ch4

애플리케이션을 여러 환경에 배포시 어떻게 설정할 것인가?

(일반적인 경우)

1. 다른 환경에 대한 각각의 설정 데이터를 포함하는 번들로 패키징하여 실행 시 플래그를 통해 설정을 선택 → 특정 환경에 대한 설정 데이터 업데이트 시 애플리케이션을 새로 빌드해야 함.
2. 각 환경에 대해 별도의 빌드를 생성

클라우드 네이티브 애플리케이션의 핵심 중 하나는 환경이 달라지더라도 애플리케이션 아티팩트는 동일.

→ 코드를 재빌드하지 않고도 교체할 수 있도록 외부화된 설정을 선호

- 설정 전략

설정 전략	특징
애플리케이션과 함께 패키징된 속성 파일	• 이 파일은 애플리케이션이 지원하는 설정 데이터의 명세서 역할을 할 수 있다. • 합리적인 기본값을 정의하는 데 유용하다. 이때 기본값은 주로 개발 환경을 위한 것이다.
환경 변수	• 환경 변수는 모든 운영체제에서 지원되므로 이식성이 우수하다. • 대부분의 프로그래밍 언어는 환경 변수에 접근할 수 있다. 자바에서는 `System.getenv()` 메서드를 사용한다. 스프링에서는 `Environment` 추상화를 사용할 수 있다. • 활성 프로파일, 호스트 명, 서비스 이름, 포트 번호와 같이 애플리케이션이 배포되는 인프라와 플랫폼에 따라 달라지는 설정 데이터를 정의하는 데 유용하다.
설정 서비스	• 설정 데이터 지속성, 감사, 책임을 제공한다. • 암호화 또는 전용 시크릿 볼트를 통해 시크릿 관리가 가능하다. • 연결 풀, 크리덴셜, 기능 플래그, 스레드 풀 및 URL과 같은 애플리케이션 고유의 설정 데이터를 정의하는 데 유용하다.

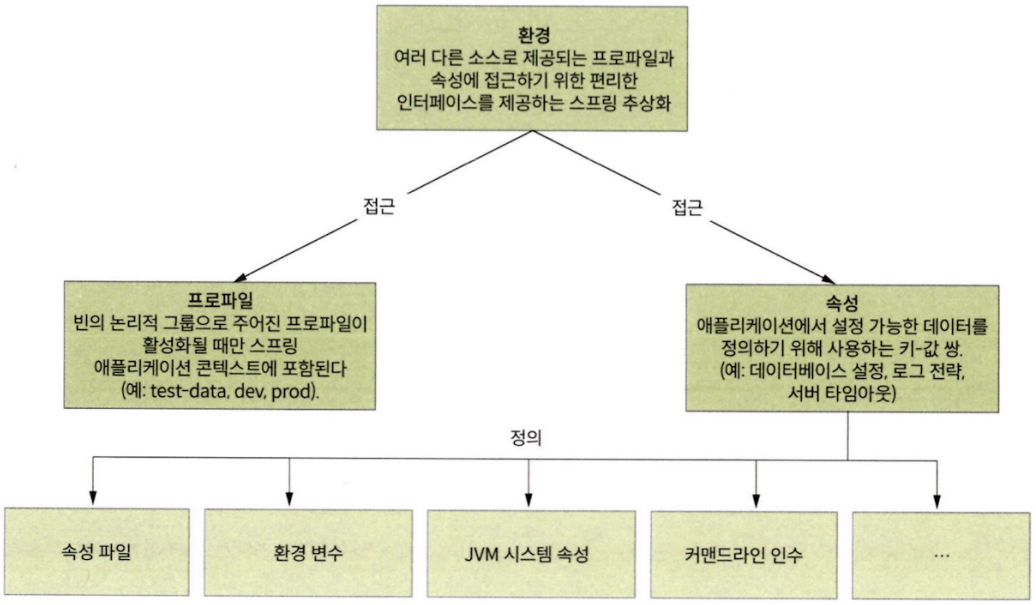

그림 4.2 Environment 인터페이스를 통해 스프링 애플리케이션 설정의 두 가지 주요 측면인 속성 및 프로파일에 대한 액세스가 가능하다.

- 스프링 애플리케이션 속성에 액세스 하는 방법
  1. `Environment` 인터페이스를 통해 속성에 접근
  2. `@Value` 애너테이션으로 속성 주입
  3. `@ConfigurationProperties` 애너테이션으로 표시된 클래스나 레코드를 통해 속성에 접근

(`@ConfigurationProprtiesScan` 애너테이션을 사용하여 메시지 동적 설정 실습 진행)

```
configurations {
    compileOnly {
        extendsFrom annotationProcessor
    }
}
```

구문 설명:

gradle 코드 복사

```
configurations {
    compileOnly {
        extendsFrom annotationProcessor
    }
}
```

이 구문은 다음을 의미합니다:

1. `compileOnly`라는 구성(configuration)을 정의합니다. 이 구성은 컴파일 시에만 필요한 의존성을 지정하는 데 사용됩니다.
2. `compileOnly`가 `annotationProcessor` 구성의 의존성을 상속받도록 설정합니다. 즉, 어노테이션 프로세서에서 필요한 의존성은 자동으로 `compileOnly`에도 포함됩니다. 이는 어노테이션 프로세서가 필요로 하는 의존성이 `compileOnly` 구성에서도 사용되도록 설정하는 방법입니다.

왜 이렇게 할까요?

- 어노테이션 프로세서에서 사용되는 라이브러리가 컴파일 시에만 필요하고, 실제 실행 환경에서는 필요 없을 때, `compileOnly`와 `annotationProcessor`를 분리해서 설정할 수 있습니다.
- 이 설정은 어노테이션 처리와 컴파일 의존성을 깔끔하게 관리하면서 빌드 크기를 줄이거나 실행 시 불필요한 의존성을 제거하는 데 유리합니다.

- 스프링 부트에서는 특정 프로파일이 활성화 된 경우에만 로드하도록 설정 데이터 그룹을 정의할 수 있다.
  - 한번에 1개 or 다건의 profile 활성화 가능
  - 활성화된 profile이 없을 수도 있음
  - 어떤 프로파일에도 할당되지 않은 빈은 항상 활성화 됨

프로파일의 사용 사례

- 지정된 프로파일이 활성화된 경우에만 bean그룹을 로드하는 것

1. 프로파일을 기능 플래그로 사용

- 책의 예시에서는 해당 프로파일을 초기 데이터 적재용으로 사용하는 방법으로 실습을 진행함. (테스트 데이터 로드 기능 on/off 식)

> 💡 이때 배포 환경을 그룹화와 연관지어선 안된다.
>
> *dev, prod 프로파일을 사용해 빈을 조건부로 로드하는 경우가 있는데 이 경우 애플리케이션이 환경과 결합되어 일반적인 클라우드 네이티브 app에서는 바람직하지 않다 → 확장성이 떨어짐.*
>
> *(예시로 staging 환경이 추가되는 사례를 들었음)*
>
> **조건부로 로드할 bean 그룹과 연관된 경우 프로파일을 기능 플래그처럼 사용할 것을 추천.**
>
> **프로파일을 어디에서 활성화 할 것인지를 생각하지 말고, 해당 프로파일이 어떤 기능을 제공하는지 고려하고 프로파일의 이름도 그에 따라 정하자.**
>
> 그러나 특정 플랫폼에서 인프라 이슈를 처리하는 빈이 필요할 수도 있다. 예를들어 쿠버네티스 환경에 배포할 때만 로드해야 하는 빈이 있다면 `kubernetes`라는 프로파일을 정의할 수 있음.

- `spring.profiles.active` 속성을 사용해 테스트 데이터 프로파일을 활성화할 수 있으나, `bootRun` 작업을 실행할 때 로컬 환경을 위해 `testdata 프로파일` 을 설정하도록 함.

```
bootRun {
    systemProperty 'spring.profiles.active', 'testdata'
}
```

- `gradlew` 명령어로 실행해야만 적용됨. 인텔리제이에서 설정 적용하려면 실행/디버그 설정의 `VM options`에서 `spring.profiles.active` 시스템 속성을 추가해야함.

2. 프로파일을 설정 그룹으로 사용

- 속성 파일명 끝에 프로파일을 추가하고 그 파일에 설정 데이터를 정의하는 것.

- (e.g. `application-dev.yml`)

외부화된 구성

- 외부화된 설정을 사용하면 동일한 애플리케이션 코드의 불가변 빌드를 일관되게 사용하면서 배포 위치에 따라 설정을 다르게 할 수 있다.

  → 즉, 애플리케이션을 빌드하여 패키지를 만든 후 더이상 변경하지 않는다는 것.

- **스프링 속성 우선순위**
  1. CLI 인수
     - `--` 접두사를 사용 (—polar.greeting="...")
  2. JVM 속성
     - `-D` 접두사 사용 (-Dpolar.greeting="...")
  3. 환경 변수
     - 모든 운영체제가 지원하기 때문에 어떤 환경에서도 이식 가능
     - 스프링 부트는 완화된 바인딩을 통해 `POLAR_GREETING` → `polar.greeting` 으로 인식 가능 (POLAR_GREETING="...")
  4. 속성 파일 (application.yml)
  5. 기본 설정 값

(이후 해당 구성 우선순위에 따라 jar파일 실행을 통해 실습 진행함)

---

spring cloud config 서버로 설정 관리

- **환경변수 설정의 문제**
  1. 설정 데이터를 어디에 저장해야하나?
  2. 세부적인 액세스 제어가 불가능
  3. 설정 데이터의 수정 추적/감시는 어떻게 해야하나?
  4. 설정 데이터 변경 후 재시작이 아닌 런타임에 다시 읽을 수 있도록 할 수 없나?

5. 어플리케이션 인스턴스가 증가하면 분산 방식으로 설정 처리가 어렵다
6. 설정 암호화 지원이 불가하여 암호를 안전하게 저장 불가능

세가지 유형의 해결 방식

1. **설정 서비스**
   - spring cloud alibaba (⇒ alibaba nacos)
   - spring cloud config

     ⇒ 깃 저장소, 데이터 저장소, vault와 같이 장착 가능한 데이터 소스에 의해 지원되는 설정 서비스 제공)
   - spring cloud consul (⇒ hashiCorp Consul)
   - spring cloud vault (⇒ hashiCorp Vault)
   - spring cloud zookeeper (⇒ Apache Zookeeper)
2. **클라우드 공급업체 서비스**
   - AWS: aws 파라미터 스토어, aws 시크릿 관리자
   - Azure: 애저 키 볼트
   - GCP: GCP 비밀 관리자
3. **클라우드 플랫폼 서비스**
   - 쿠버네티스: 컨피그맵, 시크릿

설정 서비스의 여러 방식 중 어떤것을 선택할 지는 인프라와 요구사항에 따라 다르다.

중앙 집중식 설정

- **주요 구성 요소**
  1. 설정 데이터에 대한 데이터 저장소로 지속성, 버전관리, 액세스 제어 제공
  2. 데이터 저장소에 기반해 설정 데이터 관리, 여러 앱에 설정 데이터를 제공하는 서버의 역할

→ 데이터가 저장되는 방식에 관계없이 설정 서버는 통합 인터페이스를 통해 타 애플리케이션에 설정 데이터를 보내준다.

(깃을 사용하여 설정 데이터를 저장하는 실습 진행)

스프링 클라우드 컨피그가 설정 데이터를 찾는 법

- `{application}` : spring.application.name 속성에 정의된 애플리케이션의 이름
- `{profile}`: spring.profiles.active 속성에 의해 정의된 활성 프로파일 중 하나
- `{label}` : 특정 저장소에서 사용하는 식별자.
  - 깃의 경우 태그, 브랜치 이름 또는 커밋 ID가 될 수 있다.
  - 버전으로 구분된 설정 파일 세트를 식별하는데 유용

```
/{application}/application-{profile}.yml
/{application}/application.yml
/{application}-{profile}.yml
/{application}.yml
/application-{profile}.yml
/application.yml
```

(e.g. `/catalog-service/application.yml` , `/catalog-service.yml`)

1. 프로젝트 생성

- 스프링 클라우드 컨피그 서버
  - 스프링 웹을 토대로 설정 서버를 구축할 수 있는 라이브러리와 유틸리티 제공

2. 설정 서버 활성화

- `@EnableConfigServer` 로 활성화

3. 설정 서버 설정

- `spring.cloud.config.server.git.uri`
  - 설정 데이터 저장소 위치
- `spring.cloud.config.server.git.default-label`
  - 기본 브랜치

4. **높은 가용성을 위한 설정**
    - 컨피그 서비스는 시스템의 단일 장애 지점이 될 수 있어 높은 가용성이 필요함.
        - `timeout` 속성을 사용하여 접속 시간을 제한
        - `clone-on-start` 속성을 사용해 저장소 복제가 서비스 시작시 이루어질 수 있도록 설정
            - 서비스 시작단계가 약간 느려지지만 원격 저장소 통신이 실패하면 배포가 신속히 실패됨.
            - 설정 데이터 저장소의 로컬 복사본은 원격 저장소와의 통신이 일시적으로 실패하더라도 내결함성 향상됨.
        - `force-pull` 속성을 사용하여 로컬 복제본과 원격 저장소가 동일한 데이터를 사용할 수 있게 한다.
            - `basedir` 속성을 통해 복제되는 위치를 지정할 수 있다.

```
http :8888/catalog-service/default
http :8888/catalog-service/prod
```

- prod라고 prod 세팅만 보내주는 것은 아님. 우선순위라는게 있기 때문에 prod가 1순위 defalt가 2순위로 두 설정 모두 `propertySources` 라는 리스트에 같이 보내줌.
    - 대신 default로 호출시는 prod는 당연히 못봄.

해당 엔드 포인트를 직접 호출할 필요는 없지만, 어떤 방식으로 제공되는지는 알아야 한다.

스프링 클라우드 컨피그 클라이언트로 설정 서버 사용

설정 클라이언트 구축(카탈로그 서비스)

1. 컨피그 클라이언트에 대한 의존성을 추가함. (`spring-cloud-starter-config`)
2. 카탈로그 서비스에서 컨피그 클라이언트에 대한 설정을 작성
    a. yml 파일에서 uri, name 등등을 설정함.

내결함성이 높은 설정 클라이언트 구축

1. 타임아웃 설정

```
# 설정 서버에 연결될 때까지 기다리는 시간
request-connect-timeout: 5000
# 설정 서버에서 설정 데이터를 읽을 때까지 기다리는 시간
request-read-timeout: 5000
```

2. 재시도 패턴 사용

```
# 설정 서버 연결 실패를 치명적 오류로 인식
fail-fast: true
retry:
  max-attempts: 6
  # 최초 재시도 지연 시간
  initial-interval: 1000
  # 재시도 지연 최대 시간
  max-interval: 2000
  # 재시도 간격 증가율
  multiplier: 1.1
```

- 재시도 작동은 `fail-fast`가 'true' 일 경우에만 작동함.

런타임 시 설정 새로고침

- 스프링 클라우드 컨피그는 런타임에 설정 업데이트가 가능하다

  → 설정 저장소에 새 변경사항이 푸시될 때마다 설정 서버와 통합된 모든 앱에 신호를 보내고, 앱은 영향받는 부분을 재로드 한다.

- **핫 리로드**

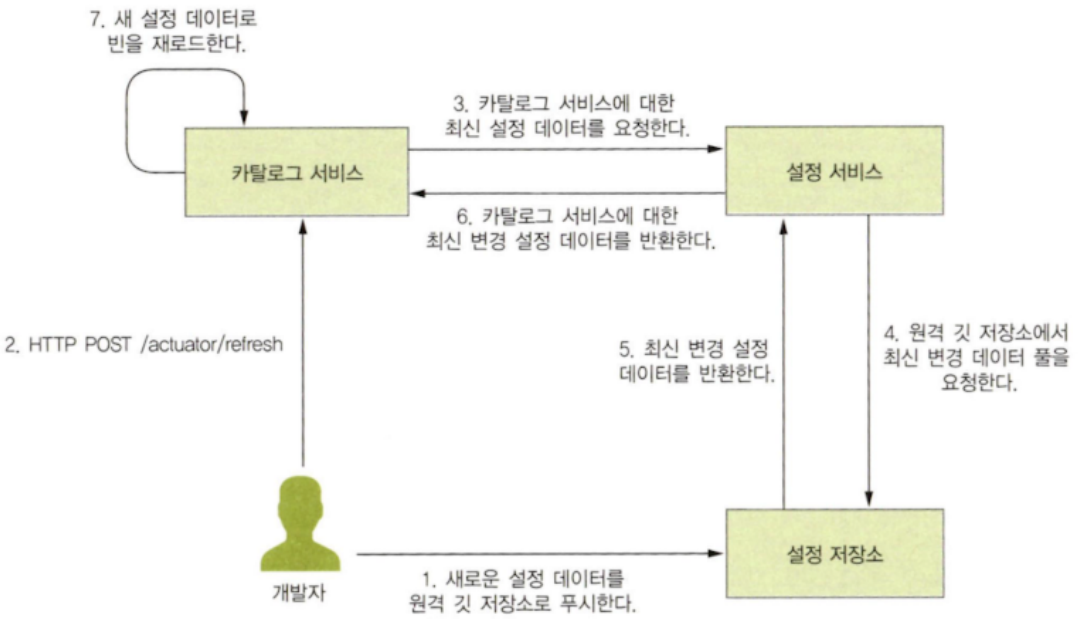

그림 4.11 설정 서비스를 지원하는 깃 저장소의 설정이 변경되면 그 변경된 설정을 사용해 애플리케이션의 일부를 새로고침하라는 신호를 카탈로그 서비스로 보낸다.

- 운영 환경에서는 명시적인 새로고침보다 웹훅을 통해 변경사항 알림을 설정하는 것이 좋다. (14장에 나옴)

1. 설정 새로고침 활성화

- 스프링 부트 액추에이터를 사용해 post 요청으로 애플리케이션 콘텍스트 내에서 `RefreshScopeRefreshedEvent`를 발생하도록 할 수 있다.
  - 액추에이터 의존성 추가
  - refresh 엔드포인트 활성화
- `RefreshScopeRefreshedEvent` 는 새로고침이 트리거 될 때마다 다시 로드할 빈에 `@RefreshScope` 를 달아줘야 한다.
  - 하지만 `@ConfigurationProperties` 덕에 기본적으로 해당 이벤트를 듣고 있어 프로퍼티 빈이 최신 설정 값을 가지고 다시 로드된다.

2. 런타임에 설정 변경

- 앱이 실행중인 상태에서 설정 저장소 메시지 업데이트 후 `http POST :9001/actuator/refresh` 호출 (RefreshScopeRefreshedEvent 트리거)

```
❯ http POST :9001/actuator/refresh
HTTP/1.1 200
Connection: keep-alive
Content-Type: application/vnd.spring-boot.actuator.v3+json
Date: Thu, 19 Sep 2024 15:45:08 GMT
Keep-Alive: timeout=15
Transfer-Encoding: chunked

[
    "config.client.version",
    "polar.greeting"
]
```