

Cloud Native Project

목차

목차

1. 프로그램 소개

프로젝트 개요

1. 백엔드 서버: Python FastAPI를 이용한 RESTful API 제공
2. 프론트엔드: Node.js - Svelte로 구현된 사용자 인터페이스 제공
3. 데이터베이스 : MySQL
4. Reverse Proxy: Nginx Reverse Proxy

주요 특징

2. 시스템 구성

1. 시스템 아키텍처

- mysql- MySQL 데이터베이스 컨테이너:
- myfastapi - FastAPI 백엔드 컨테이너:
- myfront - Svelte 프론트엔드 컨테이너:
- myproxy - Nginx Reverse Proxy 컨테이너:

2. 소스 파일 구성

- Docker-compose.yaml
- Myfastapi/Dockerfile
- frontend/Dockerfile

3. 데이터 베이스 테이블 구성

1. user 테이블
2. question 테이블
3. answer 테이블
4. question_vote 테이블
5. answer_vote 테이블

3. 통신 절차

1. 클라이언트 요청 흐름
2. 프론트엔드와 백엔드 간 통신
- 2.1 프론트엔드에서 API 요청

요청 절차:

- 2.2 프론트엔드에서의 요청 예시
3. 백엔드 FastAPI의 요청 처리
- 3.1 FastAPI의 요청 처리 구조

요청 처리 절차

- 3.2 CRUD 로직 예시 (question_crud.py)

기능 설명

기능 설명

4. 데이터베이스와의 통신

요청 처리 흐름:

5. Nginx Reverse Proxy 역할

최종 통신 절차 요약

4. 주요 기능

4.1 질문 관리

기능 설명

1. 질문 등록

2. 질문 목록 조회

3. 질문 상세 조회

4. 질문 수정

5. 질문 삭제

6. 질문 추천

4.2 답변 관리

기능 설명

1. 답변 등록

2. 답변 삭제

3. 답변 추천

4.3 사용자 관리 - 세션 관리

기능 설명

1. Svelte의 스토리지를 이용한 세션 처리

Svelte Store를 활용한 상태 관리

2. 회원가입 (UserCreate.svelte)

3. 로그인 (UserLogin.svelte)

4. 로그아웃

5. 인증 처리

스토어를 이용한 인증 상태 확인

API 요청 시 토큰 포함

6. 종합 흐름

5. 구현 화면 캡처

1. 질문 목록 화면

1.1 페이지 동적 처리

2. 회원 가입 화면

3. 로그인 화면

3.1 -로그인 후 질문 목록 화면

4. 질문 등록 화면

5. 질문 상세 화면

5.1 답변 등록 화면

5.2 질문/ 답변 추천

5.3 질문/답변 수정 화면

5.4 질문/ 답변 수정 후 화면

6. 프로그램 설치 및 실행 방법

1. [git clone](#)을 통해 소스파일 다운로드
2. [프론트엔드 포트인 8002 포트 허용](#)
3. [받은 폴더로 이동하여 도커 컴포즈 파일 빌드](#)
4. [DNS 호스트 파일 변경](#)
5. host os에서 <http://cn-gna.com> 접속

7. 개발 과정 문제 해결

1. [Myfastapi 컨테이너가 바로 종료됨](#)

[원인](#)

[해결](#)

2. [myfront 컨테이너가 바로 종료됨](#)

[원인](#)

[해결](#)

1. 프로그램 소개

프로젝트 개요

Docker 컨테이너를 활용하여 웹 기반의 Q&A 시스템을 구현한 것으로 시스템은 다음 네 가지 주요 구성 요소로 이루어져 있음

1. 백엔드 서버: Python FastAPI를 이용한 RESTful API 제공

- 내부적으로 Starlette이라는 비동기 프레임워크를 사용하여 빠른 성능을 보여줌
- Pydantic을 통한 입출력 정의, 입출력 값의 검증과 Swagger를 통한 API 테스트 가능
- SQLAlchemy를 사용하여 ORM(객체 관계 매핑) 구현

2. 프론트엔드: Node.js - Svelte로 구현된 사용자 인터페이스 제공

- **No virtual DOM** - React나 Vue.js와 같은 프레임워크는 가상 DOM을 사용하지만, Svelte는 실제 DOM을 직접 조작. Svelte는 앱을 실행 시점(runtime)에서 해석하지 않고 빌드 시점(build time)에서 Vanilla JavaScript 번들로 컴파일하기 때문에 속도가 빠르고 별도의 라이브러리 배포가 필요 없어 간편함
- **Truly reactive** - 복잡한 상태 관리를 위한 지식이나 추가 라이브러리가 필요 없으며, Svelte는 순수 자바스크립트만으로 반응성 기능을 이해하기 쉽게 구현

3. 데이터베이스 : MySQL

4. Reverse Proxy: Nginx Reverse Proxy

- 클라이언트의 요청을 프론트엔드와 백엔드로 적절히 라우팅
-

주요 특징

- 컨테이너 기반 마이크로서비스 구성: 각 구성 요소를 독립적인 컨테이너로 구성하여 유연성과 확장성을 확보
 - 세션 및 데이터 공유: 컨테이너 간 네트워크와 볼륨 공유를 통해 데이터와 세션을 효율적으로 관리
 - 주제: 사용자가 질문을 등록하고 답변을 관리할 수 있는 간단한 Q&A 게시판
-

2. 시스템 구성

1. 시스템 아키텍처

mysql- MySQL 데이터베이스 컨테이너:

- 데이터를 관리하며, 백엔드 서버와 연결.
- 초기 데이터는 `backup.sql` 로 설정.

myfastapi - FastAPI 백엔드 컨테이너:

- MySQL과 통신하며 RESTful API 제공.
- 데이터 모델 정의 및 CRUD 기능 포함.

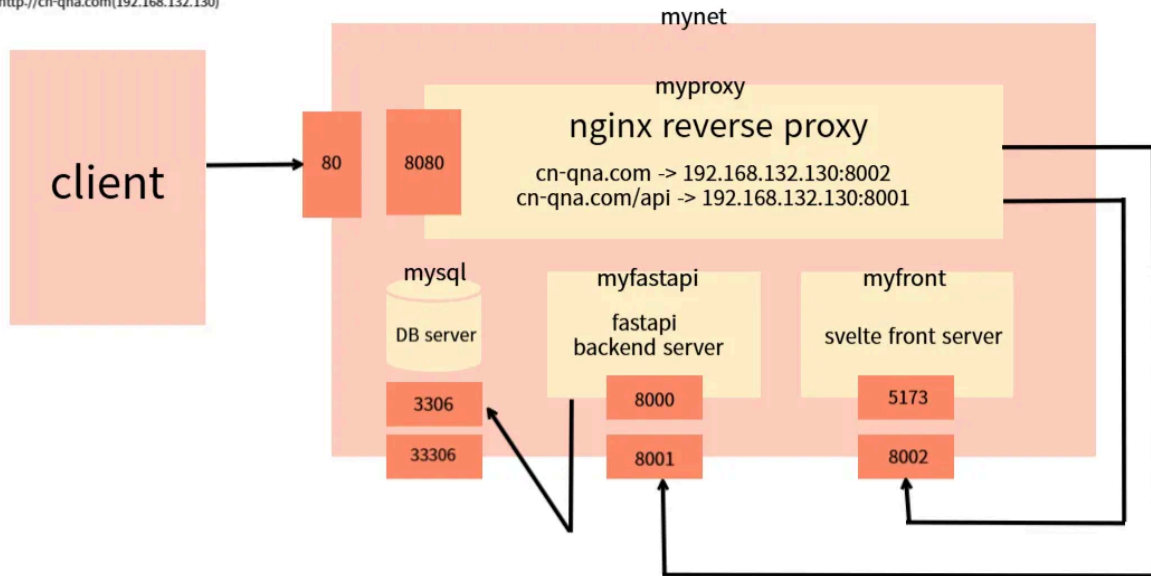
myfront - Svelte 프론트엔드 컨테이너:

- 사용자 인터페이스 제공.
- FastAPI 백엔드와 HTTP 요청을 통해 통신.

myproxy - Nginx Reverse Proxy 컨테이너:

- 클라이언트 요청을 프론트엔드와 백엔드 컨테이너로 적절히 라우팅.
- HTTP 및 HTTPS 요청을 처리하며, WebSocket 지원.

http://cn-qna.com(192.168.132.130)



2. 소스 파일 구성

<https://github.com/tyoon11/fastapi-CN-QandA>

```
fastapi-CN-QandA/
├── db/                    # 데이터베이스 관련 파일
│   └── backup.sql        # 데이터베이스 백업 파일
├── docker-compose.yaml   # Docker Compose 설정 파일
├── frontend/             # 프론트엔드 프로젝트 디렉토리
│   ├── Dockerfile        # 프론트엔드 Dockerfile
│   ├── README.md         # 프로젝트 설명 파일
│   ├── index.html        # 진입점 HTML 파일
│   ├── jsconfig.json     # JavaScript 구성 파일
│   ├── node_modules/     # NPM 종속성 폴더
│   ├── package-lock.json # NPM 종속성 잠금 파일
│   ├── package.json      # NPM 종속성 정의 파일
│   ├── public/           # 정적 파일 폴더
│   │   └── vite.svg      # Vite 관련 정적 파일
│   └── src/              # Svelte 소스 코드
│       ├── App.svelte    # 메인 Svelte 컴포넌트
│       ├── app.css       # 스타일시트 파일
│       ├── assets/       # 애셋 파일 폴더
│       │   └── svelte.svg # Svelte 로고
│       └── components/   # Svelte 컴포넌트
```

```

| | | └─ Error.svelte # 에러 화면 컴포넌트
| | | └─ Navigation.svelte # 내비게이션 컴포넌트
| | └─ lib/ # 상태 관리 및 API 라이브러리
| | | └─ api.js # API 요청 관련 코드
| | | └─ store.js # Svelte 상태 관리 코드
| | └─ main.js # 애플리케이션 진입점
| | └─ routes/ # 라우트 컴포넌트 폴더
| | | └─ AnswerModify.svelte # 답변수정 UI
| | | └─ Detail.svelte # 상세 조회 UI
| | | └─ Home.svelte # 홈 화면 UI
| | | └─ QuestionCreate.svelte # 질문 생성 UI
| | | └─ QuestionModify.svelte # 질문 수정 UI
| | | └─ UserCreate.svelte # 사용자 생성 UI
| | | └─ UserLogin.svelte # 로그인 UI
| | └─ vite-env.d.ts # Vite 환경 타입 선언 파일
| └─ svelte.config.js # Svelte 설정 파일
| └─ vite.config.js # Vite 설정 파일
└─ myfastapi/ # 백엔드 FastAPI 프로젝트 디렉토리
| └─ Dockerfile # FastAPI Dockerfile
| └─ __pycache__/ # Python 캐시 폴더
| └─ database.py # 데이터베이스 초기화 파일
| └─ domain/ # 도메인별 API 모듈
| | └─ answer/ # 답변 관련 파일
| | | └─ answer_crud.py # 답변 CRUD 기능
| | | └─ answer_router.py # 답변 API 라우터
| | | └─ answer_schema.py # 답변 데이터 스키마
| | └─ question/ # 질문 관련 파일
| | | └─ question_crud.py # 질문 CRUD 기능
| | | └─ question_router.py # 질문 API 라우터
| | | └─ question_schema.py # 질문 데이터 스키마
| | └─ user/ # 사용자 관련 파일
| | | └─ user_crud.py # 사용자 CRUD 기능
| | | └─ user_router.py # 사용자 API 라우터
| | | └─ user_schema.py # 사용자 데이터 스키마
| └─ main.py # FastAPI 메인 실행 파일
| └─ models.py # 데이터베이스 모델 정의
| └─ requirements.txt # Python 종속성 파일
└─ nginx/ # Nginx 설정 관련 디렉토리

```

	custom/	# 사용자 정의 설정 파일 폴더
	default.conf	# 기본 Nginx 설정 파일
	requirements.txt	# 프로젝트 종속성 파일

Docker-compose.yaml

- 소스 파일에 이미지 빌드에 필요한 도커 파일, db 파일, nginx reverse proxy 설정 파일, 도커 컴포즈 파일 모두 존재
- 도커 컴포즈 파일에 실행에 필요한 모든 환경을 정의해 놓았기 때문에 사용자는 단순 소스 파일을 받아 도커 컴포즈 파일을 빌드하면 됨

```
version: '3'

services:
  mysql:
    image: mysql:8.0
    container_name: mysql
    environment:
      MYSQL_ROOT_PASSWORD: 123456
      MYSQL_DATABASE: myapp-sql
      MYSQL_USER: myapp-sql
      MYSQL_PASSWORD: 123456
      TZ: "Asia/Seoul" # 한국 시간대 설정
    volumes:
      # 기존의 db파일을 가져와 생성
      - ./db/backup.sql:/docker-entrypoint-initdb.d/init.sql
    ports:
      - "33306:3306" # MySQL 접근 포트
    networks:
      - mynet

  myfastapi:
    build:
      context: ./myfastapi
      dockerfile: Dockerfile
    container_name: myfastapi
    ports:
      - "8001:8000"
```

```
environment:
  - SQLALCHEMY_DATABASE_URL=mysql+pymysql://myapp-sql:123456@n
  - TZ=Asia/Seoul # 한국 시간 설정
volumes:
  - ./myfastapi:/src # 로컬 소스 코드 바인딩
depends_on:
  - mysql
networks:
  - mynet
```

```
myfront:
  build:
    context: ./frontend
    dockerfile: Dockerfile
  container_name: myfront
  ports:
    - "8002:5173" # 프론트엔드 접근 포트
  volumes:
    - ./frontend:/src # 로컬 소스 코드 바인딩
    - node_modules:/src/node_modules # 컨테이너 내부 node_modules를 별도로 볼 수 있도록
  networks:
    - mynet
  depends_on:
    - myfastapi
```

```
nginx-proxy-manager:
  image: jlesage/nginx-proxy-manager
  container_name: myproxy
  ports:
    - "8181:8181" # Nginx Proxy Manager 웹 UI 포트
    - "80:8080" # HTTP 트래픽을 위한 포트
    - "443:4443" # HTTPS 트래픽을 위한 포트
  volumes:
    - /docker/appdata/nginx-proxy-manager:/config # 기본 설정 데이터
    - ./nginx/custom:/config/nginx/custom # 커스텀 설정 디렉토리 - 리버스 프록시
  networks:
    - mynet
  depends_on:
```


- myfastapi
- myfront

networks:

mynet:

driver: bridge

volumes:

node_modules:

Myfastapi/Dockerfile

```
# myfastapi/Dockerfile
FROM python:3.11-alpine
# 작업 디렉토리 설정
WORKDIR /src

# 필요한 라이브러리 설치
COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt

EXPOSE 8000

CMD ["uvicorn", "main:app", "--host", "0.0.0.0", "--port", "8000"]
```

frontend/Dockerfile

```
# frontend/Dockerfile
FROM node:22-alpine

# 작업 디렉토리 설정
WORKDIR /src

# 필요한 패키지 설치
COPY package*.json ./

RUN npm install
```

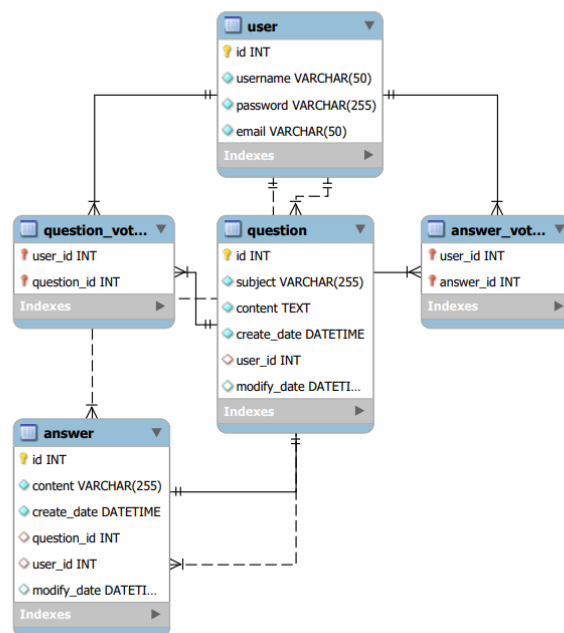
COPY . .

RUN npm run build

EXPOSE 5173

CMD ["npm", "run", "dev", "--", "--host"]

3. 데이터 베이스 테이블 구성



1. **user** 테이블

- **역할**: 사용자 정보를 저장
- **컬럼**:
 - **id** (INT, Primary Key): 각 사용자를 고유하게 식별하기 위한 ID.
 - **username** (VARCHAR(50)): 사용자의 이름.
 - **password** (VARCHAR(255)): 사용자 비밀번호 (해시 처리된 형태).

- `email` (VARCHAR(50)): 사용자 이메일 주소.
 - **관계:**
 - `question` 테이블과 1:N 관계 (한 사용자가 여러 질문 작성 가능).
 - `answer` 테이블과 1:N 관계 (한 사용자가 여러 답변 작성 가능).
 - `question_vote` 와 `answer_vote` 테이블을 통해 질문과 답변에 투표 가능.
-

2. `question` 테이블

- **역할:** 사용자가 작성한 질문 정보를 저장
 - **컬럼:**
 - `id` (INT, Primary Key): 질문 고유 식별 ID.
 - `subject` (VARCHAR(255)): 질문 제목.
 - `content` (TEXT): 질문 내용.
 - `create_date` (DATETIME): 질문 생성 날짜와 시간.
 - `user_id` (INT, Foreign Key): 질문 작성자를 `user` 테이블과 연결.
 - `modify_date` (DATETIME): 질문 수정 날짜와 시간.
 - **관계:**
 - `user` 테이블과 N:1 관계 (여러 질문이 하나의 사용자에게 속함).
 - `answer` 테이블과 1:N 관계 (한 질문에 여러 답변 가능).
 - `question_vote` 테이블과 N:M 관계 (여러 사용자가 질문에 투표 가능).
-

3. `answer` 테이블

- **역할:** 질문에 대한 답변 정보를 저장
- **컬럼:**
 - `id` (INT, Primary Key): 답변 고유 식별 ID.
 - `content` (VARCHAR(255)): 답변 내용.
 - `create_date` (DATETIME): 답변 생성 날짜와 시간.
 - `question_id` (INT, Foreign Key): 답변이 속한 질문을 `question` 테이블과 연결.
 - `user_id` (INT, Foreign Key): 답변 작성자를 `user` 테이블과 연결.

- `modify_date` (DATETIME): 답변 수정 날짜와 시간.
 - 관계:
 - `user` 테이블과 N:1 관계 (여러 답변이 하나의 사용자에게 속함).
 - `question` 테이블과 N:1 관계 (여러 답변이 하나의 질문에 속함).
 - `answer_vote` 테이블과 N:M 관계 (여러 사용자가 답변에 투표 가능).
-

4. `question_vote` 테이블

- 역할: 사용자가 질문에 투표한 정보를 저장
 - 컬럼:
 - `user_id` (INT, Foreign Key): 투표한 사용자 ID.
 - `question_id` (INT, Foreign Key): 투표 대상 질문 ID.
 - 관계:
 - `user` 테이블과 N:1 관계.
 - `question` 테이블과 N:1 관계.
 - 특징:
 - 복합 Primary Key (`user_id` , `question_id`)를 통해 동일 사용자가 같은 질문에 중복 투표하지 않도록 제한.
-

5. `answer_vote` 테이블

- 역할: 사용자가 답변에 투표한 정보를 저장
 - 컬럼:
 - `user_id` (INT, Foreign Key): 투표한 사용자 ID.
 - `answer_id` (INT, Foreign Key): 투표 대상 답변 ID.
 - 관계:
 - `user` 테이블과 N:1 관계.
 - `answer` 테이블과 N:1 관계.
 - 특징:
 - 복합 Primary Key (`user_id` , `answer_id`)를 통해 동일 사용자가 같은 답변에 중복 투표하지 않도록 제한
-

3. 통신 절차

1. 클라이언트 요청 흐름

1.1 브라우저에서 요청

- 사용자가 브라우저에서 `http://<도메인>` 으로 접속.
- 요청이 Nginx Reverse Proxy로 전달되고, Nginx가 요청을 프론트엔드 Svelte 애플리케이션 컨테이너(`myfront`)로 라우팅.
- `index.html` 을 통해 Svelte 애플리케이션이 로드.

2. 프론트엔드와 백엔드 간 통신

Svelte 프론트엔드가 FastAPI 백엔드와 통신하는 방식은 주로 `fetch` API를 통해 이루어짐.

2.1 프론트엔드에서 API 요청

- `api.js` 파일의 `fastapi` 함수가 HTTP 요청을 구성하고 FastAPI 백엔드에 요청을 전송
- `frontend/src/lib/api.js`

```
import qs from "qs"
import { access_token, username, is_login } from "./store"
import { get } from 'svelte/store'
import { push } from 'svelte-spa-router'

// FastAPI 서버 URL을 동적으로 설정
const backendHost = window.location.hostname; // 현재 호스트 이름
const backendPort = "8001"; // FastAPI 서버의 포트
const fastapiBaseUrl = `http://${backendHost}:${backendPort}`;

// fastapi 서버와의 http 요청을 처리하는 함수
const fastapi = (operation, url, params, success_callback, failure_callback)
⇒ {
  let method = operation
  let content_type = 'application/json'
  let body = JSON.stringify(params)

  // 로그인 요청 시 헤더의 Content-Type 및 body 포맷 변경
  if(operation === 'login') {
```

```

    method = 'post'
    content_type = 'application/x-www-form-urlencoded'
    body = qs.stringify(params)
  }

  // 서버 URL 생성
  let _url = fastapiBaseUrl + url;
  if(method === 'get') {
    // GET 요청의 경우 URL에 쿼리 파라미터 추가
    _url += "?" + new URLSearchParams(params)
  }
  // HTTP 요청 옵션 설정
  let options = {
    method: method,
    headers: {
      "Content-Type": content_type
    }
  }

  // Access Token이 있을 경우 Authorization 헤더에 추가
  const _access_token = get(access_token)
  if (_access_token) {
    options.headers["Authorization"] = "Bearer " + _access_token
  }

  // get 이외의 요청은 body를 포함
  if (method !== 'get') {
    options['body'] = body
  }

  // API 요청
  fetch(_url, options)
    .then(response => {
      if(response.status === 204) { // No content
        if(success_callback) {
          success_callback()
        }
      }
      return
    })

```

```

    }
    response.json()
    .then(json => {
      if(response.status >= 200 && response.status < 300) { // 성공
(200 ~ 299)
        if(success_callback) {
          success_callback(json)
        }
      }else if(operation !== 'login' && response.status === 401) { //
인증 실패(401)
        // 인증 토큰 만료 시 처리
        access_token.set('')
        username.set('')
        is_login.set(false)
        alert("로그인이 필요합니다.")
        push('/user-login')
      }else {
        // 실패 콜백 호출 또는 오류 알림
        if (failure_callback) {
          failure_callback(json)
        }else {
          alert(JSON.stringify(json))
        }
      }
    })
  })
  .catch(error => {
    // JSON 응답 처리 중 오류 발생 시
    alert(JSON.stringify(error))
  })
})
}

export default fastapi

```

요청 절차:

1. HTTP 요청 구성:

- `fastapi` 함수는 `operation`, `url`, `params` 를 받아 요청의 메서드(GET, POST 등), URL, 데이터를 설정.
- `VITE_SERVER_URL` 을 통해 백엔드 FastAPI 서버의 URL(`http://<myfastapi>:8000`)을 가져옴.
- 필요한 경우 `Authorization` 헤더에 Access Token을 추가.

2. HTTP 요청 실행:

- `fetch` API를 통해 요청 전송.
- 요청 성공 시 `success_callback` 실행, 실패 시 `failure_callback` 또는 오류 메시지 표시.

2.2 프론트엔드에서의 요청 예시

질문 상세 조회:

```
function get_question() {  
  fastapi("get", "/api/question/detail/" + question_id, {}, (json) => {  
    question = json  
  })  
}
```

- `GET /api/question/detail/` 요청이 FastAPI 백엔드로 전송되어 질문 데이터를 json 형식으로 가져옴

질문 생성:

```
// 질문 등록 함수  
function post_question(event) {  
  event.preventDefault()  
  let url = "/api/question/create"  
  let params = {  
    subject: subject,  
    content: content,  
  }  
  
  // 질문 생성 API 호출  
  fastapi('post', url, params,  
    (json) => {
```



```

        push("/") // 성공 시 메인 페이지로 이동
    },
    (json_error) => {
        error = json_error
    }
)
}

```

- `POST /api/question/create` 요청이 FastAPI로 전달되며 새로운 질문이 생성

3. 백엔드 FastAPI의 요청 처리

3.1 FastAPI의 요청 처리 구조

- FastAPI는 `router` 를 통해 클라이언트의 요청을 처리하며, URL 경로와 관련된 API를 모듈화 하여 관리
- 아래는 주요 라우팅 파일 중 하나인 `question_router.py` 의 예시

```

from fastapi import APIRouter, Depends, HTTPException
from sqlalchemy.orm import Session
from starlette import status

from database import get_db
from domain.question import question_schema, question_crud
from domain.user.user_router import get_current_user
from models import User

# 질문 관련 라우터 생성
router = APIRouter(
    prefix="/api/question",
)

# 질문 목록 조회 API
@router.get("/list", response_model=question_schema.QuestionList)
def question_list(db: Session = Depends(get_db),
                  page: int = 0, size: int = 10, keyword: str = ''):
    # 질문 목록을 페이징과 검색 기능을 포함하여 조회

```

```

total, _question_list = question_crud.get_question_list(
    db, skip=page*size, limit=size, keyword=keyword)
return {
    'total': total,
    'question_list': _question_list
}

# 질문 상세 조회 API
@router.get("/detail/{question_id}", response_model=question_schema.Question)
def question_detail(question_id: int, db: Session = Depends(get_db)):
    question = question_crud.get_question(db, question_id=question_id)
    return question

# 질문 생성 API
@router.post("/create", status_code=status.HTTP_204_NO_CONTENT)
def question_create(_question_create: question_schema.QuestionCreate,
    db: Session = Depends(get_db),
    current_user: User = Depends(get_current_user)):
    question_crud.create_question(db=db, question_create=_question_create,
        user=current_user)

# 질문 수정 API
@router.put("/update", status_code=status.HTTP_204_NO_CONTENT)
def question_update(_question_update: question_schema.QuestionUpdate,
    db: Session = Depends(get_db),
    current_user: User = Depends(get_current_user)):
    db_question = question_crud.get_question(db, question_id = _question_update.question_id)
    if not db_question:
        raise HTTPException(status_code=status.HTTP_400_BAD_REQUEST,
            detail="데이터를 찾을수 없습니다.")
    if current_user.id != db_question.user.id:
        raise HTTPException(status_code=status.HTTP_400_BAD_REQUEST,
            detail='수정 권한이 없습니다.')
    question_crud.update_question(db=db, db_question=db_question,
        question_update=_question_update)

```

```

# 질문 삭제 API
@router.delete('/delete', status_code=status.HTTP_204_NO_CONTENT)
def question_delete(_question_delete: question_schema.QuestionDelete,
                    db: Session = Depends(get_db),
                    current_user: User = Depends(get_current_user)):
    db_question = question_crud.get_question(db, question_id=_question_delete.question_id)
    if not db_question:
        raise HTTPException(status_code=status.HTTP_400_BAD_REQUEST,
                            detail='데이터를 찾을수 없습니다.')
    if current_user.id != db_question.user.id:
        raise HTTPException(status_code=status.HTTP_400_BAD_REQUEST,
                            detail='삭제 권한이 없습니다.')
    question_crud.delete_question(db=db, db_question=db_question)

# 질문 추천/ 추천 취소 API
@router.post('/vote', status_code=status.HTTP_204_NO_CONTENT)
def question_vote(_question_vote: question_schema.QuestionVote,
                  db: Session = Depends(get_db),
                  current_user: User = Depends(get_current_user)):
    db_question = question_crud.get_question(db, question_id = _question_vote.question_id)
    if not db_question:
        raise HTTPException(status_code=status.HTTP_400_BAD_REQUEST,
                            detail="데이터를 찾을수 없습니다.")
    question_crud.vote_question(db, db_question=db_question, db_user=current_user)

```

요청 처리 절차

1. 라우터에서 요청 수신:

- `@router.get("/list")` : 질문 목록을 가져오는 **GET 요청** 처리.
- `@router.post("/create")` : 새로운 질문을 생성하는 **POST 요청** 처리.
- `@router.put("/update")` : 질문을 수정하는 **PUT 요청** 처리.
- `@router.delete("/delete")` : 질문을 삭제하는 **DELETE 요청** 처리.

- `@router.post("/vote")` : 질문에 추천하거나 추천을 취소하는 **POST 요청** 처리.

2. CRUD 기능 호출:

- `question_crud` 모듈을 호출해 데이터베이스와 상호작용.
- 예외 처리(`HTTPException`)를 통해 클라이언트에 적절한 오류 메시지 반환.

3.2 CRUD 로직 예시 (`question_crud.py`)

질문 목록 조회:

- 데이터베이스의 모든 질문 데이터를 조회 후 반환.

```
# 질문 목록 조회
def get_question_list(db: Session, skip: int = 0, limit: int = 10,
                      keyword: str = ''):
    """
    - skip : 조회 시작 인덱스
    - limit: 최대 조회 개수
    - keyword: 검색어
    """
    question_list = db.query(Question)
    if keyword:
        search = '%%{}%%'.format(keyword)
        # 답변 및 사용자 정보를 서브쿼리로 포함
        sub_query = db.query(Answer.question_id, Answer.content, User.username
                              .outerjoin(User, and_(Answer.user_id == User.id)).subquery())
        question_list = question_list \
            .outerjoin(User) \
            .outerjoin(sub_query, and_(sub_query.c.question_id == Question.id)) \
            .filter(Question.subject.ilike(search) | # 질문제목
                    Question.content.ilike(search) | # 질문내용
                    User.username.ilike(search) | # 질문작성자
                    sub_query.c.content.ilike(search) | # 답변내용
                    sub_query.c.username.ilike(search) # 답변작성자
                    )
    total = question_list.distinct().count()
    question_list = question_list.order_by(Question.create_date.desc()) \
        .offset(skip).limit(limit).distinct().all()
```

```
return total, question_list # 전체 건수, 페이징 적용된 질문 목록
```

기능 설명

1. 페이징:

- `skip` 과 `limit` 파라미터를 사용해 특정 범위의 데이터만 조회.

2. 검색:

- `keyword` 를 사용해 질문 제목, 내용, 작성자, 답변 내용 등에서 검색.
- `ilike` 를 사용해 대소문자를 구분하지 않는 검색 수행.

3. 결과 반환:

- `total` : 전체 질문 개수.
- `question_list` : 검색 및 페이징이 적용된 질문 목록.

질문 생성:

```
# 질문 생성
def create_question(db: Session, question_create: QuestionCreate, user: User)
    db_question = Question(subject=question_create.subject,
                           content=question_create.content,
                           create_date=datetime.now(),
                           user=user)
    db.add(db_question)
    db.commit()
```

기능 설명

1. 질문 생성:

- 전달된 `question_create` 데이터와 로그인한 사용자 정보를 사용해 질문 객체 생성.

2. 데이터베이스 저장:

- `db.add(db_question)` : 데이터베이스 세션에 새로운 객체 추가.
- `db.commit()` : 변경 사항을 데이터베이스에 저장.

4. 데이터베이스와의 통신

FastAPI의 `question_crud` 모듈에서 데이터베이스 모델(`models.py`)과 세션(`database.py`)을 통해 MySQL과 통신

데이터베이스 모델 예시:

```
from sqlalchemy import Column, Integer, String, Text, DateTime, ForeignKey
from sqlalchemy.orm import relationship
from database import Base

# 질문 모델
class Question(Base):
    __tablename__ = 'question'

    id = Column(Integer, primary_key=True) # 질문 ID (기본키)
    subject = Column(String(255), nullable=False) # 질문 제목
    content = Column(Text, nullable=False) # 질문 내용
    create_date = Column(DateTime, nullable=False) # 생성일
    user_id = Column(Integer, ForeignKey("user.id"), nullable=True) # 작성자 ID (외래키)
    user = relationship("User", backref='question_users') # 작성자와의 관계 설정
    modify_date = Column(DateTime, nullable=True) # 수정일
    # 추천 사용자와의 관계 설정
    voter = relationship('User', secondary=question_voter, backref='question_voters')
```

데이터베이스 세션 생성

database.py:

```
import os
import time
from sqlalchemy import create_engine, text
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy.orm import sessionmaker
from sqlalchemy.exc import OperationalError
```

```

# 데이터베이스 URL - mysql 사용자
SQLALCHEMY_DATABASE_URL = os.getenv(
    "SQLALCHEMY_DATABASE_URL", "mysql+pymysql://myapp-sql:123456@r
)

# 데이터베이스 엔진 생성
def create_db_engine():
    # 데이터 베이스 연결 실패 시 5초마다 재시도
    while True:
        try:
            engine = create_engine(SQLALCHEMY_DATABASE_URL)
            # 연결 확인을 위한 테스트 쿼리 실행
            with engine.connect() as conn:
                conn.execute(text("SELECT 1")) # 수정된 부분
            print("데이터 베이스 연결 성공")
            return engine
        except OperationalError:
            print("데이터 베이스 연결 실패, 5초 뒤 재시도됩니다..")
            time.sleep(5)

# 데이터 베이스 엔진
engine = create_db_engine()

# 세션 생성
SessionLocal = sessionmaker(autocommit=False, autoflush=False, bind=engine)

Base = declarative_base()

# 데이터 베이스 세션 생성 및 종료
def get_db():
    db = SessionLocal()
    try:
        yield db

```

```
finally:
    db.close()
```

요청 처리 흐름:

1. FastAPI → CRUD → 데이터베이스 모델 → MySQL:

- 백엔드는 SQLAlchemy ORM을 통해 MySQL과 통신.
- 데이터 삽입, 조회, 갱신, 삭제 작업 수행.

2. MySQL → CRUD → FastAPI → 프론트엔드:

- MySQL에서 데이터를 조회한 결과를 FastAPI가 프론트엔드로 반환.

5. Nginx Reverse Proxy 역할

Nginx는 클라이언트 요청을 적절한 컨테이너로 전달

Nginx 설정:

`nginx/default.conf`

```
server {
    listen 8080;
    server_name cn-qna.com www.cn-qna.com;

    # 프론트엔드 라우팅
    location / {
        proxy_pass http://myfront:5173; # 프론트엔드 컨테이너
        proxy_http_version 1.1;
        proxy_set_header Upgrade $http_upgrade;
        proxy_set_header Connection "upgrade";
        proxy_set_header Host $host;
        proxy_set_header X-Real-IP $remote_addr;
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
    }

    # API 라우팅
    location /api {
        proxy_pass http://myfastapi:8000; # FastAPI 컨테이너
        proxy_http_version 1.1;
        proxy_set_header Upgrade $http_upgrade;
```



```

    proxy_set_header Connection "upgrade";
    proxy_set_header Host $host;
    proxy_set_header X-Real-IP $remote_addr;
    proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
}
}

```

- `/` 요청: Svelte 프론트엔드로 라우팅.
- `/api` 요청: FastAPI 백엔드로 라우팅.

최종 통신 절차 요약

1. 브라우저 요청 → Nginx Reverse Proxy → 프론트엔드(Svelte).
2. 프론트엔드 요청(API 호출) → Nginx Reverse Proxy → 백엔드(FastAPI).
3. 백엔드 → MySQL 데이터베이스 → 응답 → 프론트엔드 → 클라이언트.

4. 주요 기능

4.1 질문 관리

기능 설명

- 사용자는 질문을 등록, 조회, 수정, 삭제, 추천할 수 있음
- 아래는 각 기능의 동작 방식과 관련 API에 대한 설명

1. 질문 등록

- **설명:** 사용자가 질문의 제목과 내용을 입력해 새로운 질문을 등록
- **API:** `POST /api/question/create`

등록 절차:

1. 사용자가 제목과 내용을 입력 후 제출 버튼 클릭.
2. `api.js` 의 `fastapi` 함수를 통해 FastAPI 백엔드로 요청 전송.
3. 성공 시 질문 목록 화면으로 이동.

프론트엔드:

- `QuestionCreate.svelte` 에서 질문 등록 화면 구성.
 - 사용자 입력 값(`subject` , `content`)을 FastAPI로 전송.
-

2. 질문 목록 조회

- **설명:** 사용자가 등록된 모든 질문을 목록 형태로 조회.
- **API:** `GET /api/question/list?page=0&size=10&keyword=`

조회 절차:

1. 질문 목록 페이지(`Home.svelte`)에서 페이지 로드 시 API 호출.
2. FastAPI로부터 데이터를 받아 목록을 화면에 표시.

프론트엔드:

- `Home.svelte` 에서 질문 목록을 테이블 형태로 출력.
 - 질문 제목 클릭 시 해당 질문의 상세 페이지로 이동.
-

3. 질문 상세 조회

- **설명:** 사용자가 특정 질문의 상세 내용을 확인.
- **API:** `GET /api/question/detail/{question_id}`

조회 절차:

1. 사용자가 질문 제목을 클릭하면 `Detail.svelte` 로 이동.
2. `api.js` 를 통해 FastAPI 백엔드에서 질문 데이터를 받아와 화면에 출력.

프론트엔드:

- `Detail.svelte` 에서 질문 내용, 작성자, 작성일, 추천 수 표시.
 - 해당 질문에 답변 목록도 함께 표시.
-

4. 질문 수정

- **설명:** 사용자가 자신의 질문을 수정.
- **API:** `PUT /api/question/update`

수정 절차:

1. `Detail.svelte` 에서 수정 버튼 클릭 시 `QuestionModify.svelte` 로 이동.

- 수정된 내용을 입력 후 제출하면 FastAPI로 데이터 전송.
- 성공 시 질문 상세 화면으로 이동.

프론트엔드:

- `QuestionModify.svelte` 에서 제목과 내용을 수정할 수 있는 폼 제공.
-

5. 질문 삭제

- 설명:** 사용자가 자신의 질문을 삭제.
- API:** `DELETE /api/question/delete`

삭제 절차:

- `Detail.svelte` 에서 삭제 버튼 클릭.
- FastAPI로 삭제 요청 전송.
- 성공 시 질문 목록 화면으로 이동.

프론트엔드:

- 삭제 확인창(`window.confirm`) 표시 후 삭제 요청 처리.
-

6. 질문 추천

- 설명:** 사용자가 질문을 추천하거나 추천을 취소.
- API:** `POST /api/question/vote`

추천 절차:

- `Detail.svelte` 에서 추천 버튼 클릭.
- FastAPI로 추천 요청 전송.
- 성공 시 추천 수가 갱신.

프론트엔드:

- `Detail.svelte` 에서 추천 버튼과 추천 수 표시.
-

4.2 답변 관리

기능 설명

- 사용자는 특정 질문에 답변을 작성하거나 기존 답변을 삭제, 추천할 수 있음
-

1. 답변 등록

- **설명:** 사용자가 특정 질문에 대해 답변 작성.
- **API:** `POST /api/answer/create/{question_id}`

등록 절차:

1. `Detail.svelte` 에서 답변 입력 후 등록 버튼 클릭.
2. FastAPI로 답변 데이터를 전송.
3. 성공 시 답변 목록 갱신.

프론트엔드:

- `Detail.svelte` 에서 답변 작성 폼 제공.
 - 입력 필드는 로그인 상태에서만 활성화.
-

2. 답변 삭제

- **설명:** 사용자가 자신의 답변을 삭제.
- **API:** `DELETE /api/answer/delete`

삭제 절차:

1. 답변 삭제 버튼 클릭 시 삭제 확인창 표시.
2. FastAPI로 삭제 요청 전송.
3. 성공 시 답변 목록 갱신.

프론트엔드:

- `Detail.svelte` 에서 답변 삭제 버튼 제공.
-

3. 답변 추천

- **설명:** 사용자가 특정 답변을 추천하거나 추천을 취소.
- **API:** `POST /api/answer/vote`

추천 절차:

1. 답변 추천 버튼 클릭.
2. FastAPI로 추천 요청 전송.
3. 성공 시 추천 수 갱신.

프론트엔드:

- `Detail.svelte` 에서 답변 추천 버튼과 추천 수 표시.

4.3 사용자 관리 - 세션 관리

기능 설명

사용자 관리는 회원가입, 로그인, 로그아웃을 통해 이루어지며, FastAPI와 Svelte의 스토리지를 활용하여 세션 상태를 처리

1. Svelte의 스토리지를 이용한 세션 처리

Svelte Store를 활용한 상태 관리

Svelte에서는 `store` 를 통해 전역 상태 관리를 제공.

사용자 인증 상태와 관련된 정보를 저장하고 관리하기 위해 `lib/store.js` 파일에서 Svelte의 `writable` 스토어를 설정

store.js:

```
import { writable } from 'svelte/store'

// 로컬 스토리지와 연동되는 Svelte writable 스토어 생성
const persist_storage = (key, initialValue) => {
  // 로컬 스토리지에서 초기값 가져오기
  const storedValueStr = localStorage.getItem(key)
  const store = writable(storedValueStr != null ? JSON.parse(storedValueStr) : initialValue)

  // 값 변경 시 로컬 스토리지에 저장
  store.subscribe((val) => {
    localStorage.setItem(key, JSON.stringify(val))
  })
  return store
}

export const page = persist_storage("page", 0)
export const access_token = persist_storage("access_token", "")
```

```
export const username = persist_storage("username", "")
export const is_login = persist_storage("is_login", false)
export const keyword = persist_storage("keyword", "")
```

2. 회원가입 (UserCreate.svelte)

기능 설명:

- 사용자가 이메일, 사용자명, 비밀번호를 입력해 FastAPI로 데이터를 전송.
- 성공 시 로그인 페이지로 이동.

스토리지 사용:

- 회원가입 자체에서는 `store` 를 활용하지 않으나, FastAPI의 응답 상태에 따라 처리.

UserCreate.svelte:

```
<script>
import { push } from 'svelte-spa-router'
import fastapi from "../lib/api"
import Error from "../components/Error.svelte"

let error = {detail:[]}
let username = ''
let password1 = ''
let password2 = ''
let email = ''

// 회원 가입 함수
function post_user(event) {
  event.preventDefault()
  let url = "/api/user/create"
  let params = {
    username: username,
    password1: password1,
    password2: password2,
    email: email
  }
  // 사용자 생성 API 호출
  fastapi('post', url, params,
```

```

    (json) => {
      push('/user-login') // 성공 시 로그인 페이지 이동
    },
    (json_error) => {
      error = json_error
    }
  )
}
</script>

```

3. 로그인 (UserLogin.svelte)

기능 설명:

- 사용자가 이메일과 비밀번호를 입력하면 FastAPI에서 인증을 처리하고 JWT 토큰을 반환.
- 성공 시 `store` 에 토큰과 사용자 정보를 저장해 세션 상태를 유지.

스토리지 사용:

- 로그인 성공 시, `access_token` , `username` , `is_login` 을 업데이트.

코드 예제:

```

<script>
  import { push } from 'svelte-spa-router'
  import fastapi from "../lib/api"
  import Error from "../components/Error.svelte"
  import { access_token, username, is_login } from "../lib/store"

  let error = {detail:[]}
  let login_username = ""
  let login_password = ""

  // 로그인 함수
  function login(event) {
    event.preventDefault()
    let url = "/api/user/login"
    let params = {
      username: login_username,

```

```

    password: login_password,
  }
  // 사용자 로그인 API 호출
  fastapi('login', url, params,
    (json) => {
      $access_token = json.access_token
      $username = json.username
      $is_login = true
      push("/") // 성공 시 메인 페이지로 이동
    },
    (json_error) => {
      error = json_error
    }
  )
}
</script>

```

4. 로그아웃

기능 설명:

- 로그아웃 버튼 클릭 시, `store` 를 초기화하여 세션을 종료.
- FastAPI와의 서버 요청 없이 로컬에서 상태를 관리.

스토리지 사용:

- 로그아웃 시, `access_token` , `username` , `is_login` 초기화.

`Navigation.svelte` :

```

<!-- 네비게이션 메뉴 목록 -->
<div class="collapse navbar-collapse" id="navbarSupportedContent">
  <ul class="navbar-nav me-auto mb-2 mb-lg-0">
    <!-- 로그인 상태 -->
    {#if $is_login }
      <li class="nav-item">
        <a use:link href="/user-login" class="nav-link" on:click={() => {
          $access_token = ''
          $username = ''
          $is_login = false

```



```

    }}>로그아웃 ({ $username})</a>
  </li>
  { :else }
  <!-- 로그인 상태 -->
  <li class="nav-item">
    <a use:link class="nav-link" href="/user-create">회원가입</a>
  </li>
  <li class="nav-item">
    <a use:link class="nav-link" href="/user-login">로그인</a>
  </li>
  { /if }
</ul>
</div>

```

5. 인증 처리

스토어를 이용한 인증 상태 확인

`is_login` 값을 통해 사용자가 로그인 상태인지 확인하고, 비로그인 상태에서는 특정 기능을 제한.

API 요청 시 토큰 포함

`api.js` 파일에서 `Authorization` 헤더에 JWT 토큰을 자동으로 추가하여 인증을 처리

api.js :

```

import { get } from 'svelte/store';
import { access_token } from './store';

const fastapi = (operation, url, params, success_callback, failure_callback)
⇒ {
  let method = operation
  let content_type = 'application/json'
  let body = JSON.stringify(params)

  // 로그인 요청 시 헤더의 Content-Type 및 body 포맷 변경
  if(operation === 'login') {
    method = 'post'
    content_type = 'application/x-www-form-urlencoded'
  }
}

```

```

    body = qs.stringify(params)
  }

  // 서버 URL 생성
  let _url = fastapiBaseUrl + url;
  if(method === 'get') {
    // GET 요청의 경우 URL에 쿼리 파라미터 추가
    _url += "?" + new URLSearchParams(params)
  }
  // HTTP 요청 옵션 설정
  let options = {
    method: method,
    headers: {
      "Content-Type": content_type
    }
  }

  // Access Token이 있을 경우 Authorization 헤더에 추가
  const _access_token = get(access_token)
  if (_access_token) {
    options.headers["Authorization"] = "Bearer " + _access_token
  }

  // get 이외의 요청은 body를 포함
  if (method !== 'get') {
    options['body'] = body
  }

```

6. 종합 흐름

1. 로그인:

- 사용자 로그인 시 JWT 토큰과 사용자 정보를 `store`에 저장.
- `Authorization` 헤더를 통해 인증 요청 처리.

2. 로그아웃:

- `store` 초기화로 세션 종료.

3. 비로그인 상태 관리:

- `is_login` 이 `false` 인 경우, 주요 기능(질문/답변 작성 등) 비활성화.

4. UI 상태 반영:

- 로그인 여부에 따라 UI 버튼(로그인, 로그아웃 등) 동적으로 변경.

5. 구현 화면 캡처

1. 질문 목록 화면

번호	제목	글쓴이	작성일시
303	질문 수정 화면입니다. 1	aaa	2024년 11월 21일 10:41 오후
302	zzz	aaa	2024년 11월 17일 12:57 오전
301	테스트 데이터입니다.[197]		2024년 11월 16일 03:41 오후
300	테스트 데이터입니다.[205]		2024년 11월 16일 03:41 오후
299	테스트 데이터입니다.[138]		2024년 11월 16일 03:41 오후
298	테스트 데이터입니다.[199]		2024년 11월 16일 03:41 오후
297	테스트 데이터입니다.[200]		2024년 11월 16일 03:41 오후
296	테스트 데이터입니다.[203]		2024년 11월 16일 03:41 오후
295	테스트 데이터입니다.[188]		2024년 11월 16일 03:41 오후
294	테스트 데이터입니다.[204]		2024년 11월 16일 03:41 오후

1.1 페이지 동적 처리

Cloud Native

← → ↻ 주의 포함

cn-qna.com

🔍 ☆ 🏠 | 📄 🌐 ⋮

CN Q&A 회원가입 로그인

질문 등록하기

찾기

번호	제목	글쓴이	작성일시
212	테스트 데이터입니다.[276]		2024년 11월 16일 03:41 오후
211	테스트 데이터입니다.[277]		2024년 11월 16일 03:41 오후
210	테스트 데이터입니다.[278]		2024년 11월 16일 03:41 오후
209	테스트 데이터입니다.[279]		2024년 11월 16일 03:41 오후
208	테스트 데이터입니다.[280]		2024년 11월 16일 03:41 오후
207	테스트 데이터입니다.[281]		2024년 11월 16일 03:41 오후
206	테스트 데이터입니다.[284]		2024년 11월 16일 03:41 오후
205	테스트 데이터입니다.[293]		2024년 11월 16일 03:41 오후
204	테스트 데이터입니다.[295]		2024년 11월 16일 03:41 오후
203	테스트 데이터입니다.[291]		2024년 11월 16일 03:41 오후

이전

5

6

7

8

9

10

11

12

13

14

15

다음

2. 회원 가입 화면

Cloud Native

← → ↻ 주의 포함

cn-qna.com/#/user-create

🔍 ☆ 🏠 | 📄 🌐 ⋮

CN Q&A 회원가입 로그인

회원 가입

사용자 이름

비밀번호

비밀번호 확인

이메일

생성하기

3. 로그인 화면

Cloud Native

← → ↻ 주의 요함 cn-qna.com/#/user-login

CN Q&A 회원가입 로그인

로그인

사용자 이름

비밀번호

로그인

3.1 -로그인 후 질문 목록 화면

Cloud Native

← → ↻ 주의 요함 cn-qna.com/#/

CN Q&A 로그아웃 (aaa)

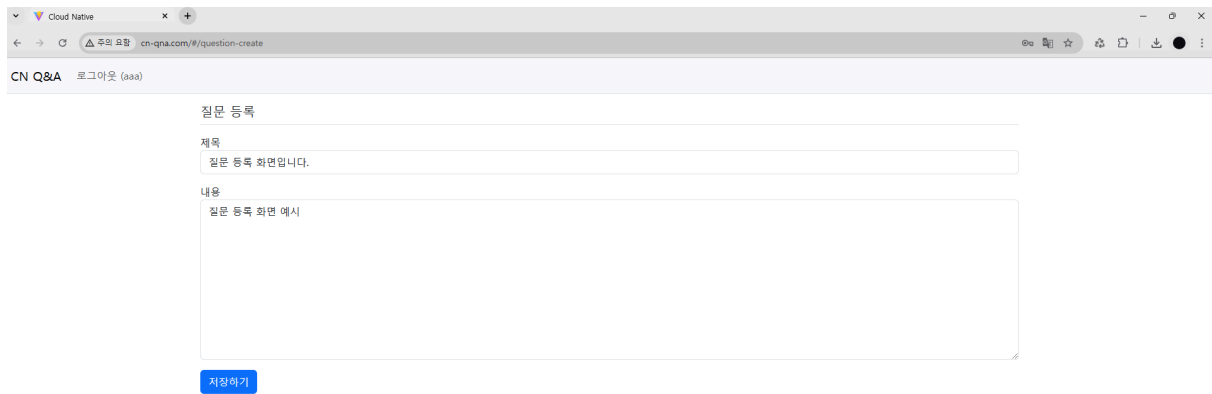
질문 등록하기

찾기

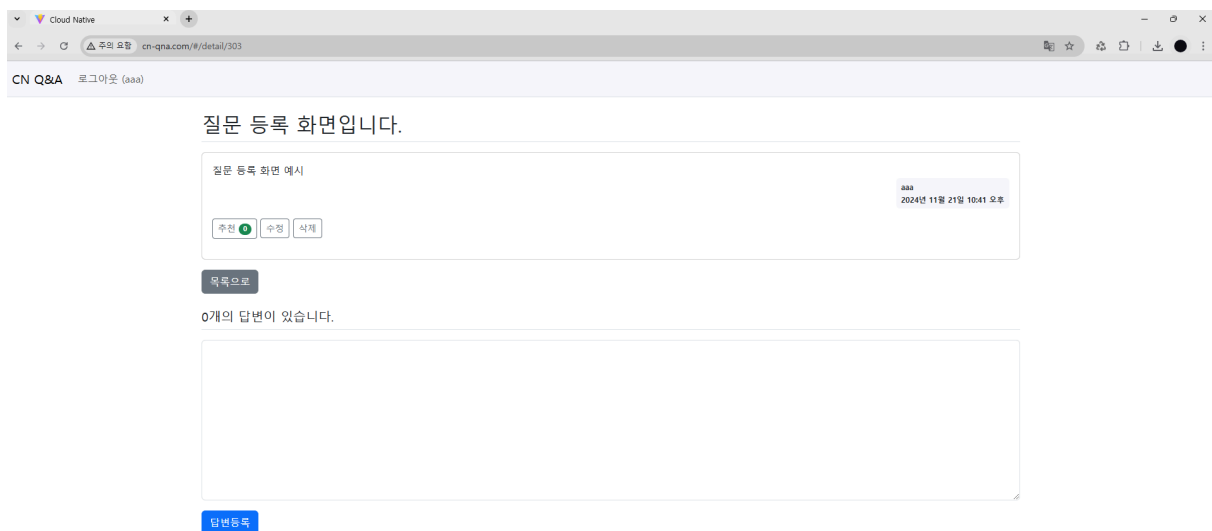
번호	제목	글쓴이	작성일시
212	테스트 데이터입니다.[276]		2024년 11월 16일 03:41 오후
211	테스트 데이터입니다.[277]		2024년 11월 16일 03:41 오후
210	테스트 데이터입니다.[278]		2024년 11월 16일 03:41 오후
209	테스트 데이터입니다.[279]		2024년 11월 16일 03:41 오후
208	테스트 데이터입니다.[280]		2024년 11월 16일 03:41 오후
207	테스트 데이터입니다.[282]		2024년 11월 16일 03:41 오후
206	테스트 데이터입니다.[294]		2024년 11월 16일 03:41 오후
205	테스트 데이터입니다.[293]		2024년 11월 16일 03:41 오후
204	테스트 데이터입니다.[295]		2024년 11월 16일 03:41 오후
203	테스트 데이터입니다.[291]		2024년 11월 16일 03:41 오후

이전 5 6 7 8 9 10 11 12 13 14 15 다음

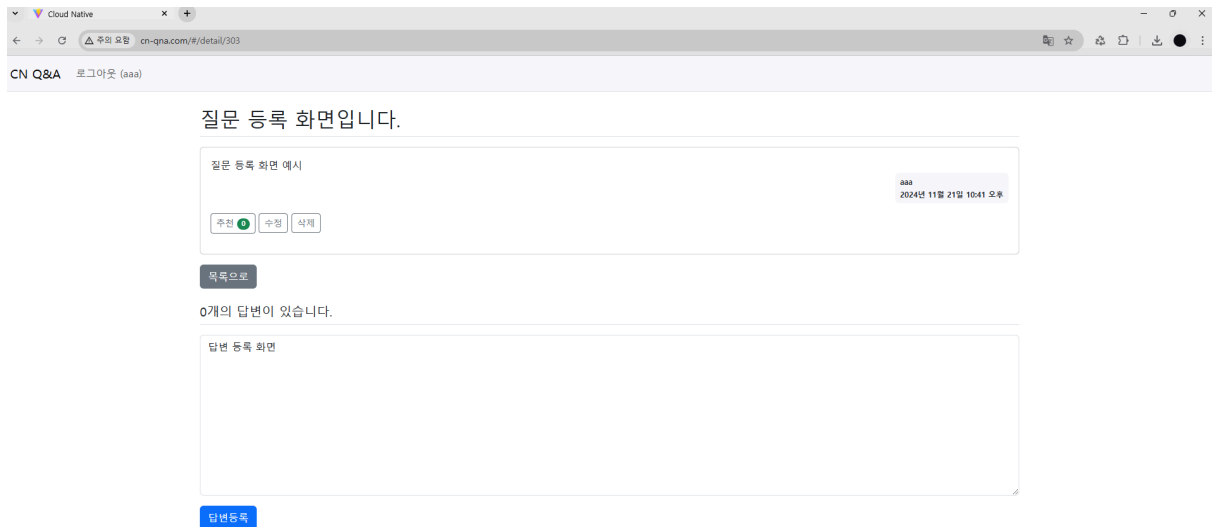
4. 질문 등록 화면



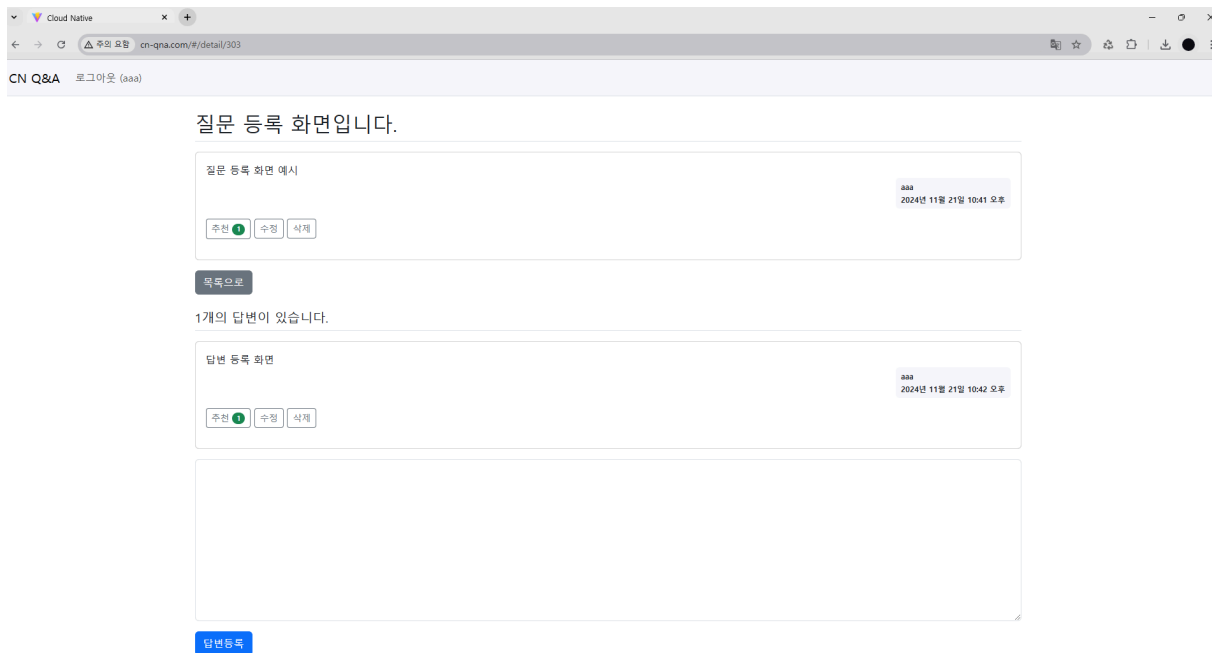
5. 질문 상세 화면



5.1 답변 등록 화면



5.2 질문/ 답변 추천



5.3 질문/답변 수정 화면

Cloud Native

← → ↻ 주의 요함 cn-qna.com/#/question-modify/303

CN Q&A 로그아웃 (aaa)

질문 수정

제목

질문 수정 화면입니다.

내용

질문 수정 화면 예시

수정하기

Cloud Native

← → ↻ 주의 요함 cn-qna.com/#/answer-modify/3

CN Q&A 로그아웃 (aaa)

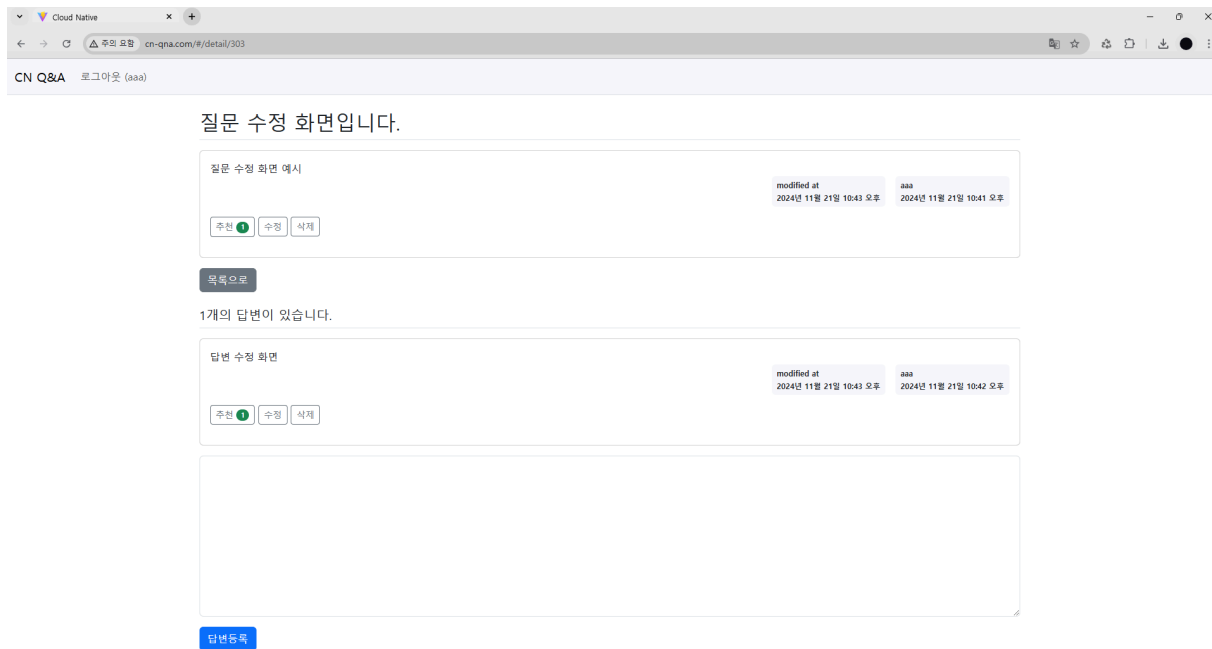
답변 수정

내용

답변 수정 화면

수정하기

5.4 질문/ 답변 수정 후 화면



6. 프로그램 설치 및 실행 방법

1. git clone을 통해 소스파일 다운로드

```
kty@master:~$ git clone https://github.com/tyoon11/fastapi-CN-QandA.git
```

```
git clone https://github.com/tyoon11/fastapi-CN-QandA.git
```

2. 프론트엔드 포트인 8002 포트 허용

```
kty@master:~/fastapi-CN-QandA$ sudo ufw allow 8002
규칙이 추가되었습니다
규칙이 추가되었습니다 (v6)
kty@master:~/fastapi-CN-QandA$
```

```
sudo ufw allow 8002
```

3. 받은 폴더로 이동하여 도커 컴포즈 파일 빌드

```
cd fastapi-CN-QandA
docker-compose up -d
```

```
kty@master:~$ cd fastapi-CN-QandA/
kty@master:~/fastapi-CN-QandA$ docker-compose up -d
```

```
Creating mysql ... done
Creating myfastapi ... done
Creating myfront ... done
Creating myproxy ... done
kty@master:~/fastapi-CN-QandA$
```

4. DNS 호스트 파일 변경

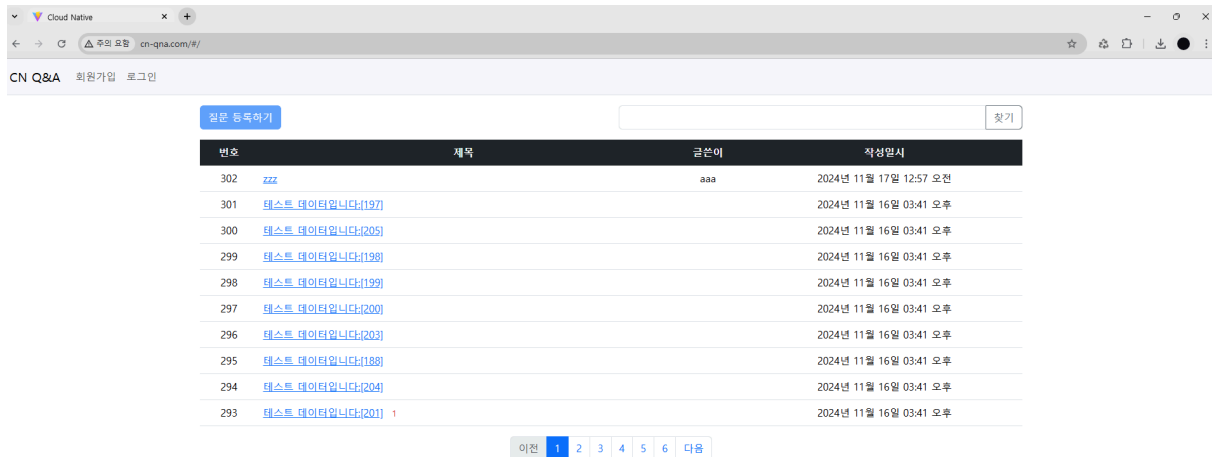
C:\Windows\System32\drivers\hosts → 메모장 관리자 권한으로 실행

```
# Copyright (c) 1993-2009 Microsoft Corp.
#
# This is a sample HOSTS file used by Microsoft TCP/IP for Windows.
#
# This file contains the mappings of IP addresses to host names. Each
# entry should be kept on an individual line. The IP address should
# be placed in the first column followed by the corresponding host name.
# The IP address and the host name should be separated by at least one
# space.
#
# Additionally, comments (such as these) may be inserted on individual
# lines or following the machine name denoted by a '#' symbol.
#
# For example:
#
# 102.54.94.97 rhino.acme.com # source server
# 38.25.63.10 x.acme.com # x client host
#
# localhost name resolution is handled within DNS itself.
# 127.0.0.1 localhost
# ::1 localhost
#
192.168.132.130 cn-qna.com
```

{우분투 guest IP} cn-qna.com

- hosts 파일 맨 끝에 추가

5. host os에서 http://cn-qna.com 접속



번호	제목	글쓴이	작성일시
302	zzz	aaa	2024년 11월 17일 12:57 오전
301	테스트 데이터입니다:[197]		2024년 11월 16일 03:41 오후
300	테스트 데이터입니다:[205]		2024년 11월 16일 03:41 오후
299	테스트 데이터입니다:[198]		2024년 11월 16일 03:41 오후
298	테스트 데이터입니다:[199]		2024년 11월 16일 03:41 오후
297	테스트 데이터입니다:[200]		2024년 11월 16일 03:41 오후
296	테스트 데이터입니다:[203]		2024년 11월 16일 03:41 오후
295	테스트 데이터입니다:[188]		2024년 11월 16일 03:41 오후
294	테스트 데이터입니다:[204]		2024년 11월 16일 03:41 오후
293	테스트 데이터입니다:[201] 1		2024년 11월 16일 03:41 오후

7. 개발 과정 문제 해결

1. Myfastapi 컨테이너가 바로 종료됨

- 로그 확인

```
$ docker-compose logs myfastapi
```

```
myfastapi | sqlalchemy.exc.OperationalError: (pymysql.err.OperationalError)
myfastapi | (Background on this error at: https://sqlalche.me/e/20/e3q8)
kty@master:~$
```

원인

- mysql 컨테이너에 연결이 안됨

- docker-compose 파일에 depends_on 옵션을 사용하더라도 mysql 컨테이너가 완전히 준비되기 전에 FastAPI가 실행 될 수 있음

해결

- fastapi에 연결 재시도 로직 추가

/fastapi/database.py

```
# database.py 파일
import os
import time
from sqlalchemy import create_engine
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy.orm import sessionmaker
from sqlalchemy.exc import OperationalError

SQLALCHEMY_DATABASE_URL = os.getenv("SQLALCHEMY_DATABASE_URL")

# 데이터베이스 엔진 생성 함수에 재시도 로직 추가
def create_db_engine():
    while True:
        try:
            engine = create_engine(SQLALCHEMY_DATABASE_URL)
            # 연결 확인을 위한 테스트 쿼리 실행
            with engine.connect() as conn:
                conn.execute(text("SELECT 1"))
            print("Successfully connected to the database.")
            return engine
        except OperationalError:
            print("Database connection failed. Retrying in 5 seconds...")
            time.sleep(5)

engine = create_db_engine()
SessionLocal = sessionmaker(autocommit=False, autoflush=False, bind=engine)
Base = declarative_base()

def get_db():
    db = SessionLocal()
```

```
try:
    yield db
finally:
    db.close()
```

→ 해결 완료

2. myfront 컨테이너가 바로 종료됨

- 로그확인

```
$ docker-compose logs myfront
```

```
myfront    | > frontend@0.0.0 dev
myfront    | > vite --host
myfront    |
myfront    | sh: 1: vite: not found
```

원인

- /frontend/Dockerfile 에서 RUN npm install 을 실행하였지만
- docker-compose.yaml에서 로컬소스 바인딩을 하여 생성된 모듈파일이 들어있는 컨테이너 내 파일을 덮어씀

해결

- node_modules를 별도의 볼륨을 만들어 볼륨 바인딩
- yaml파일 수정

```
myfront:
  build:
    context: ./frontend
    dockerfile: Dockerfile
  container_name: myfront
  ports:
    - "8002:5173" # 프론트엔드 접근 포트
  volumes:
```

```
- ./frontend:/src # 로컬 소스 코드 바인딩
- node_modules:/src/node_modules # 컨테이너 내부 node_modules를 별도로 볼 수 있도록 설정
networks:
  - mynet
depends_on:
  - myfastapi
```

→ 해결 완료
