # Declares dependencies on other services
      - mongo  # The app service depends on the mongo service
      - redis  # The app service depends on the redis service

  mongo:  # The second service, your MongoDB database
    build: ./mongo  # Instructions for Docker to build your MongoDB's Docker image
    environment:  # Sets environment variables in the container
      - MONGO_INITDB_ROOT_USERNAME=admin  # The username for the MongoDB root account
      - MONGO_INITDB_ROOT_PASSWORD=password  # The password for the MongoDB root account
    ports:
      - "27017:27017"  # Exposes MongoDB's port to the host machine
    volumes:
      - mongo-data:/data/db  # Mounts the named volume "mongo-data" to /data/db in the container

  redis:  # The third service, your Redis database
    image: redis  # The Docker image to use for the Redis service
    ports:
      - "6379:6379"  # Exposes Redis's port to the host machine

volumes:
  mongo-data:  # Declares a named volume that persists data between container restarts
```

2.2 mongo 디렉토리 아래에 Dockerfile을 생성합니다 . Dockerfile은 도커에게 이미지 빌드 방법을 지시하는 데 사용됩니다. 첫 번째 줄은 시작할 기본 이미지를 나타냅니다. 예를 들어 이 빌드에서는 From mongo 문 을 통해 최신 mongo 이미지를 사용합니다 . 다음 지침 세트는 mongodb 셸을 설치합니다.

```
FROM mongo

RUN apt-get update \
    && apt-get install -y mongodb-org-shell \
    && rm -rf /var/lib/apt/lists/*
```

2.3 앱 디렉터리 아래에 Dockerfile을 만듭니다 . 컨테이너는 현재 LTS(Long Term Su pport) 버전인 Node.js 버전 18로 빌드됩니다. 종속성 라이브러리는 package.jso

n 파일을 검사하여 설치됩니다 . 앱 디렉터리의 모든 파일이 컨테이너에 복사된 다음 npm 이 시작됩니다.

```
# Use the official lightweight Node.js 18 image.
# https://hub.docker.com/_/node
FROM node:18

# Create and change to the app directory.
WORKDIR /app

# Copy application dependency manifests to the container image.
# A wildcard is used to ensure both package.json AND package-lock.json are copied.
# Copying this separately prevents re-running npm install on every code change.
COPY package*.json ./

# Install production dependencies.  Read from package.json for list of dependencies.
RUN npm install

# Copy local code to the container image.
COPY . .

# Start the application. This is the same as running npm start from the command line.
CMD [ "npm", "start" ]
```

2.4 app 디렉토리 아래에 package.json 파일을 생성합니다 . 각 버전 번호 앞의 ^ 기호는 npm 이 지정된 주 버전 다음에 최신 부/패치 버전을 설치함을 의미합니다. 공유 개발 환경에서 이것은 좋은 생각이 아닐 수 있습니다. 일반적으로 동일한 버전을 동기화하고 잠그는 것이 좋습니다.

```
{
  "name": "myapp",
  "version": "1.0.0",
  "description": "",
  "main": "app.js",
  "scripts": {
    "start": "node app.js"
  },
  "dependencies": {
    "express": "^4.17.1",
    "mongoose": "^5.12.13",
    "redis": "^3.1.2"
  }
}
```

2.5 초기 app.js Node.js 프로그램을 만듭니다. 이 간단한 프로그램은 MongoDB 및 Redis에 대한 연결을 생성합니다. 또한 포트 3000에서 RESTful API 액세스를

제공하기 위해 express 인스턴스를 생성합니다. 3개의 단순 경로(REST 엔드포인트)를 생성합니다. / 의 루트 경로는 단순히 환영 메시지로 여러분을 맞이할 것입니다. mongodb 경로는 MongoDB 서버 의 데이터베이스 목록을 제공하여 MongoDB에 대한 연결을 테스트합니다 . 마지막으로 redis 경로는 단순히 이 페이지에 액세스한 횟수에 대한 카운터를 제공합니다. 카운터를 설정하여 그렇게 합니다. 액세스할 때마다 증가하는 키:값 .

필요한 모듈을 설정합니다.

- express - Express 라이브러리는 Node.js용으로 널리 사용되는 웹 애플리케이션 프레임워크입니다. 웹 애플리케이션 및 API 구축을 위한 강력한 기능 및 도구 세트를 제공합니다. Express는 미니멀하고 유연하며 독립적으로 설계되어 개발자가 애플리케이션의 아키텍처 및 디자인을 완전히 제어할 수 있습니다.
- mongoose - Mongoose 라이브러리는 Node.js 및 MongoDB용 ODM(Object-Document Mapping) 라이브러리입니다. JavaScript 개체를 모델링하고 MongoDB 문서에 매핑하기 위한 스키마 기반 솔루션을 제공하여 MongoDB와 상호 작용하는 편리한 방법을 제공합니다.
- redis - Node.js용 Redis 라이브러리는 Redis와 상호 작용할 수 있는 클라이언트 라이브러리입니다. 라이브러리는 문자열, 해시, 목록, 세트, 정렬된 세트 등과 같은 다양한 데이터 구조에 대한 액세스를 허용하며 메모리에 조작하고 저장할 수 있습니다.

```
// Import necessary modules
const express = require('express');
const mongoose = require('mongoose');
const redis = require('redis');

// Instantiate the Express application
const app = express();
```

새 redis 클라이언트 개체를 인스턴스화합니다. node.js에서는 충돌이나 덮어쓰기 가능성을 방지하기 위해 인스턴스화된 클래스를 저장할 때 상수를 사용하는 것이 관례입니다. 우리는 성공적으로 연결되었는지 또는 오류가 있었는지 확인하기 위해 on 메소드를 사용할 것입니다. 콘솔 로그에 적절한 메시지를 보냅니다.

```
// Create and configure the Redis client
const client = redis.createClient({
    host: process.env.REDIS_HOST
});

// Log successful Redis connection
client.on('connect', () => {
    console.log('Connected to Redis');
});

// Log Redis errors
client.on('error', err => {
    console.log('Redis error: ', err);
});
```

mongoose 라이브러리 개체를 사용하여 MongoDB에 연결합니다. 대부분의 MongoDB 클라이언트는 연결 풀링을 사용합니다. 이를 통해 클라이언트는 연결 풀링을 활용할 수 있습니다 . 새 연결을 생성하면 연결 풀이 폴링되고 사용 가능한 경우 재사용됩니다.

```
// Connect to MongoDB
mongoose.connect(process.env.MONGO_URL, { useNewUrlParser: true, useUnifiedTopology: true })
    .then(() => console.log('MongoDB connected...'))
    .catch(err => console.error(err));
```

Express 애플리케이션의 경로는 REST 엔드포인트입니다. 따라서 GET, POST, PUT 및 DELETE 작업은 데이터베이스 CRUD(생성, 읽기, 업데이트, 삭제) 작업을 지원하는 데 가장 자주 사용됩니다. 경로 내에서 끝점에 대한 서비스 논리를 정의합니다. 경로는 콜백 함수를 사용하여 REST 끝점에 대한 요청 및 응답을 처리하는 방법을 정의합니다.

JavaScript에서 콜백 함수는 다른 함수에 인수로 전달되고 나중에 실행 흐름의 특정 지점에서 호출되는 함수입니다. 콜백을 사용하여 Express는 특정 이벤트(예: 들어오는 요청)가 발생할 때 나중에 호출될 함수를 제공하여 애플리케이션의 동작을 정의하는 비동기식 프로그래밍 스타일을 따릅니다. 이를 통해 서버가 여러 요청을 동시에 처리할 수 있는 비차단 및 이벤트 기반 동작이 가능합니다. 다른 함수에 대한 콜백 함수로 메인 함수에서 비동기 작업의 결과로 인수를 받는 경우가 많습니다.

예를 들어, app.get ( )은 들어오는 요청과 클라이언트로 다시 보낼 응답을 나타내는 req (요청의 약자) 및 res (응답의 약자)를 제공합니다 . 이러한 개체는 콜백 함수에 매개 변수로 전달됩니다. (req,res) => {} 표기법 을 화살표 함수 표기법 이라고 합니다 . 이 콜백 함수 내에서 요청을 처리하고 응답을 생성하는 논리를 정의합니다.

앞서 인스턴스화한 Express 애플리케이션 개체인 app 을 사용하여 루트 경로를 정의합니다 . app.get 메소드 의 두 번째 매개변수는 콜백 함수입니다. 루트 경로는 단순히 res 클래스를 사용하여 환영 메시지를 반환합니다.

```
// Define the root route
app.get('/', (req, res) => {
    res.send("Welcome to MongoDB Redis class.");
});
```

redis 에 대한 경로를 정의합니다 . 여기에서 우리 는 Redis 클라이언트의 set 및 get 메서드는 Redis 메모리 내 데이터베이스 서버에서 key:value 쌍을 저장하고 호출합니다 .

Arrow 함수 표기법 대신 일반 function( ){} 표기법을 사용합니다. 둘 다 약간의 차이를 제외하고 동일한 기능을 제공합니다. 화살표 함수 표기법은 ES6(ECMAScript 2015)의 일부로 도입되었으며 일반 함수에 비해 더 간결한 구문을 제공합니다. 그러나 그들은 this. <attribute> 구문 을 사용할 수 없습니다 . 이 기능이 필요하지 않기 때문에 여기에서는 두 표기법 모두 작동합니다.

```
// Define the /redis route
app.get('/redis', (req, res) => {
    // Get the counter value from Redis
    client.get('counter', function(err, reply) {
        let counter;
        // If the counter exists, use its value. Otherwise, start at 1.
        if(reply) {
            counter = reply;
        } else {
            counter = 1;
        }
        // Increment and save the counter back to Redis
        client.set('counter', ++counter);
        // Send the counter value as response
        res.json({counter: counter});
    });
});
```

마지막으로 Express 서버를 시작하고 포트 3000에서 수신 대기합니다.

```
// Define the application's port and start the server
const port = process.env.PORT || 3000;
app.listen(port, () => console.log(`Server running on port ${port}`));
```

3. Docker 컨테이너 구축

3.1 터미널에서 myapp 디렉토리 로 이동합니다 .

3.2 다음 명령을 실행합니다. 이 명령을 실행하기 전에 Docker Desktop을 실행 중인지 확인하십시오.

```
docker-compose up --build
```

4. 애플리케이션 테스트

4.1 브라우저에서 다음 주소로 이동합니다.

```
http://localhost:3000/

http://localhost:3000/mongodb

http://localhost:3000/redis
```

4.2 /redis 엔드포인트 에서 브라우저를 새로 고치고 카운터가 증가하는지 관찰하십시오.