PORTFOLIO

빵집 정보 웹어플리케이션 “빵자국” 포트폴리오

이름	이 찬 우
이메일	chw210601@gmail.com
깃허브	https://github.com/LChanwoo

Project

프로젝트 소개

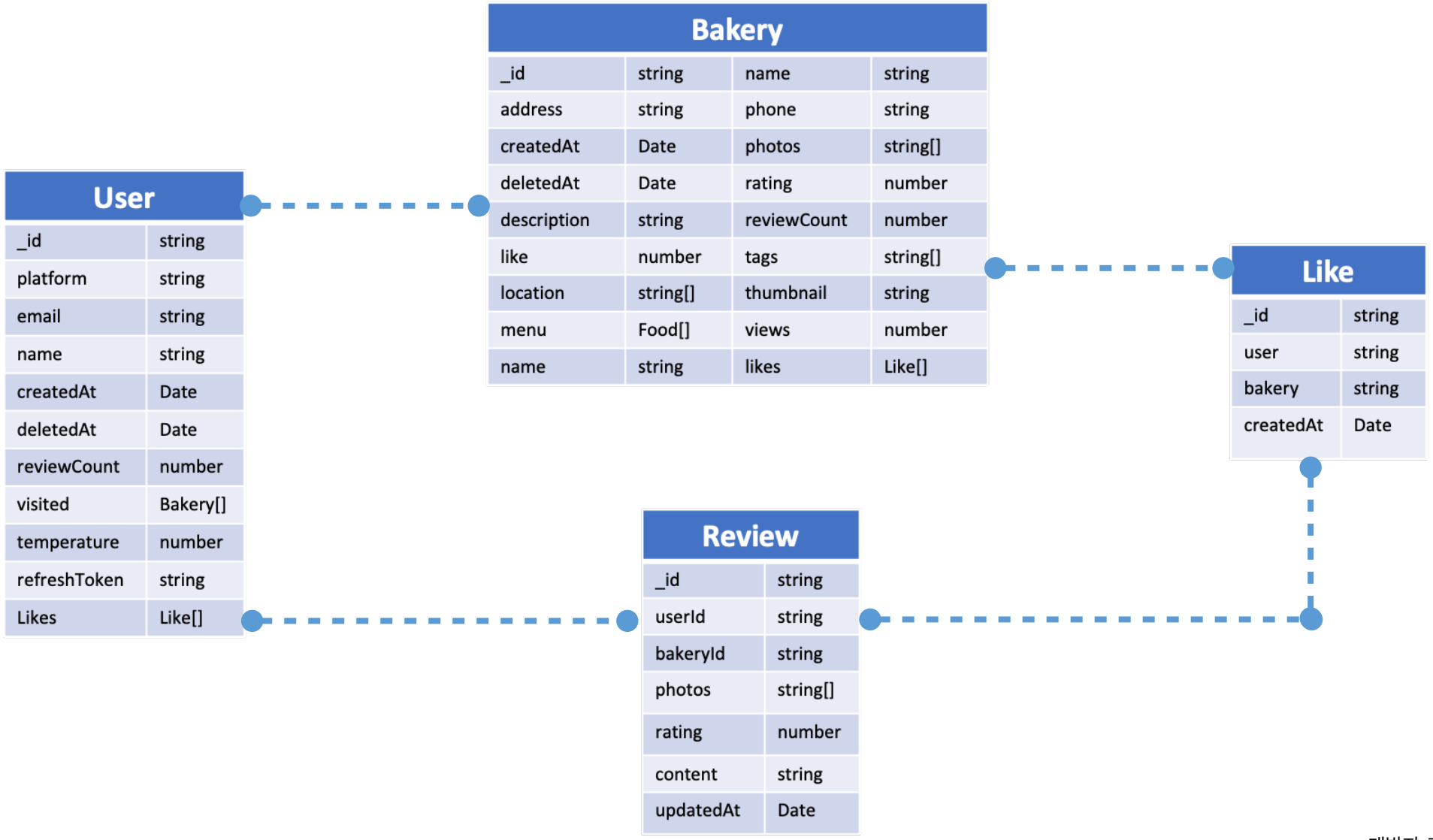
About project

빵자국은 숨은 빵 맛집등을 찾아서 추천해주는 웹 애플리케이션입니다.

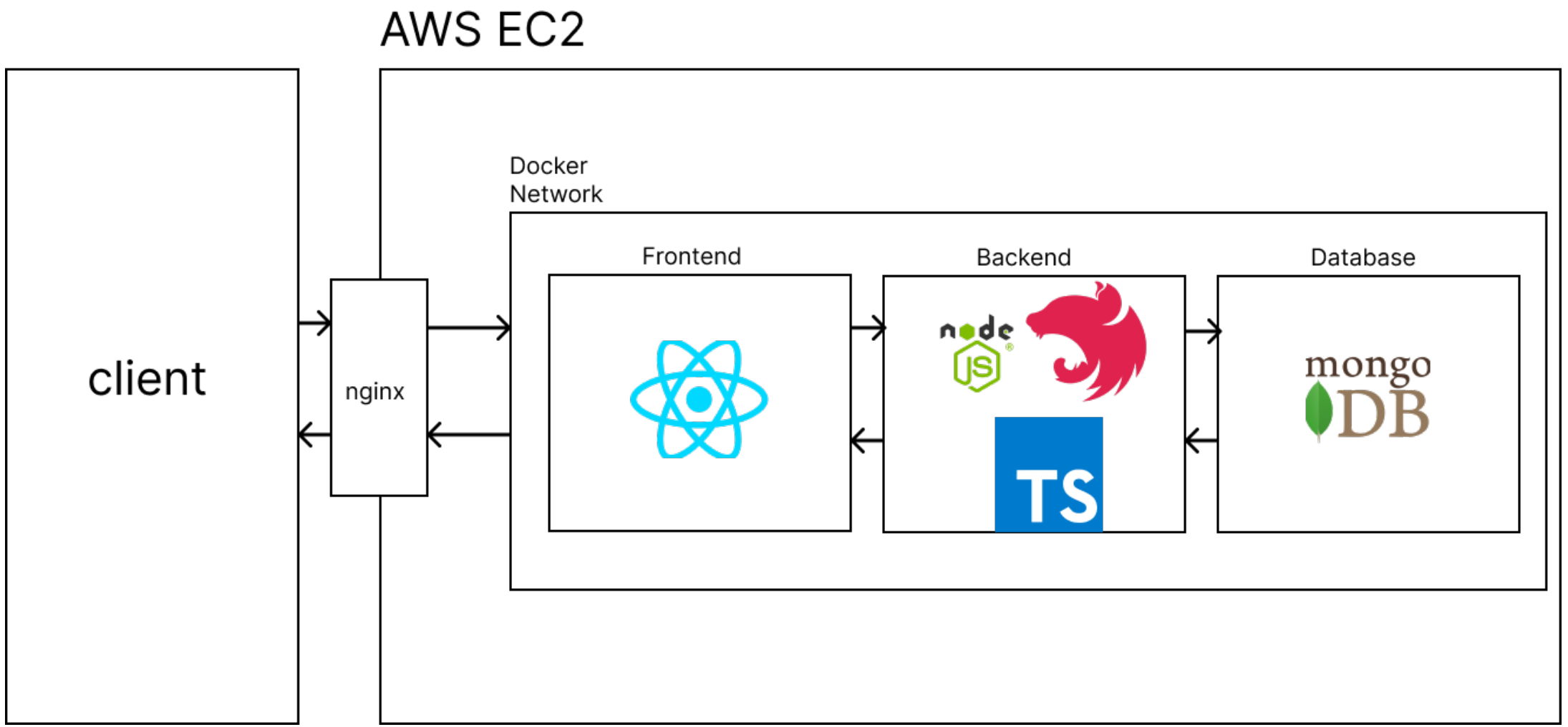
Introduce project

작업 기간	2023. 01 ~ (진행 중)
인력 구성(기여도)	디자이너 1명 / BE 1명 / FE 1명 (BE 기여도 100%)
프로젝트 목적	팀 프로젝트로 디자이너, 프론트엔드 개발자와의 협업능력을 배양하고, Nestjs 프레임워크에 숙달되는 것
프로젝트 내용	숨은 빵 맛집을 추천해주는 빵집 추천 웹애플리케이션
주요 업무 및 상세 역할	1) EC2 Instance 관리 2) 서비스 기획 3) 백엔드 작업 4) 아키텍쳐 설계
사용언어 및 개발 환경	React, AWS, EC2, Nodejs, Nestjs, mongoDB,
Github	https://github.com/teamH2/backend

Main work Database Structure(mongoDB)



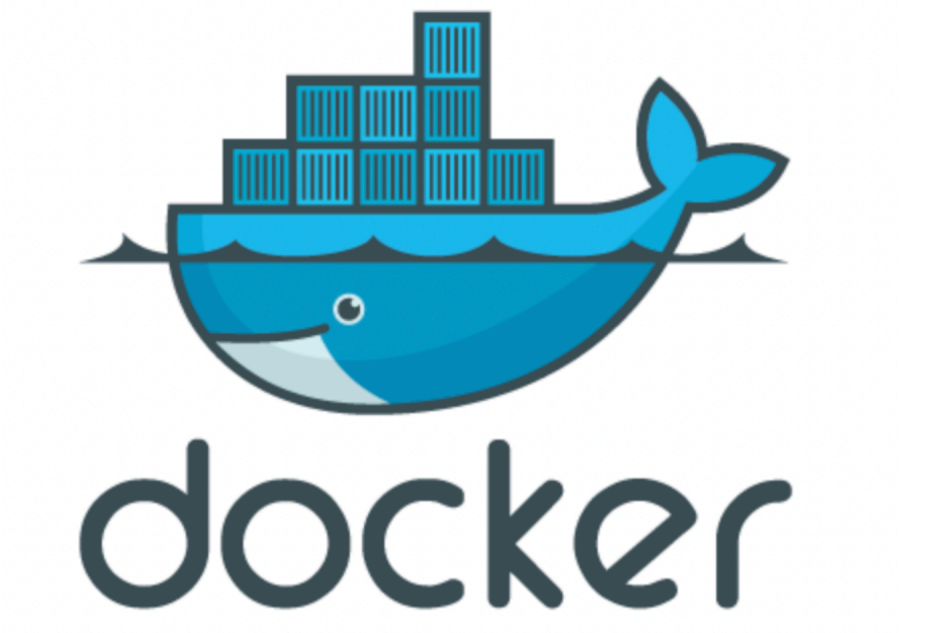
Main work Architecture



Main work Docker

Docker는 컨테이너 가상화 기술을 이용하여 애플리케이션을 배포하는데 사용됩니다. docker 애플리케이션을 컨테이너화하므로 애플리케이션이 실행되는 환경이 동일할 때 예측 가능한 동작을 하도록 할 수 있습니다. 또한 컨테이너를 필요한 만큼 추가하여 에플리케이션을 확장할 수도 있습니다. 추가로, docker는 컨테이너를 격리된 환경에서 애플리케이션을 실행하므로, 호스트 서버와 분리되어 보안이 강화된다는 장점이 있습니다.

즉, 환경일치성, 확장성, 보안, 빠른 배포등을 이유로 docker를 사용하게 되었습니다.



Main work Docker

Docker image build - nestjs

docker를 이용하여 nestjs를 빌드하여 배포하는 dockerfile의 캡쳐화면입니다. dockerfile 코드의 작성으로 인해 빠르게 빌드하고 배포할 수 있었습니다.

dockerfile > ...

```
1   FROM node:18 AS development
2
3
4   RUN mkdir -p /usr/src/app/
5
6   WORKDIR /usr/src/app
7
8   COPY package*.json ./
9
10  COPY . /usr/src/app
11
12  RUN npm install
13
14  RUN npm run build
15
16  FROM node:18-alpine AS production
17
18  ARG NODE_ENV=production
19  ENV NODE_ENV=${NODE_ENV}
20
21  WORKDIR /usr/src/app
22
23  COPY package*.json ./
24
25  COPY . /usr/src/app
26
27  RUN npm install
28
29  VOLUME /var/log/nestjs
30
31
32  EXPOSE 8080:8080
33
34
35  COPY --from=development /usr/src/app/dist ./dist
36
37  CMD ["node", "dist/main"]
38
```

Main work Docker

Docker image build - mongoDB

mondoDB를 컨테이너화 할 때 사용된. docker-compose 코드입니다. 이 코드로 인해 docker 컨테이너를 생성할 때 필요한 설정을 미리 해둘 수 있었습니다.

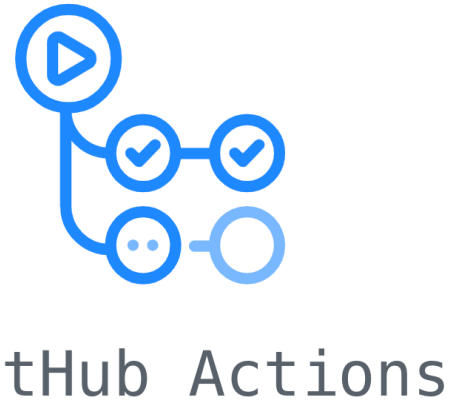
또한 volumes를 설정하여 db에 저장되는 데이터를 호스트 서버에서도 관리할 수 있게 만들었습니다.

```
version: '3.8'

services:
  mongo:
    image: mongo
    container_name: ${MONGO_CONTAINER}
    volumes:
      - ./mongodb_data_container:/data/db
    restart: always
    env_file:
      - .env
    environment:
      - MONGO_INITDB_DATABASE=${MONGO_DATABASE}
      - MONGO_INITDB_ROOT_USERNAME=${MONGO_USERNAME}
      - MONGO_INITDB_ROOT_PASSWORD=${MONGO_PASSWORD}
      - TZ=Asia/Seoul
    ports:
      - ${MONGO_PORT}:${MONGO_PORT}
```

Main work

Github Action



GitHub Actions는 GitHub에서 제공하는 지속적인 통합 및 배포 서비스입니다. GitHub Actions는 간단한 YAML 파일로 설정할 수 있습니다. 이를 이용하여 쉽게 CI/CD 파이프라인을 설정할 수 있습니다. 또한 macOS, Windows, Linux 다양한 플랫폼을 제공하여, docker 컨테이너가 실행될 환경과 같은 환경에서 build 할 수 있습니다.

Main work Github Action

```yaml
- name: Create .env file
  run: |
    jq -r 'to_entries|map("\(.key)=\(.value|tostring)")|.[]' <<< "$SECRETS_CONTEXT" > ./.env
    cat ./.env
  env:
    SECRETS_CONTEXT: ${{ toJson(secrets) }}
- name: Build and push
  id: docker_build
  uses: docker/build-push-action@v4
  with:
    context: .
    file: ./dockerfile
    builder: ${{ steps.buildx.outputs.name }}
    cache-from: type=gha
    cache-to: type=gha,mode=max
    push: ${{ github.event_name != 'pull_request' }}
    tags: ${{env.DOCKER_IMAGE}}:latest
```

CI/CD - 동적 환경변수 파일 생성 - 캐싱

좌측의 코드는 github action의 설정파일인 main.yml 파일입니다.
github repository에서 바로 build하여 cicd를 구축할 경우, 환경변수 .env파일을 넣을 수 없다는 문제점이 있습니다. 이 문제를 해결하기 위해 github secrest에 환경변수를 미리 입력해두고, 빌드될 때 자동으로 .env파일을 동적 생성하여 환경변수를 사용할 수 있게 하였습니다.

ci/cd를 할 때 반복적으로 같은 node_modules를 다운받고, 처음부터 docker빌드를 하면 시간이 많이 걸립니다. 이를 방지하고자 캐시기능을 넣어 ci/cd할 때 보다 빠른속도로 빌드하도록 만들었습니다.

Main work

Github Action

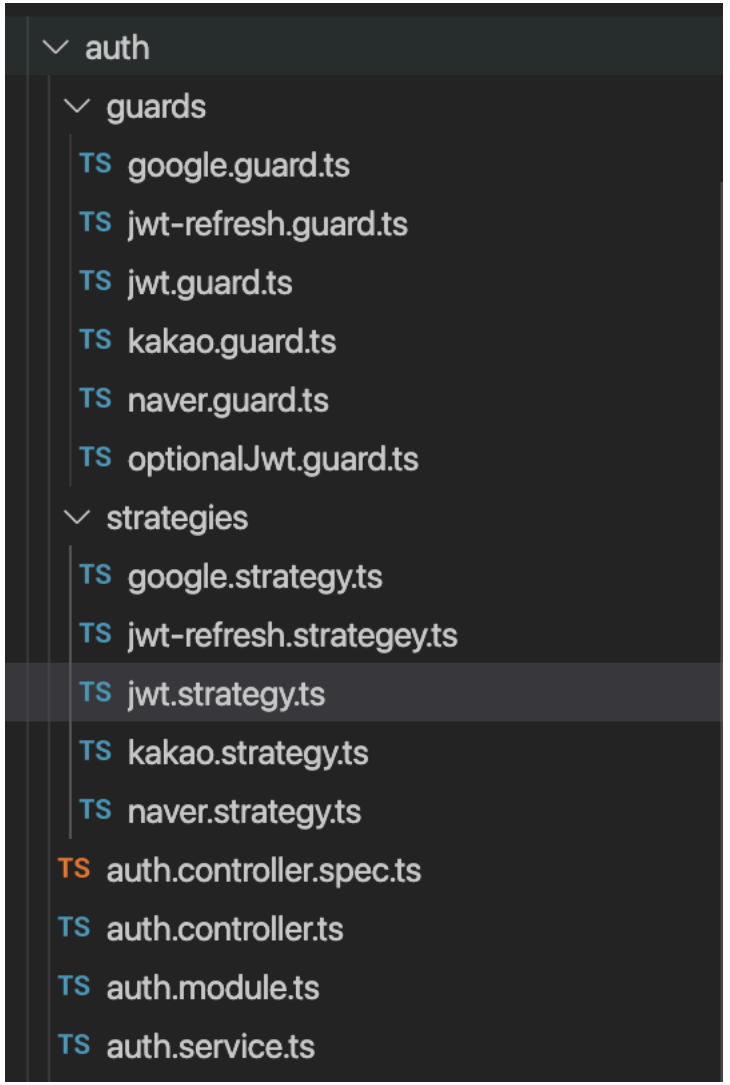
```
- name: Docker run
  run: |
    docker run  -d -p 8080:8080 -v /home/ubuntu/server/logs:/usr/src/app/logs
    --name ${{env.CONTAINER_NAME}} --restart always ${{env.DOCKER_IMAGE}}:latest
- name: Docker network connect
  run: |
    docker ps -a
    docker network connect server-network ${{env.CONTAINER_NAME}}
- name: Clean up docker images
  run:
    docker image prune
```

Docker container 실행, network 연결

Github action 서버에서 빌드한 빌드 이미지를 AWS EC2 인스턴스 서버로 가지고 와서 컨테이너를 생성하였습니다. 또한 docker로 생성된 mongodb에 연결하기 위해 컨테이너를 같은 network에 추가하여 서로 연결이 가능하도록 하였습니다.

Main work 로그인

소셜 로그인과 JWT



Main work 로그인

AuthStrategy, 검증

Passportjs를 이용하여 소셜로그인을 구현하였습니다. Stategy를 작성하여, 클라이언트로부터 각 소셜로그인의 Access Token을 전달받아 검증하도록 하였습니다. 이 때, 데이터베이스에 저장되어 있지 않은 유저가 로그인했을 경우 새롭게 데이터베이스에 추가하도록 하였습니다. 이 과정에서 새로운 유저가 가입되었음을 기록하는 로그를 작성하도록 코드를 작성하였습니다.

```
@Injectable()
export class GoogleStrategy extends PassportStrategy(Strategy, 'google') {
  constructor(
    private readonly authService: AuthService,
  ) {
    super({
        usernameField: 'credential'
    });
  }

  async validate(credential:string ,_, done: any,){
    try{
        const user = await this.authService.getGooglePayload(credential);
        return user;
    }catch(err){
        logger.error(err);
        return false;
    }
  }
}
```

```
async getGooglePayload(credential:string) {
  try{
    const ticket = await googleclient.verifyIdToken({
      idToken: credential,
      audience: process.env.OAUTH_ID,
    });
    const { email, name, picture } = ticket.getPayload();
    const userinfo = {
      email,
      platform: Platform.GOOGLE,
    }
    const user = await this.userModel.findOne(userinfo).exec();
    if(!user){
      const newUser = await this.userModel.create({
        email,
        name,
        thumbnail: picture,
        platform: Platform.GOOGLE,
      })
      logger.log(`${newUser.platform} ${newUser.email} is created`)
      return newUser;
    }
    return user;
  }catch(err){
    console.log(err)
  }
}
```

Main work 로그인

```
@Injectable()
export class JwtStrategy extends PassportStrategy(Strategy,'jwt'){
    constructor(
        private readonly authService: AuthService,
        private readonly uersService: UsersService,
    ){
        super({
            secretOrKey: process.env.JWT_ACCESS_TOKEN_SECRET,
            jwtFromRequest: ExtractJwt.fromExtractors(
                [JwtStrategy.cookieExtractor,
                ExtractJwt.fromAuthHeaderAsBearerToken()
            ]),
            ignoreExpiration: false,
        });
    }
    private static cookieExtractor(req:Request) :string|null {
        const jwt_name = process.env.JWT_NAME;
        if(req.cookies && jwt_name in req.cookies && req.cookies[jwt_name] !== ''){
            return req.cookies[jwt_name];
        }
        return null
    }

    async validate(payload : any) {
        const { _id } = payload;
        try{
            const user = await this.uersService.findUserById(_id);
            if(!user){
                throw new UnauthorizedException("user not found");
            }
            return payload;
        }catch(err){
            throw new UnauthorizedException("user not found");
        }
    }
}
```

JWT 검증

JWT를 검증할 때는 api 요청의 쿠키정보를 이용하여 검증합니다. 이 때, JWT에 저장된 유저정보가 DB에 존재하지 않을 경우 401에러를 반환하도록 하였습니다.

Main work 로그인

```
@Get('/refresh')
@UseGuards(JwtRefreshGuard)
async refresh(@User() user :any,@Req() req : Request, @Res() res: Response): Promise<any> {
    return await this.authService.refreshToken(user,req,res);
}
```

```
async getUserRefreshTokenMatches(refreshToken: string,userId: string): Promise<boolean> {
    const user = await this.userService.findUserById(userId);
    if (user == null) {
      throw new UnauthorizedException('user not found');
    }
    const isRefreshTokenMatch = await bcrypt.compare(
      refreshToken,
      user.refreshToken,
    );
    if (isRefreshTokenMatch) {
      return true;
    } else {
      return false
    }
}
```

```
async refreshToken( user: any, req ,res: Response ) : Promise< any >{
  try{
    if(user){
        const { accessToken, accessOption } = await this.getCookieWithJwtAccessToken(user);
        res.cookie(this.configService.get('JWT_NAME'), accessToken,accessOption);
        return res.status(201).end();
    }else{
        throw new UnauthorizedException('you are not logged in');
    }
  }catch(e){
      throw new HttpException(e.message, e.status);
  }
}
```

Token Refresh

JWT가 만료되었을 때, JWT를 재발급하기 위해 사용되는 refreshToken 기능을 넣었습니다.
refreshToekn을 검증할 때는 암호화하여 DB에 저장해둔 토큰과 클라이언트로부터 받은 토큰을 비교하여, 일치하면 JWT를 재발급하도록 하였습니다.

Main work

Logging

Winston 모듈을 사용하여 서버로그를 저장하도록 하였습니다. 이 때 info, warn, error로 각각 나누어서 저장하도록 하였습니다. 이 프로젝트에서는 docker를 사용하였으므로 미리 volume 설정을 하여 docker container와 host server의 폴더를 공유하도록 하여 log를 contianer 외부에서도 관리할 수 있게 하였습니다. 또한 api요청이 있을 때 마다 middleware를 이용하여 로그를 저장하도록 하였습니다.

```
@Injectable()
export class LoggerMiddleware implements NestMiddleware {
  private logger = new Logger('HTTP');

  use(request: Request, response: Response, next: NextFunction): void {
    const { ip, method, originalUrl } = request;

    response.on('finish', () => {
      const { statusCode } = response;
      const contentLength = response.get('content-length');
      this.logger.log(
        `${method} ${originalUrl} ${statusCode} ${contentLength} - ${ip}`,
      );
    });

    next();
  }
}
```

```
const dailyOptions = (level: string) => {
  return {
    level,
    datePattern: 'YYYY-MM-DD',
    dirname: logDir + `/${level}`,
    filename: `%DATE%.${level}.log`,
    maxFiles: 30, //30일치 로그파일 저장
    zippedArchive: true, // 로그가 쌓이면 압축하여 관리
  };
};

const env = process.env.NODE_ENV;
const logDir = __dirname + '/../../logs'; // log 파일을 관리할 폴더
export const winstonLogger = WinstonModule.createLogger({
  transports: [
    new winston.transports.Console({
      level: 'production',
      format:
        winston.format.combine(
          winston.format.timestamp(),
          utilities.format.nestLike('backend', {
            prettyPrint: true,
          }),
        ),
    }),

    // info, warn, error 로그는 파일로 관리
    new winstonDaily(dailyOptions('info')),
    new winstonDaily(dailyOptions('warn')),
    new winstonDaily(dailyOptions('error')),
  ],
});
```

Main work 좋아요

```
async create(userId: string, bakeryId: string): Promise<Like> {

    const existingLike = await this.likeModel.findOne({ user:userId, bakery:bakeryId });
    if (existingLike) {
        throw new HttpException('User has already liked this bakery', 400);
    }

    const like = new this.likeModel({
        user:userId,
        bakery:bakeryId,
    });
    await like.save()
    await this.bakeryModel.findByIdAndUpdate(bakeryId, { $inc: { like: 1 }, $push: { likes: like._id } });
    await this.userModel.findByIdAndUpdate(userId, { $push: { likes: like._id } });
    return await like.save();
}
```

마음에 드는 빵집을 북마크하는 기능인 “좋아요” 기능을 구현하였습니다. 빵집,유저정보와 Join하여 다른 컬렉션에서도 쉽게 좋아요 정보를 사용할 수 있도록 하였습니다.

좋아요를 취소할 때는 여러 검증과정을 거친 후. 빵집, 유저 컬렉션으로터 좋아요 정보를 삭제하도록 하였습니다.

```
async delete(userId: string, bakeryId: string): Promise<Like> {
    try{
        const existingLike = await this.likeModel.findOne({ user: userId, bakery: bakeryId });
        if (!existingLike) {
            throw new HttpException('User has not liked this bakery', 400);
        }
        if(existingLike.user.toString() !== userId){
            throw new HttpException('User has not liked this bakery', 400);
        }
        if(existingLike.bakery.toString() !== bakeryId){
            throw new HttpException('User has not liked this bakery', 400);
        }
        const bakeryUpdate = await this.bakeryModel.findByIdAndUpdate(
            bakeryId, { $inc: { like: -1 } },
            { $pull: {likes:existingLike._id} })
            .exec();
        if(bakeryUpdate.like<0){
            await this.bakeryModel.findByIdAndUpdate(bakeryId, { like: 0 }).exec();
            throw new HttpException('서버에러',500);
        }
        await this.userModel.findByIdAndUpdate(userId, { $pull: { likes: existingLike._id } }).exec();
        return await this.likeModel.findOneAndDelete({ user: userId, bakery: bakeryId }).exec();
    }catch(error){
        logger.error(error);
        if(error instanceof HttpException){
            throw new HttpException('User has not liked this bakery', 400);
        }
        throw new HttpException('서버에러',500);
    }
}
```

Main work

Interceptor

```
@Injectable()
export class SuccessInterceptor implements NestInterceptor {
  intercept(context: ExecutionContext, next: CallHandler): Observable<any> {
    const ctx = context.switchToHttp();
    const response = ctx.getResponse();
    const request = ctx.getRequest();
    const status = response.statusCode;
    return next.handle().pipe(
      map((data) => {
        return {
          success: true,
          statusCode: status,
          data
        };
      }),
    );
  }
}
```

API 요청 성공 시 인터셉터

클라이언트의 요청에 성공적으로 응답했을 때, 클라이언트측에서 알기 쉽도록 성공여부, status code, data 3부분으로 나누어 응답하도록 하였습니다

Main work

Interceptor

```
@Catch(HttpException)
export class HttpExceptionFilter implements ExceptionFilter {
  catch(exception: HttpException, host: ArgumentsHost,) {
    const ctx = host.switchToHttp();
    const response = ctx.getResponse<Response>();
    const request = ctx.getRequest<Request>();
    const status = exception.getStatus();
    const error = exception.getResponse();
    logger.error(`[${request.ip}] ${request.method} ${request.url} ${status} ${JSON.stringify(error)}`);
    response.status(status).json({
      statusCode: status,
      success: false,
      timestamp: new Date().toISOString(),
      path: request.url,
      error,
    });
  }
}
```

API 요청 실패 시 인터셉터

클라이언트의 요청에 응답하기를 실패했을 때, 클라이언트측에서 알기 쉽도록 성공여부, status code, 시간, url, error 6 부분으로 나누어 응답하도록 하였습니다.
이 때 에러를 로깅하여 분석할 수 있도록 하였습니다

Main work

Algorithm

하버사인 알고리즘

현재 클라이언트의 위치와 빵집사이의 거리를 계산하기 위하여 하버사인 알고리즘을 사용하였습니다. 하버사인 알고리즘은 지구상의 두 지점 간의 거리를 계산하는 방법 중 하나입니다. 이 알고리즘은 두 지점의 위도와 경도 정보를 사용하여 지구 상에서 두 지점 사이의 거리를 계산합니다.

```
//haversine algorithm
private getDistanceFromLatLonInKm(
    lat1: number,
    lon1: number,
    lat2: number,
    lon2: number,
): number {
    const R = 6371; // 지구 반경 (km)
    const dLat = this.deg2rad(lat2 - lat1);
    const dLon = this.deg2rad(lon2 - lon1);
    const a =
        Math.sin(dLat / 2) * Math.sin(dLat / 2) +
        Math.cos(this.deg2rad(lat1)) *
        Math.cos(this.deg2rad(lat2)) *
        Math.sin(dLon / 2) *
        Math.sin(dLon / 2);
    const c = 2 * Math.atan2(Math.sqrt(a), Math.sqrt(1 - a));
    const d = R * c; // 두 지점 사이의 거리 (km)
    return d;
}

private deg2rad(deg: number): number {
    return deg * (Math.PI / 180);
}
```

Main work

API docs

Swagger

swagger를 사용하여 api 문서를 작성하였습니다. parameter부터 응답 code까지, 제가 프론트엔드 작업자라는 생각으로 하나하나 자세하게 작성하였습니다.

Thanks