제출용 포팅 메뉴얼

1장. 신규 서비스 기본 정보 확정하기

1.1 작성 방법

- 이 장은 **생각보다 중요하지만, 할 일은 단순**하다.
- 아래 템플릿들을 **그대로 복붙해서** 서비스별로 채우면 된다.
- 서비스 하나당 1장씩 만들어도 되고, 여러 서비스면 동일 양식을 복제해서 쓰면 된다.

1.2 서비스 메타 정보 시트

> 이 서비스가 “누구냐”를 정의하는 표

아래 표를 문서에 그대로 넣고 채우면 된다.

```
[1-1. 서비스 메타 정보]

- 서비스 코드명(내부에서 부르는 이름):
  예) user, world, paint, snapshot ...

- Kubernetes 서비스 이름(최종):
  예) user-svc, world-svc, paint-svc ...

- 한 줄 설명:
  예) "유저 프로필/권한 관리", "월드/블록 상태 조회·저장"

- 역할 유형 (택 1 또는 복수):
  [ ] HTTP API 서버
  [ ] Kafka Consumer/Producer
  [ ] 배치/스케줄러
  [ ] 기타: _____

- 기본 HTTP 포트:
  (가능하면 8080 사용)
```

- 사용 언어 / 프레임워크 / 버전:
  예) Java 21 + Spring Boot 3.5.x

- 빌드 도구:
  예) Gradle Kotlin DSL / Maven

- 소스코드 경로:
  예) repo/BE/services/<service-name>

1.3 외부 의존성 체크 시트

> 이 서비스가 “어디에 기대고 사는지” 정리

1.3.1 데이터베이스

[1-3-1. DB 의존성]

- DB 사용 여부:
  [ ] 사용함
  [ ] 사용 안함

- DB 종류:
  [ ] PostgreSQL (RDS)
  [ ] H2 (개발용)
  [ ] 기타: _____

- 운영 시 실제로 붙을 DB:
  예) RDS PostgreSQL (colombus-test-db)

- 스키마/DB 이름 (예상):
  예) colombus_user, colombus_world ...

- 마이그레이션 도구:
  [ ] Flyway
  [ ] Liquibase
  [ ] 직접 SQL 관리
  [ ] 없음

- 비고:
  (특이사항 있으면 적기: 대용량 테이블, 파티션, 인증 방식 등)

1.3.2 Redis

[1-3-2. Redis 의존성]

- Redis 사용 여부:
  [ ] 사용함
  [ ] 사용 안함

- 사용 용도 (복수 선택 가능):
  [ ] 캐시
  [ ] 세션 저장
  [ ] Pub/Sub
  [ ] 기타: _____

- 운영 시 붙을 Redis:
  예) ElastiCache Redis cluster (SSL)

- 세션 저장을 Redis에 할 계획인지:
  [ ] 예
  [ ] 아니오 (JWT 등 다른 방식)

1.3.3 Kafka

[1-3-3. Kafka 의존성]

- Kafka 사용 여부:
  [ ] 사용함
  [ ] 사용 안함

- 브로커:
  [ ] Confluent Cloud
  [ ] 자체 Kafka
  [ ] 기타: _____

- 사용하는 토픽 목록:
  - Producer:
    1) 토픽명: _____
    2) 토픽명: _____
  - Consumer:
    1) 토픽명: _____
    2) 토픽명: _____

- Consumer Group ID (예상):
  예) colombus-<service>, ws-svc-group ...

- 트랜잭션 사용 여부:
  [ ] 사용함 (transaction-id-prefix 필요)
  [ ] 사용 안함

1.3.4 인증 / 외부 API

[1-3-4. Auth / 외부 API]

- Auth0와 직접 연동하는지:
  [ ] 예 (OAuth2 로그인/콜백 처리)
  [ ] 아니오

- 내부 JWT 검증/발급 역할:
  [ ] JWT 검증만 함
  [ ] JWT 발급도 함
  [ ] 관련 없음

- 외부 REST API 사용:
  [ ] 없음
  [ ] 있음 → 목록 작성
    - API명 / Base URL / 인증 방식
      1) _____
      2) _____

- S3 사용 여부:

[ ] 사용함 (어떤 파일을 저장/읽는지 간단히 적기)
[ ] 사용 안함

1.4 리소스 / 운영 특성 시트

> “이 서비스가 얼마나 빡세게 도는지” 감만 잡아두는 부분
>
> → 나중에 Deployment 리소스 설정할 때 참고

[1-4. 리소스 / 운영 특성]

- 예상 트래픽 유형:
  [ ] 로그인/회원가입 같이 중요도 높은 API 위주
  [ ] 내부용 API (관리, 배치, 보조 서비스)
  [ ] 실시간 스트림/웹소켓 연동
  [ ] 기타: _____

- 대략적인 RPS (없으면 감으로):
  - 평상시: _____ req/s
  - 피크: _____ req/s

- 지연 시간 민감도:
  [ ] 높음 (100ms 이내가 이상적)
  [ ] 보통 (200~500ms 정도 허용)
  [ ] 낮음 (배치/비동기 위주)

- 대략 예상하는 초기 리소스 설정:
  (완벽할 필요 없음, 감으로 적기)
  - CPU: _____ m (예: 250m, 500m)
  - Memory: _____ Mi (예: 512Mi, 1024Mi)
  - 초기 replica 수: _____ 개 (예: 1, 2, 3)

- 배포 전략:
  [ ] 항상 2개 이상 띄우고 싶음 (무중단/가용성 중요)
  [ ] 1개만 떠 있어도 됨 (배치/관리용)

1.5 엔드포인트/포트 정리 시트

> 헬스 체크 + 주요 API 몇 개를 미리 적어두면,
>
> 나중에 모니터링/테스트할 때 이 표만 보면 된다.

```
[1-5. 엔드포인트/포트 정리]

- 서버 포트:
  예) 8080

- 헬스 체크 엔드포인트:
  - liveness:   예) /actuator/health/liveness
  - readiness:  예) /actuator/health/readiness
  - 전체 health: 예) /actuator/health

- 주요 API 엔드포인트 예시:
  (테스트용으로 자주 호출할 것들)

  1) [GET]  /api/<service>/...
     설명: _____

  2) [POST] /api/<service>/...
     설명: _____

  3) [기타] _____
```

1.6 1장 완료 체크리스트

아래 항목이 모두 **YES**면 1장은 끝난 거다.

```
[1장 체크리스트]

[ ] 서비스 메타 정보 시트를 작성했다.
[ ] 서비스가 사용하는 DB/Redis/Kafka/Auth/S3 의존성을 표로 정리했다.
[ ] 이 서비스가 어떤 역할(API/워커/배치)인지 명확하게 체크했다.
[ ] 대략적인 리소스 사용량(감)과 replica 개수에 대한 생각을 적어놨다.
[ ] 기본 포트/헬스 엔드포인트/테스트용 API 몇 개를 적어놨다.
```

2장. 애플리케이션 설정 포팅 (env 기반으로 정리)

2.1 프로필 파일 구조 만들기

1. `src/main/resources` 아래에 파일 3개를 맞춘다.

```
src/main/resources/
  application.yml
  application-local.yml
  application-prod.yml
```

1. 기존에 `application.yml` 하나에 다 섞여 있으면:
   - 공통값만 `application.yml` 로 남기고
   - 로컬/운영 값은 각각 `-local` , `-prod` 로 옮긴다.

2.2 공통 env 키 이름부터 확정하기

1장에서 정리한 의존성 기준으로, **env 키 이름을 먼저 박아둔다.**

(값은 나중에 Secrets Manager에서 채울 거고, 여기서는 “이름만 확정”)

필요한 것들만 골라서 쓰면 된다.

```
[필수 후보 – DB]
POSTGRES_HOST
POSTGRES_PORT
POSTGRES_DB
POSTGRES_USER
POSTGRES_PASSWORD

[필수 후보 – Redis]
REDIS_HOST
REDIS_PORT
REDIS_PASSWORD
SPRING_DATA_REDIS_SSL_ENABLED   (예: true/false)
```

```
[필수 후보 – Kafka]
KAFKA_BOOTSTRAP_SERVERS
KAFKA_ACCESS_KEY
KAFKA_SECRET_KEY
KAFKA_SECURITY_PROTOCOL        (예: SASL_SSL)
KAFKA_SASL_MECHANISM           (예: PLAIN)
KAFKA_GROUP_ID
KAFKA_CLIENT_ID
KAFKA_TX_PREFIX                (트랜잭션 쓰면)

[필수 후보 – Auth0]
AUTH0_CLIENT_ID
AUTH0_CLIENT_SECRET
AUTH0_DOMAIN                   (또는 AUTH0_ISSUER_URI)

[필수 후보 – 내부 JWT]
INTERNAL_JWT_ISSUER
INTERNAL_JWT_AUDIENCE
INTERNAL_JWT_KEY_PATH          (.key 폴더 경로 등)

[필수 후보 – S3]
AWS_ACCESS_KEY_ID              (IRSA 쓰면 나중에 제거)
AWS_SECRET_ACCESS_KEY
AWS_REGION
S3_BUCKET_NAME
```

👉 **이름만 일단 확정**해두고, 이걸 기준으로 2.3에 YAML 템플릿을 맞춘다.

2.3 `application-prod.yml` 템플릿 만들기

운영(EKS)에서 쓸 설정은 전부 **env placeholder만** 있게 만든다.

필요한 블록만 골라서 쓰면 된다.

```
# src/main/resources/application-prod.yml

server:
  port: 8080
```

```
spring:
  profiles:
    active: prod

  datasource:
    url: jdbc:postgresql://${POSTGRES_HOST}:${POSTGRES_PORT}/${POS
TGRES_DB}
    username: ${POSTGRES_USER}
    password: ${POSTGRES_PASSWORD}
    driver-class-name: org.postgresql.Driver

  data:
    redis:
      host: ${REDIS_HOST}
      port: ${REDIS_PORT:6379}
      password: ${REDIS_PASSWORD}
      ssl:
        enabled: ${SPRING_DATA_REDIS_SSL_ENABLED:true}

  kafka:
    bootstrap-servers: ${KAFKA_BOOTSTRAP_SERVERS}
    properties:
      security.protocol: ${KAFKA_SECURITY_PROTOCOL:SASL_SSL}
      sasl.mechanism: ${KAFKA_SASL_MECHANISM:PLAIN}
      sasl.jaas.config: >
        org.apache.kafka.common.security.plain.PlainLoginModule required
        username='${KAFKA_ACCESS_KEY}'
        password='${KAFKA_SECRET_KEY}';
    consumer:
      group-id: ${KAFKA_GROUP_ID}
      auto-offset-reset: latest
    producer:
      transaction-id-prefix: ${KAFKA_TX_PREFIX:colombus_tx_}

  security:
    oauth2:
      client:
```

```
      registration:
        auth0:
          client-id: ${AUTH0_CLIENT_ID}
          client-secret: ${AUTH0_CLIENT_SECRET}
          scope: openid,profile,email
          redirect-uri: "{baseUrl}/login/oauth2/code/auth0"
      provider:
        auth0:
          issuer-uri: https://${AUTH0_DOMAIN}/

management:
  endpoints:
    web:
      exposure:
        include: health,info
  endpoint:
    health:
      probes:
        enabled: true
```

- 안 쓰는 블록(DB/Redis/Kafka/Auth0 등)은 과감히 삭제해도 된다.
- Auth0 안 쓰는 서비스면 `spring.security.oauth2` 부분 전체 제거.

2.4 `application.yml` (공통) 최소화

여기에는 **환경에 상관없는 공통 설정**만 둔다.

```
apiVersion: external-secrets.io/v1
kind: ExternalSecret
metadata:
  name: gateway-app-secret
  namespace: colombus
spec:
  refreshInterval: 1h
  secretStoreRef:
    name: colombus-asm-store
    kind: ClusterSecretStore
  target:
```

```
    name: gateway-app-secret
    creationPolicy: Owner
  dataFrom:
    - extract:
        key: colombus/gateway
---
apiVersion: v1
kind: Service
metadata:
  name: gateway
  namespace: colombus
  labels: { app: gateway }
spec:
  type: ClusterIP
  selector: { app: gateway }
  ports:
    - name: http
      port: 80
      targetPort: 8080
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: gateway
  namespace: colombus
  labels: { app: gateway }
spec:
  replicas: 1
  selector:
    matchLabels: { app: gateway }
  template:
    metadata:
      labels: { app: gateway }
    spec:
      nodeSelector:
        kubernetes.io/arch: arm64
      containers:
        - name: gateway
```

```
        image: 970103397792.dkr.ecr.ap-northeast-2.amazonaws.com/colo
mbus/gateway:latest
        imagePullPolicy: Always
        ports:
          - containerPort: 8080
        env:
          - name: SERVER_PORT      # ★ Spring이 server.port 로 인식하는 환경
변수
            value: "8080"
        envFrom:
          - secretRef: { name: colombus-global-secret }
          - secretRef: { name: gateway-app-secret }
        readinessProbe:
          httpGet: { path: /actuator/health, port: 8080 }
          initialDelaySeconds: 10
          periodSeconds: 10
        livenessProbe:
          httpGet: { path: /actuator/health, port: 8080 }
          initialDelaySeconds: 30
          periodSeconds: 30
```

```
# src/main/resources/application.yml

spring:
  application:
    name: <서비스이름>  # 예: user, world, paint

server:
  port: 8080

management:
  endpoints:
    web:
      exposure:
        include: health,info
  endpoint:
    health:
```

```
    probes:
      enabled: true
```

- 여기서 DB/Redis/Kafka 설정은 최대한 빼고, prod/local 에서만 넣는 걸 목표로.

2.5 `application-local.yml` 템플릿 만들기

로컬 개발용.

편한대로 써도 되지만, **운영이랑 구조는 비슷하게** 맞춰두는 게 나중에 편하다.

```
# src/main/resources/application-local.yml

server:
  port: 8080

spring:
  profiles:
    active: local

  datasource:
    url: jdbc:postgresql://localhost:5432/<local_db_name>
    username: local
    password: local
    driver-class-name: org.postgresql.Driver

  data:
    redis:
      host: localhost
      port: 6379
      ssl:
        enabled: false

  kafka:
    bootstrap-servers: localhost:9092
    consumer:
      group-id: local-<서비스이름>
      auto-offset-reset: earliest
```

```
management:
  endpoints:
    web:
      exposure:
        include: health,info
  endpoint:
    health:
      probes:
        enabled: true
```

- 여기서는 굳이 env 안 써도 되지만, 써도 상관없다.
- 개발자가 docker-compose로 로컬 DB/Redis/Kafka 띄워놓고 쓰는 구조면 이쪽에 맞춰 작성.

2.6 Actuator / 헬스 엔드포인트 정리

1. `build.gradle.kts` (또는 `build.gradle`) 에 Actuator 추가:

```
dependencies {
    implementation("org.springframework.boot:spring-boot-starter-actuator")
}
```

1. `application.yml` 또는 `application-*.yml` 에 이미 management 블록이 없으면,
   2.4/2.5의 management 설정 그대로 넣는다.

이러면:

- `/actuator/health`
- `/actuator/health/liveness`
- `/actuator/health/readiness`

까지 자동으로 열린다.

2.7 로컬에서 빠르게 테스트

구조가 제대로 섰는지 **로컬에서 한 번만 체크**한다.

1. `local` 프로필 실행:

```
SPRING_PROFILES_ACTIVE=local \
./gradlew bootRun
```

1. 다른 터미널에서:

```
curl -s http://localhost:8080/actuator/health | jq .
```

- `status: "UP"` 나오면 OK.

(Windows라면 `set SPRING_PROFILES_ACTIVE=local` 후 `gradlew bootRun`)

2.8 2장 완료 체크리스트

```
[2장 체크리스트]

[ ] src/main/resources에 application.yml / application-local.yml / application-prod.yml 3개가 준비됐다.
[ ] 운영에서 사용하는 설정(DB/Redis/Kafka/Auth0/S3 등)은 모두 env placeholder(${...})로 바뀌었다.
[ ] 공통 설정(포트, 앱 이름, actuator 기본값)은 application.yml로 정리됐다.
[ ] 로컬 프로필(local)로 앱을 실행했고, /actuator/health가 정상 응답하는 걸 확인했다.
```

3장. Docker 이미지 만들기 (로컬 검증용)

3.1 Dockerfile 위치 정하기

1. 서비스 코드 루트(예: `repo/BE/services/<service-name>`) 로 이동.
2. 거기에 `Dockerfile` 파일 하나 만든다.

```
repo/BE/services/<service-name>/
  ├─ build.gradle[.kts] / pom.xml
  ├─ src/
  └─ Dockerfile        ← 이거
```

3.2 표준 Dockerfile 템플릿 붙여넣기

아래를 **그대로 복붙**해서 `Dockerfile`에 넣고,

`<service-name>` 정도만 바꿔주면 된다.

```
# ---------- Build stage ----------
FROM amazoncorretto:21-alpine AS build
WORKDIR /workspace
ENV GRADLE_USER_HOME=/home/gradle/.gradle
RUN apk add --no-cache bash curl unzip zip tar findutils
# Gradle/설정
COPY gradle ./gradle
COPY gradlew .
RUN sed -i "s/\r$//" gradlew && chmod +x gradlew
COPY settings.gradle.kts .
COPY build.gradle.kts .

# 모듈별 gradle.kts만 먼저 복사해서 의존성 캐시
COPY services/media/build.gradle.kts services/media/
COPY common/*/build.gradle.kts      common/

COPY contracts/*/build.gradle.kts   contracts/
# 윈도 줄바꿈 대비

# 의존성만 먼저 내려 캐시층 확보
RUN --mount=type=cache,target=/home/gradle/.gradle \
    ./gradlew :services:<service-name>:dependencies --no-daemon

# 실제 소스 복사
COPY services/<service-name>/src services/<service-name>/src
COPY common              common

COPY contracts           contracts
# media 빌드
RUN --mount=type=cache,target=/home/gradle/.gradle \
    ./gradlew :services:<service-name>:bootJar \
      --no-daemon -x test \
      -PskipJooq && \
```

```
    cp services/<service-name>/build/libs/*.jar /tmp/app.jar


# ---------- Run stage ----------
FROM amazoncorretto:21-alpine
RUN addgroup -g 10001 -S app || true && adduser -u 10001 -S -D -H -G ap
p app || true
WORKDIR /app
COPY --from=build /tmp/app.jar /app/app.jar
EXPOSE 8088
USER 10001:10001
ENTRYPOINT ["java","-XX:+UseG1GC","-XX:MaxRAMPercentage=75","-ja
r","/app/app.jar"]
```

- Gradle이 아니라 Maven이면 위에서 `gradle` 부분만 `mvn package` 쪽으로 바꾸면 됨 (필요하면 나중에 따로 챕터에 추가).

3.3 로컬에서 Docker 이미지 빌드

서비스 루트로 가서:

```
cd repo/BE/services/<service-name>

# <tag>는 편한대로, 예: test 또는 0.0.1
docker build -t <service-name>:<tag> .
```

예:

```
docker build -t user-svc:test .
```

- 이 명령이 에러 없이 끝나면 **이미지 빌드 성공**.

3.4 컨테이너 실행 & 헬스체크

1. 컨테이너 실행:

```
docker run --rm -p 8080:8080 \
  -e SPRING_PROFILES_ACTIVE=local \
  <service-name>:<tag>
```

예:

```
docker run --rm -p 8080:8080 \
  -e SPRING_PROFILES_ACTIVE=local \
  user-svc:test
```

1. 다른 터미널에서 헬스 체크:

```
curl -s http://localhost:8080/actuator/health
```

- `{"status":"UP"}` 또는 비슷한 JSON 나오면 OK.

1. 필요하면 주요 API도 한 번 찍어본다:

```
curl -v http://localhost:8080/api/<service>/...
```

3.5 ECR 푸시를 위한 태그 준비

4장에서 ECR에 올릴 때 쓸 태그 패턴을 미리 맞춰두면 편하다.

추천 패턴:

```
<ACCOUNT_ID>.dkr.ecr.ap-northeast-2.amazonaws.com/colombus/<service-name>:<tag>
```

예를 들어, 로컬 이미지 태그를 이렇게 한 번 맞춰서 찍어본다:

```
ACCOUNT_ID=970103397792
REGION=ap-northeast-2
SERVICE=user-svc
TAG=test

docker tag ${SERVICE}:${TAG} \
```

```
  ${ACCOUNT_ID}.dkr.ecr.${REGION}.amazonaws.com/colombus/${SERVICE}:${TAG}
```

(실제 `docker push`는 4장에서 다룰 거라 여기서는 태그만 연습 느낌으로 맞춰두면 됨.)

3.6 3장 완료 체크리스트

[3장 체크리스트]

[ ] 서비스 루트에 Dockerfile을 만들었다.
[ ] docker build로 <service-name>:<tag> 이미지를 빌드했다.
[ ] docker run + SPRING_PROFILES_ACTIVE=local로 컨테이너를 띄워봤다.
[ ] http://localhost:8080/actuator/health 를 호출해서 UP 응답을 받았다.
[ ] ECR용 태그 형식을 이해했고, tag 명명 규칙을 대략 정했다.

4장. ECR 레포지토리 생성 & Docker 이미지 푸시

4.1 준비: 공통 변수 먼저 정리

매번 쓰기 귀찮으니까 쉘에서 공통 변수 하나 세트해두는 걸 추천.

```
# 계정 / 리전 / 서비스 이름 / 태그 한 번에 정리
ACCOUNT_ID=970103397792
REGION=ap-northeast-2
SERVICE=<service-name>      # 예: user-svc, world-svc
TAG=<tag>                   # 예: test, 0.1.0, git SHA 등

ECR_HOST=${ACCOUNT_ID}.dkr.ecr.${REGION}.amazonaws.com
REPO_NAME=colombus/${SERVICE}
IMAGE=${ECR_HOST}/${REPO_NAME}:${TAG}
```

- `SERVICE`, `TAG`만 바꾸면 나머지는 자동으로 따라간다고 생각하면 편함.

4.2 ECR 리포지토리 생성

4.2.1 레포지토리 이름 규칙

- **패턴**: `colombus/<service-name>`
  - 예: `colombus/user-svc`, `colombus/world-svc`

4.2.2 CLI로 생성

한 번만 실행하면 된다. 이미 있으면 에러 나니까, 에러면 "아 이미 있구나" 정도로 생각하면 됨.

```
aws ecr create-repository \
  --repository-name ${REPO_NAME} \
  --image-scanning-configuration scanOnPush=true \
  --region ${REGION}
```

- 이미 있으면 `RepositoryAlreadyExistsException` 뜸 → 무시해도 됨.

4.3 ECR 로그인

Docker가 ECR에 push 하려면 로그인 필요.

```
aws ecr get-login-password --region ${REGION} \
  | docker login \
    --username AWS \
    --password-stdin ${ECR_HOST}
```

- `Login Succeeded` 나오면 성공.

4.4 로컬 이미지에 ECR 태그 달기

3장에서 만든 로컬 이미지가 `SERVICE:TAG` 라고 가정하고:

```
docker tag ${SERVICE}:${TAG} ${IMAGE}
```

예:

```
# 예시
SERVICE=user-svc
TAG=test
```

```
ACCOUNT_ID=970103397792
REGION=ap-northeast-2
ECR_HOST=${ACCOUNT_ID}.dkr.ecr.${REGION}.amazonaws.com
REPO_NAME=colombus/${SERVICE}
IMAGE=${ECR_HOST}/${REPO_NAME}:${TAG}

docker tag ${SERVICE}:${TAG} ${IMAGE}
```

4.5 ECR로 Docker 이미지 푸시

그냥 push:

```
docker push ${IMAGE}
```

예:

```
docker push 970103397792.dkr.ecr.ap-northeast-2.amazonaws.com/colombus/user-svc:test
```

- 처음 푸시하면 레이어 올리느라 시간 좀 걸릴 수 있음.
- 성공하면 마지막에 `pushed` 로그들 나오고 끝.

4.6 이미지 푸시 확인

CLI로 간단히 확인:

```
aws ecr describe-images \
  --repository-name ${REPO_NAME} \
  --region ${REGION} \
  --query 'imageDetails[?contains(imageTags, `"${TAG}"`)]'
```

- `imageDigest`, `imageTags` 등이 출력되면 성공.

또는 콘솔에서:

- ECR → `colombus/<service-name>` 레포지토리 들어가서

  → `TAG`가 원하는 값으로 떠 있는지 확인.

4.7 4장 완료 체크리스트

[4장 체크리스트]

[ ] ACCOUNT_ID / REGION / SERVICE / TAG 변수를 정리했다.
[ ] ECR에 colombus/<service-name> 레포지토리를 만들었다(또는 기존 레포지토리를 확인했다).
[ ] aws ecr get-login-password로 Docker ECR 로그인을 했다.
[ ] 로컬 이미지(<service-name>:<tag>)에 ECR 태그를 붙였다.
[ ] docker push로 ECR에 이미지를 올렸다.
[ ] aws ecr describe-images 또는 콘솔에서 TAG가 올라간 걸 확인했다.

5장. Secrets Manager & ExternalSecret 연결

5.1 시크릿 키 이름 규칙 정하기

서비스마다 **Secrets Manager key 이름**은 이렇게 맞춘다:

```
colombus/<service>
예) colombus/user
    colombus/world
    colombus/paint
```

- `<service>` 는 1장에서 정리한 "서비스 코드명" 기준 (K8s service랑 살짝 달라도 괜찮음, 통일만 잘 하면 됨).

K8s에 생성될 Secret 이름은:

```
<service>-app-secret
예) user-app-secret
    world-app-secret
```

5.2 Secrets Manager에 넣을 JSON 구조 만들기

2장에서 정리한 env 키들 기준으로, **JSON 한 덩어리**를 만든다.

예시(필요한 것만 골라서 쓰면 됨):

```
{
  "POSTGRES_HOST": "colombus-test-db.c9ie60uac74v.ap-northeast-2.rds.amazonaws.com",
  "POSTGRES_PORT": "5432",
  "POSTGRES_DB": "colombus_user",
  "POSTGRES_USER": "ssafy",
  "POSTGRES_PASSWORD": "********",

  "REDIS_HOST": "clustercfg.colombus-redis.zwrbq8.apn2.cache.amazonaws.com",
  "REDIS_PORT": "6379",
  "REDIS_PASSWORD": "********",
  "SPRING_DATA_REDIS_SSL_ENABLED": "true",

  "KAFKA_BOOTSTRAP_SERVERS": "pkc-921jm.us-east-2.aws.confluent.cloud:9092",
  "KAFKA_ACCESS_KEY": "XXXXXXXX",
  "KAFKA_SECRET_KEY": "YYYYYYYY",
  "KAFKA_SECURITY_PROTOCOL": "SASL_SSL",
  "KAFKA_SASL_MECHANISM": "PLAIN",
  "KAFKA_GROUP_ID": "colombus-<service>",
  "KAFKA_CLIENT_ID": "<service>-client",
  "KAFKA_TX_PREFIX": "colombus_tx_",

  "AUTH0_CLIENT_ID": "aaaaaa",
  "AUTH0_CLIENT_SECRET": "bbbbbb",
  "AUTH0_DOMAIN": "your-tenant.auth0.com",

  "INTERNAL_JWT_ISSUER": "colombus-auth",
  "INTERNAL_JWT_AUDIENCE": "colombus-internal",
  "INTERNAL_JWT_KEY_PATH": "/app/.key/<service>",

  "AWS_REGION": "ap-northeast-2",
  "S3_BUCKET_NAME": "colombus-<bucket>"
}
```

- 서비스마다 필요한 키만 남기고, 안 쓰는 건 빼버리면 됨.
- 값은 당연히 실제 값으로 채움.

5.3 AWS CLI로 Secrets Manager에 저장

CLI 한 방으로 JSON을 넣을 수 있다.

```
SECRET_NAME="colombus/<service>"  # 예: colombus/user
REGION="ap-northeast-2"

aws secretsmanager create-secret \
  --name "${SECRET_NAME}" \
  --region "${REGION}" \
  --secret-string '{
    "POSTGRES_HOST":"...",
    "POSTGRES_PORT":"5432",
    "POSTGRES_DB":"...",
    "POSTGRES_USER":"...",
    "POSTGRES_PASSWORD":"...",
    "REDIS_HOST":"...",
    "REDIS_PORT":"6379",
    "REDIS_PASSWORD":"...",
    "SPRING_DATA_REDIS_SSL_ENABLED":"true"
  }'
```

- 이미 있다면 `create-secret` 대신 `put-secret-value` 사용:

```
aws secretsmanager put-secret-value \
  --secret-id "${SECRET_NAME}" \
  --region "${REGION}" \
  --secret-string '{
    ... 같은 구조 ...
  }'
```

5.4 ExternalSecret 템플릿 작성

이미 `auth-svc` 등에서 쓰고 있는 패턴을 그대로 복붙하면 된다.

5.4.1 기본 템플릿

`infra/manifests/services/<service>.yaml` 같은 곳에 아래 블록 추가:

```
apiVersion: external-secrets.io/v1
kind: ExternalSecret
metadata:
  name: <service>-app-secret
  namespace: colombus
spec:
  refreshInterval: 1h
  secretStoreRef:
    name: colombus-asm-store
    kind: ClusterSecretStore
  target:
    name: <service>-app-secret
    creationPolicy: Owner
  dataFrom:
    - extract:
        key: colombus/<service>
```

예: user 서비스라면

```
apiVersion: external-secrets.io/v1
kind: ExternalSecret
metadata:
  name: user-app-secret
  namespace: colombus
spec:
  refreshInterval: 1h
  secretStoreRef:
    name: colombus-asm-store
    kind: ClusterSecretStore
  target:
    name: user-app-secret
    creationPolicy: Owner
  dataFrom:
```

```
  - extract:
      key: colombus/user
```

- `secretStoreRef.name: colombus-asm-store` 는 이미 클러스터에 있는 ClusterSecretStore 이름 그대로 사용(지금 쓰고 있는 값).

5.5 ExternalSecret 적용 & Secret 생성 확인

1. 매니페스트 적용:

```
kubectl -n colombus apply -f services/<service>.yaml
# 또는 ExternalSecret 블록만 따로 yaml로 관리했다면 그 파일
```

1. ExternalSecret 리소스 확인:

```
kubectl -n colombus get externalsecret
kubectl -n colombus get externalsecret <service>-app-secret -o yaml
```

1. 실제 K8s Secret 생성됐는지 확인:

```
kubectl -n colombus get secret <service>-app-secret
```

1. 값 간단히 확인 (key 이름만):

```
kubectl -n colombus get secret <service>-app-secret -o jsonpath='{.data}'
| jq .
```

- `POSTGRES_HOST`, `REDIS_HOST` 같은 키들이 base64로 들어 있으면 정상.

5.6 Pod 안에서 env로 잘 들어오는지 테스트 (선택)

Deployment 만들기 전에, **임시 디버그 Pod**에서 Secret을 env로 마운트해서 확인해볼 수 있다.

```
kubectl -n colombus run debug-env \
  --image=busybox \
  --restart=Never \
```

```
  --env-from=secretRef:name=<service>-app-secret \
  -- sleep 3600
```

그 다음:

```
kubectl -n colombus exec -it debug-env -- sh
# 컨테이너 안에서
env | grep POSTGRES
env | grep KAFKA
```

확인 끝났으면:

```
kubectl -n colombus delete pod debug-env
```

5.7 5장 완료 체크리스트

```
[5장 체크리스트]

[ ] 서비스별 Secrets Manager 키 이름을 colomubus/<service> 형식으로 정했다.
[ ] 해당 키에 필요한 env 값들을 JSON으로 넣었다.
[ ] ExternalSecret 리소스를 작성했다 (name: <service>-app-secret, key: colo
mbus/<service>).
[ ] kubectl로 ExternalSecret을 적용했고, 실제 K8s Secret(<service>-app-secr
et)이 생성된 걸 확인했다.
[ ] (선택) debug Pod를 만들어 Secret이 env로 잘 주입되는지 확인했다.
```

6장. Kubernetes Deployment & Service 생성 (ARM64 전용)

6.1 공통 이름/값 먼저 정해두기

쉘에서 공통 변수 세트:

```
NAMESPACE=colombus
SERVICE=<service-name>          # 예: user-svc, world-svc
```

```
APP_NAME=<app-label>          # 보통 SERVICE랑 같게: user-svc
IMAGE="<ECR>/<repo>:<tag>"    # 4장에서 만든 이미지
SECRET_NAME=<service>-app-secret  # 5장에서 만든 시크릿 이름
```

예시:

```
NAMESPACE=colombus
SERVICE=user-svc
APP_NAME=user-svc
IMAGE="970103397792.dkr.ecr.ap-northeast-2.amazonaws.com/colombus/user-svc:test"
SECRET_NAME=user-app-secret
```

6.2 Service YAML 템플릿 작성

`infra/manifests/services/<service>.yaml` 파일에 **Service 블록**부터 넣는다

(5장에서 ExternalSecret 이미 있다면 그 아래/위에 붙이면 됨).

```
apiVersion: v1
kind: Service
metadata:
  name: <service>-svc
  namespace: colombus
  labels:
    app: <service>-svc
spec:
  type: ClusterIP
  selector:
    app: <service>-svc
  ports:
    - name: http
      port: 8080
      targetPort: 8080
```

예) user 서비스:

```
apiVersion: v1
kind: Service
metadata:
  name: user-svc
  namespace: colombus
  labels:
    app: user-svc
spec:
  type: ClusterIP
  selector:
    app: user-svc
  ports:
    - name: http
      port: 8080
      targetPort: 8080
```

- selector의 `app: <service>-svc` 가 Deployment template의 label이랑 반드시 일치해야 한다.

6.3 Deployment YAML 템플릿 작성 (ARM64 + envFrom)

같은 파일에 이어서 **Deployment 블록**을 추가한다.

(nodeSelector는 1.34에서도 그대로 표준 방식으로 사용 가능.)

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: <service>-svc
  namespace: colombus
  labels:
    app: <service>-svc
spec:
  replicas: 1
  selector:
    matchLabels:
      app: <service>-svc
```

```
  template:
    metadata:
      labels:
        app: <service>-svc
    spec:
      nodeSelector:
        kubernetes.io/arch: arm64
        karpenter.sh/nodepool: arm-general
      containers:
        - name: <service>-container
          image: <IMAGE>
          imagePullPolicy: IfNotPresent
          env:
            - name: SPRING_PROFILES_ACTIVE
              value: "prod"
          envFrom:
            - secretRef:
                name: <service>-app-secret
          ports:
            - containerPort: 8080
              name: http
          resources:
            requests:
              cpu: "250m"
              memory: "512Mi"
            limits:
              cpu: "500m"
              memory: "1024Mi"
          livenessProbe:
            httpGet:
              path: /actuator/health/liveness
              port: 8080
            initialDelaySeconds: 30
            periodSeconds: 10
            timeoutSeconds: 2
            failureThreshold: 3
          readinessProbe:
            httpGet:
```

```
            path: /actuator/health/readiness
            port: 8080
          initialDelaySeconds: 10
          periodSeconds: 5
          timeoutSeconds: 2
          failureThreshold: 3
```

예) user-svc에 적용한 완성 예:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: user-svc
  namespace: colombus
  labels:
    app: user-svc
spec:
  replicas: 1
  selector:
    matchLabels:
      app: user-svc
  template:
    metadata:
      labels:
        app: user-svc
    spec:
      nodeSelector:
        kubernetes.io/arch: arm64
        karpenter.sh/nodepool: arm-general
      containers:
        - name: user-container
          image: 970103397792.dkr.ecr.ap-northeast-2.amazonaws.com/colo
mbus/user-svc:test
          imagePullPolicy: IfNotPresent
          env:
            - name: SPRING_PROFILES_ACTIVE
              value: "prod"
          envFrom:
```

```
    - secretRef:
        name: user-app-secret
  ports:
    - containerPort: 8080
      name: http
  resources:
    requests:
      cpu: "250m"
      memory: "512Mi"
    limits:
      cpu: "500m"
      memory: "1024Mi"
  livenessProbe:
    httpGet:
      path: /actuator/health/liveness
      port: 8080
    initialDelaySeconds: 30
    periodSeconds: 10
    timeoutSeconds: 2
    failureThreshold: 3
  readinessProbe:
    httpGet:
      path: /actuator/health/readiness
      port: 8080
    initialDelaySeconds: 10
    periodSeconds: 5
    timeoutSeconds: 2
    failureThreshold: 3
```

- Probe 필드는 1.34에서도 동일한 `httpGet + initialDelaySeconds/periodSeconds` 형식으로 사용.

6.4 매니페스트 적용

`infra/manifests` 루트에서:

```
kubectl -n ${NAMESPACE} apply -f services/<service>.yaml
```

적용 후 상태 확인:

```
kubectl -n ${NAMESPACE} get svc,deployment,pods -l app=${APP_NAME}
```

6.5 롤아웃 상태 확인 & 기본 디버깅

1. 롤아웃:

```
kubectl -n ${NAMESPACE} rollout status deploy/${SERVICE}
```

1. Pod 상태가 `Running` 인지 확인:

```
kubectl -n ${NAMESPACE} get pods -l app=${APP_NAME}
```

1. 만약 `CrashLoopBackOff` / `Error` 면 로그 먼저 본다:

```
POD=$(kubectl -n ${NAMESPACE} get pods -l app=${APP_NAME} -o jsonpath='{.items[0].metadata.name}')

kubectl -n ${NAMESPACE} logs ${POD}
```

1. 설정(env) 문제 같으면:

```
kubectl -n ${NAMESPACE} exec -it ${POD} -- env | grep -E 'POSTGRES|REDIS|KAFKA|AUTH0'
```

6.6 클러스터 내부에서 Service 통신 테스트

Service 이름으로 헬스 체크를 때려본다.

1. 임시 네트워크 디버그 Pod 생성:

```
kubectl -n ${NAMESPACE} run net-debug \
  --image=busybox \
  --restart=Never \
  --command -- sh -c "sleep 3600"
```

1. 그 안에서 curl:

```
kubectl -n ${NAMESPACE} exec -it net-debug -- sh

# 컨테이너 안에서:
wget -qO- http://<service>-svc:8080/actuator/health
wget -qO- http://<service>-svc:8080/actuator/health/readiness
```

1. 끝나면 정리:

```
kubectl -n ${NAMESPACE} delete pod net-debug
```

6.7 6장 완료 체크리스트

```
[6장 체크리스트]

[ ] Service 리소스(ClusterIP)가 생성되었고, selector/app 라벨이 Deployment와 일치한다.
[ ] Deployment에 image, envFrom(secret), SPRING_PROFILES_ACTIVE=prod 가 설정되어 있다.
[ ] nodeSelector로 kubernetes.io/arch=arm64, karpenter.sh/nodepool=arm-general 이 설정되어 있다.
[ ] livenessProbe / readinessProbe 가 /actuator/health/* 엔드포인트로 붙어 있다.
[ ] kubectl rollout status 로 Deployment 롤아웃 완료를 확인했다.
[ ] Pod가 ARM64 노드(arm-general) 위에서 Running 상태인 걸 확인했다.
[ ] net-debug Pod에서 http://<service>-svc:8080/actuator/health 호출이 성공하는 걸 확인했다.
```

7장. Gateway & Ingress 라우팅 연결

7.1 API 경로(prefix) 먼저 정하기

1장에서 적어둔 서비스 역할 보면서, **공식 경로 하나를 박아두자.**

```
[7-1. API 경로 설계]
```

```
- 외부에서 쓸 기본 prefix:
  예) /api/user/**, /api/world/**, /api/paint/**

- 내부 실제 서비스:
  예) user-svc:8080, world-svc:8080

- 게이트웨이에서 strip 할 prefix:
  예) /api/user → / (서비스 기준)
```

경로 예시 패턴:

- 유저: `/api/user/** → user-svc:8080`
- 월드: `/api/world/** → world-svc:8080`
- 그림: `/api/paint/** → paint-svc:8080`

이 표만 정해두고, 아래 설정에 그대로 박는다.

7.2 Spring Cloud Gateway 라우트 추가 (gateway-svc)

> Colombus에서 보통 gateway-svc 가 /api/** 전부 받아서 내부 서비스로 포워딩.

7.2.1 gateway-svc `application.yml` 수정

gateway 코드가 있는 repo(예: `services/gateway`) 의 `application.yml` 에 라우트 추가.

```
spring:
  cloud:
    gateway:
      routes:
        - id: <service>-route
          uri: http://<service>-svc:8080
          predicates:
            - Path=/api/<service>/**
          filters:
            - StripPrefix=2   # /api/<service>/ 를 떼고 서비스에 넘길지 여부
```

예) user 서비스:

```
spring:
  cloud:
    gateway:
      routes:
        - id: user-route
          uri: http://user-svc:8080
          predicates:
            - Path=/api/user/**
          filters:
            - StripPrefix=2     # /api/user/xxx → /xxx 로 user-svc 에 전달
```

> StripPrefix 개수 = /api/user 처럼 "/" 기준 segment 수
>
> (필요 없으면 filters 전체를 빼도 됨)

7.2.2 gateway-svc 재배포

이미 Jenkins 파이프라인 있으면 거치고, 아니면 수동:

1. gateway-svc Docker 이미지 새로 빌드 + ECR 푸시 (3~4장 플로우)
2. gateway-svc Deployment 이미지 태그만 바꾸고 적용:

```
kubectl -n colombus set image deploy/gateway-svc \
  gateway-container=<ECR_HOST>/colombus/gateway-svc:<tag>

kubectl -n colombus rollout status deploy/gateway-svc
```

7.3 BFF에서 내부 서비스 호출 붙이기 (BFF 사용하는 경우)

BFF가 REST로 다른 서비스 호출하는 구조라면,

BFF → `<service>-svc:8080` **호출 코드**만 맞춰주면 된다.

예) BFF의 `application.yml` 에 base-url 설정:

```
service:
  user:
    base-url: http://user-svc:8080
```

그리고 코드에서:

```
WebClient userClient = WebClient.builder()
    .baseUrl(serviceUserBaseUrl)   // http://user-svc:8080
    .build();
```

> 핵심: K8s 안에서는 항상 http://<service>-svc:8080 으로 호출.

7.4 Ingress/ALB 쪽 확인 (필요한 경우만 수정)

이미 **Ingress → gateway-svc** 하나로 `/api/**` 를 모두 받고 있으면,

게이트웨이에 라우트만 추가하면 Ingress는 손댈 필요 없는 경우가 많다.

그래도 새로 추가해야 한다면, 예시는 이렇게:

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: colombus-api-ingress
  namespace: colombus
  annotations:
    kubernetes.io/ingress.class: alb
spec:
  rules:
    - host: api.3dplace.kr
      http:
        paths:
          - path: /api
            pathType: Prefix
            backend:
              service:
                name: gateway-svc
                port:
                  number: 8080
```

- 이미 이런 Ingress가 있고 `gateway-svc` 로 묶여 있다면 → **7.2만 해도 OK**.

적용:

```
kubectl -n colombus apply -f ingress.yaml
kubectl -n colombus get ingress
```

7.5 외부에서 실제 호출 테스트

7.5.1 ALB DNS / 도메인 확인

이미 앞에서 쓰던 값 재사용:

```
ALB_DNS=<내부 or 외부 ALB 도메인>  # 예: internal-k8s-colombusapp-....elb.amazonaws.com
```

Cloudflare/도메인 붙여놨다면:

- 예) `https://3dplace.kr/api/user/me`

7.5.2 curl로 스모크 테스트

gateway 없이 ALB DNS 바로 사용하는 경우:

```
curl -v http://$ALB_DNS/api/<service>/health
curl -v http://$ALB_DNS/api/<service>/...
```

도메인으로 붙인 경우:

```
curl -v https://3dplace.kr/api/<service>/health
```

- 200 + 기대하는 JSON/응답 나오면 성공.

7.6 7장 완료 체크리스트

```
[7장 체크리스트]

[ ] 이 서비스의 공식 API prefix(/api/<service>/...)를 정했다.
[ ] gateway-svc(application.yml)에 Path=/api/<service>/** → uri=http://<service>-svc:8080 라우트를 추가했다.
[ ] 필요한 경우 StripPrefix 설정을 넣어, 서비스 쪽에서 기대하는 path 구조를 맞췄다.
[ ] gateway-svc 이미지를 새 태그로 빌드/푸시하고, Deployment를 롤링 업데이트
```

했다.
[ ] (BFF 사용 시) BFF에서 http://<service>-svc:8080 으로 내부 호출하는 base-url을 설정했다.
[ ] Ingress가 gateway-svc 로 /api/** 를 이미 보내고 있는지 확인했다. (필요하면 rule 추가)
[ ] curl 로 도메인(or ALB DNS) + /api/<service>/... 를 호출해서 정상 응답을 확인했다.

8장. 스모크 테스트 & 모니터링 확인

8.1 클러스터 내부 스모크 테스트

8.1.1 net-debug Pod 만들기

```
NAMESPACE=colombus
SERVICE=<service-name>   # 예: user-svc

kubectl -n ${NAMESPACE} run net-debug \
  --image=busybox \
  --restart=Never \
  --command -- sh -c "sleep 3600"
```

8.1.2 Service 이름으로 헬스 체크

```
kubectl -n ${NAMESPACE} exec -it net-debug -- sh

# 컨테이너 안에서:
wget -qO- http://${SERVICE}:8080/actuator/health
wget -qO- http://${SERVICE}:8080/actuator/health/readiness
```

- 둘 다 응답이 나오면 OK (`{"status":"UP"}` 등)

끝났으면:

```
kubectl -n ${NAMESPACE} delete pod net-debug
```

8.2 외부(도메인/ALB) 스모크 테스트

8.2.1 ALB / 도메인 확인

환경에 따라 하나 골라서 쓴다.

```
ALB_DNS=<internal-ALB-DNS>     # 예: internal-k8s-colombusapp-....elb.amazonaws.com
DOMAIN=<external-domain>       # 예: 3dplace.kr
SERVICE_PREFIX=<service-path>  # 예: user, world, paint
```

8.2.2 curl 테스트

ALB DNS 기준:

```
curl -v "http://${ALB_DNS}/api/${SERVICE_PREFIX}/health"
```

도메인 기준:

```
curl -v "https://${DOMAIN}/api/${SERVICE_PREFIX}/health"
```

- HTTP 200 + JSON 나오면 OK

필요하면 실제 비즈니스 API도 한 두 개 찍어본다:

```
curl -v "https://${DOMAIN}/api/${SERVICE_PREFIX}/sample"
```

8.3 Pod 로그 확인

8.3.1 Pod 이름 가져오기

```
NAMESPACE=colombus
APP_LABEL=<service>-svc    # 예: user-svc

POD=$(kubectl -n ${NAMESPACE} get pods -l app=${APP_LABEL} -o jsonpath='{.items[0].metadata.name}')
echo $POD
```

8.3.2 로그 보기

```
kubectl -n ${NAMESPACE} logs ${POD}
```

실시간:

```
kubectl -n ${NAMESPACE} logs -f ${POD}
```

8.4 상태/리소스 사용량 확인

8.4.1 기본 상태

```
kubectl -n colombus get pods -l app=<service>-svc -o wide
kubectl -n colombus describe pod ${POD}
```

- 어느 노드(ARM64/arm-general)에 떠 있는지, restart 횟수, 이벤트 확인.

8.4.2 리소스 사용 (metrics-server 있을 때)

```
kubectl top pod -n colombus | grep <service>-svc
kubectl top node
```

- 여기서 CPU/메모리 대략 보고 6장에서 잡은 request/limit 적당했는지 감 잡기.

8.5 간단 "헬스 대시보드" 시트 만들기 (문서용)

문서에 서비스별 헬스 정보를 한 장으로 정리해두면 나중에 편하다.

```
[8-5. <service> 헬스 정리]

- 내부 헬스 체크 URL:
  - http://<service>-svc:8080/actuator/health
  - http://<service>-svc:8080/actuator/health/readiness

- 외부 헬스 체크 URL:
  - http(s)://<DOMAIN or ALB_DNS>/api/<service-prefix>/health
```

- 로그 확인 방법:
  - kubectl -n colombus logs -f <pod-name>

- 리소스 확인:
  - kubectl -n colombus get pods -l app=<service>-svc -o wide
  - kubectl top pod -n colombus | grep <service>-svc

서비스마다 이 블럭만 하나씩 복붙해서 채워두면 된다.

8.6 8장 완료 체크리스트

[8장 체크리스트]

[ ] net-debug Pod로 http://<service>-svc:8080/actuator/health 호출이 성공하는 걸 확인했다.
[ ] ALB DNS 또는 도메인 기준으로 /api/<service-prefix>/health 를 호출해 200 응답을 확인했다.
[ ] kubectl logs 로 새 서비스 Pod 로그를 볼 수 있다.
[ ] kubectl -n colombus get pods -l app=<service>-svc -o wide 로 노드/상태를 확인했다.
[ ] (metrics-server 사용 시) kubectl top pod 로 CPU/메모리 사용량을 확인했다.
[ ] 서비스별 헬스/로그/리소스 확인 방법을 문서에 정리했다.