6. 외부설정과 프로필1

#2.인강/9. 스프링부트/강의#

- 6. 외부설정과 프로필1 - 프로젝트 설정
- 6. 외부설정과 프로필1 - 외부 설정이란?
- 6. 외부설정과 프로필1 - 외부 설정 - OS 환경 변수
- 6. 외부설정과 프로필1 - 외부 설정 - 자바 시스템 속성
- 6. 외부설정과 프로필1 - 외부 설정 - 커맨드 라인 인수
- 6. 외부설정과 프로필1 - 외부 설정 - 커맨드 라인 옵션 인수
- 6. 외부설정과 프로필1 - 외부 설정 - 커맨드 라인 옵션 인수와 스프링 부트
- 6. 외부설정과 프로필1 - 외부 설정 - 스프링 통합
- 6. 외부설정과 프로필1 - 설정 데이터1 - 외부 파일
- 6. 외부설정과 프로필1 - 설정 데이터2 - 내부 파일 분리
- 6. 외부설정과 프로필1 - 설정 데이터3 - 내부 파일 합체
- 6. 외부설정과 프로필1 - 우선순위 - 설정 데이터
- 6. 외부설정과 프로필1 - 우선순위 - 전체
- 6. 외부설정과 프로필1 - 정리

프로젝트 설정

프로젝트 설정 순서

1. `external-start`의 폴더 이름을 `external`로 변경하자.
2. **프로젝트 임포트**

   File ➡ Open ➡ 해당 프로젝트의 `build.gradle`을 선택하자. 그 다음에 선택창이 뜨는데, Open as Project를 선택하자.

build.gradle 확인

```
plugins {
    id 'java'
    id 'org.springframework.boot' version '3.0.2'
    id 'io.spring.dependency-management' version '1.1.0'
}

group = 'hello'
version = '0.0.1-SNAPSHOT'
sourceCompatibility = '17'
```

```
configurations {
    compileOnly {
        extendsFrom annotationProcessor
    }
}

repositories {
    mavenCentral()
}

dependencies {
    implementation 'org.springframework.boot:spring-boot-starter'
    compileOnly 'org.projectlombok:lombok'
    annotationProcessor 'org.projectlombok:lombok'
    testImplementation 'org.springframework.boot:spring-boot-starter-test'

    //test lombok 사용
    testCompileOnly 'org.projectlombok:lombok'
    testAnnotationProcessor 'org.projectlombok:lombok'
}

tasks.named('test') {
    useJUnitPlatform()
}
```

- 스프링 부트에서 다음 라이브러리를 선택했다.
- `Lombok`
- 테스트 코드에서 lombok을 사용할 수 있도록 설정을 추가했다.

동작 확인

- 기본 메인 클래스 실행(`ExternalApplication.main()`)
- 해당 메인 메서드가 실행되고, 정상 종료되면 성공이다.

외부 설정이란?

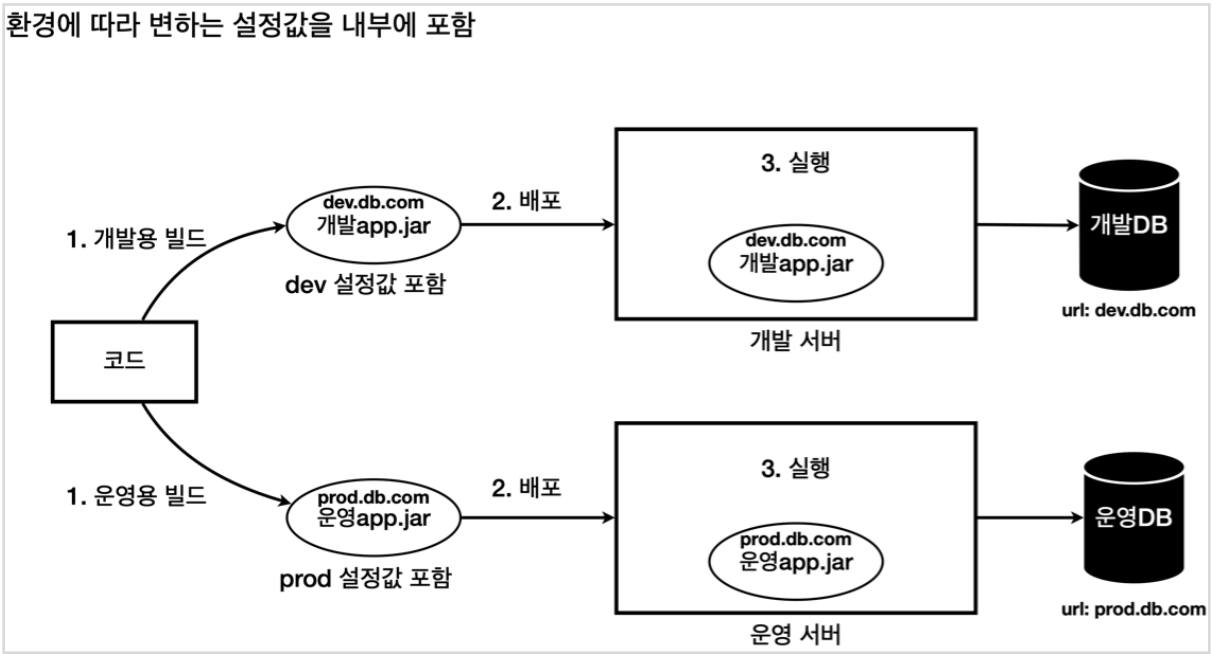
하나의 애플리케이션을 여러 다른 환경에서 사용해야 할 때가 있다. 대표적으로 개발이 잘 진행되고 있는지 내부에서 확인하는 용도의 개발 환경, 그리고 실제 고객에게 서비스하는 운영 환경이 있다.

- 개발 환경: 개발 서버, 개발 DB 사용
- 운영 환경: 운영 서버, 운영 DB 사용

문제는 각각의 환경에 따라서 서로 다른 설정값이 존재한다는 점이다. 예를 들어서 애플리케이션이 개발DB에 접근하려면 `dev.db.com` 이라는 url 정보가 필요한데, 운영DB에 접근하려면 `prod.db.com` 이라는 서로 다른 url을 사용해야 한다.

이 문제를 해결하는 가장 단순한 방법은 다음과 같이 각각의 환경에 맞게 애플리케이션을 빌드하는 것이다.

환경에 따라 변하는 설정값을 내부에 포함



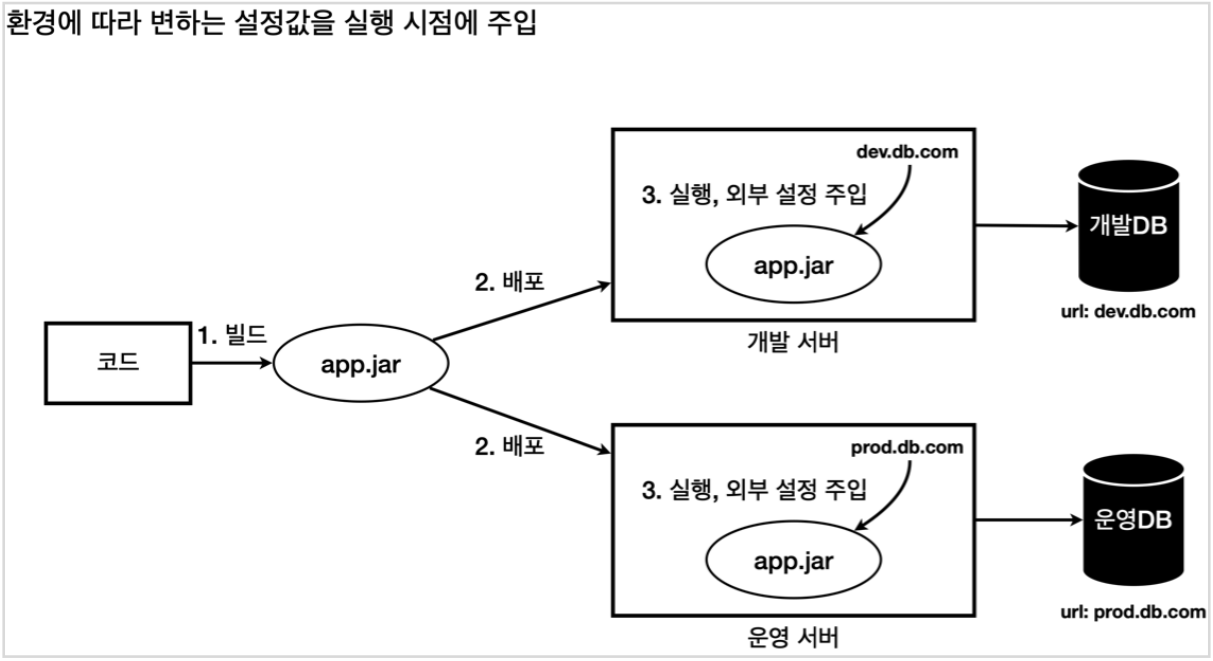
- 개발 환경에는 `dev.db.com` 이 필요하므로 이 값을 애플리케이션 코드에 넣은 다음에 빌드해서 `개발app.jar` 를 만든다.
- 운영 환경에는 `prod.db.com` 이 필요하므로 이 값을 애플리케이션 코드에 넣은 다음에 빌드해서 `운영app.jar` 를 만든다.

이렇게 하면 각각의 환경에 맞는 `개발app.jar` , `운영app.jar` 가 만들어지므로 해당 파일들을 각 환경별로 배포하면 된다.

하지만 이것은 다음과 이유로 좋은 방법이 아니다.

- 환경에 따라서 빌드를 여러번 해야 한다.
- 개발 버전과 운영 버전의 빌드 결과물이 다르다. 따라서 개발 환경에서 검증이 되더라도 운영 환경에서 다른 빌드 결과를 사용하기 때문에 예상치 못한 문제가 발생할 수 있다. 개발용 빌드가 끝나고 검증한 다음에 운영용 빌드를 해야 하는데 그 사이에 누군가 다른 코드를 변경할 수도 있다. 한마디로 진짜 같은 소스코드에서 나온 결과물인지 검증하기가 어렵다.
- 각 환경에 맞추어 최종 빌드가 되어 나온 빌드 결과물은 다른 환경에서 사용할 수 없어서 유연성이 떨어진다. 향후 다른 환경이 필요하면 그곳에 맞도록 또 빌드를 해야 한다.

그래서 보통 다음과 같이 빌드는 한번만 하고 각 환경에 맞추어 **실행 시점에 외부 설정값을 주입**한다.



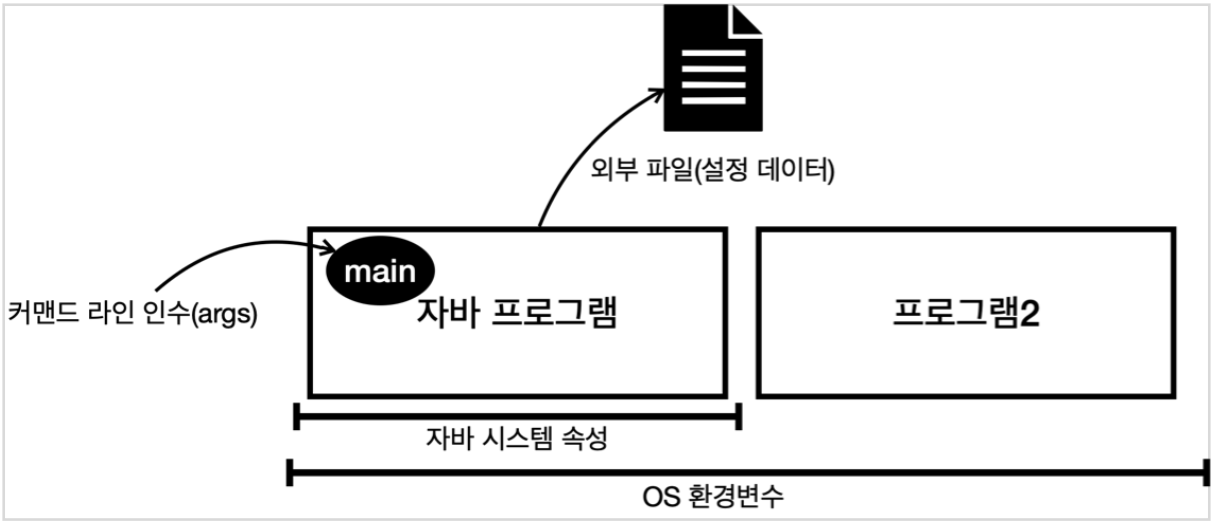
- 배포 환경과 무관하게 하나의 빌드 결과물을 만든다. 여기서는 `app.jar`를 빌드한다. 이 안에는 설정값을 두지 않는다.
- 설정값은 실행 시점에 각 환경에 따라 외부에서 주입한다.
  - 개발 서버: `app.jar`를 실행할 때 `dev.db.com` 값을 외부 설정으로 주입한다.
  - 운영 서버: `app.jar`를 실행할 때 `prod.db.com` 값을 외부 설정으로 주입한다.

이렇게 하면 빌드도 한번만 하면 되고, 개발 버전과 운영 버전의 빌드 결과물이 같기 때문에 개발환경에서 검증되면 운영 환경에서도 믿고 사용할 수 있다. 그리고 이후에 새로운 환경이 추가되어도 별도의 빌드 과정 없이 기존 `app.jar`를 사용해서 손쉽게 새로운 환경을 추가할 수 있다.

유지보수하기 좋은 애플리케이션 개발의 가장 기본 원칙은 변하는 것과 변하지 않는 것을 분리하는 것이다.
유지보수하기 좋은 애플리케이션을 개발하는 단순하면서도 중요한 원칙은 **변하는 것과 변하지 않는 것을 분리**하는 것이다. 각 환경에 따라 변하는 외부 설정값은 분리하고, 변하지 않는 코드와 빌드 결과물은 유지했다. 덕분에 빌드 과정을 줄이고, 환경에 따른 유연성을 확보하게 되었다.

외부 설정

애플리케이션을 실행할 때 필요한 설정값을 외부에서 어떻게 불러와서 애플리케이션에 전달할 수 있을까?



외부 설정은 일반적으로 다음 4가지 방법이 있다.

- OS 환경 변수: OS에서 지원하는 외부 설정, 해당 OS를 사용하는 모든 프로세스에서 사용
- 자바 시스템 속성: 자바에서 지원하는 외부 설정, 해당 JVM안에서 사용
- 자바 커맨드 라인 인수: 커맨드 라인에서 전달하는 외부 설정, 실행시 `main(args)` 메서드에서 사용
- 외부 파일(설정 데이터): 프로그램에서 외부 파일을 직접 읽어서 사용
  - 애플리케이션에서 특정 위치의 파일을 읽도록 해둔다. 예) `data/hello.txt`
  - 그리고 각 서버마다 해당 파일안에 다른 설정 정보를 남겨둔다.
    - 개발 서버 `hello.txt` : `url=dev.db.com`
    - 운영 서버 `hello.txt` : `url=prod.db.com`

먼저 앞의 3가지를 알아보자. 외부 파일(설정 데이터)은 뒤에서 설명한다.

외부 설정 - OS 환경 변수

OS 환경 변수(OS environment variables)는 해당 OS를 사용하는 모든 프로그램에서 읽을 수 있는 설정값이다. 한마디로 다른 외부 설정과 비교해서 사용 범위가 가장 넓다.

조회 방법
윈도우 OS: `set`
MAC, 리눅스 OS: `printenv`

printenv 실행 결과

```
printenv
ITERM_PROFILE=Default
XPC_FLAGS=0x0
```

```
LANG=ko_KR.UTF-8
PWD=/Users/kimyounghan
SHELL=/bin/zsh
PATH=/usr/local/bin:/usr/bin:/bin:/usr/sbin:/sbin
COMMAND_MODE=unix2003
TERM=xterm-256color
HOME=/Users/kimyounghan
TMPDIR=/var/folders/zs/...
USER=kimyounghan
XPC_SERVICE_NAME=0
LOGNAME=kimyounghan
```

현재 OS에 설정된 OS 환경 변수 값들을 출력했다.

설정 방법

OS 환경 변수의 값을 설정하는 방법은 `윈도우 환경 변수`, `mac 환경 변수` 등으로 검색해보자. 수 많은 예시를 확인할 수 있다.

애플리케이션에서 OS 환경 변수의 값을 읽어보자.

OsEnv - src/test 하위

```
package hello.external;

import lombok.extern.slf4j.Slf4j;

import java.util.Map;

@Slf4j
public class OsEnv {

    public static void main(String[] args) {
        Map<String, String> envMap = System.getenv();
        for (String key : envMap.keySet()) {
            log.info("env {}={}", key, System.getenv(key));
        }
    }
}
```

- `System.getenv()`를 사용하면 전체 OS 환경 변수를 `Map`으로 조회할 수 있다.
- `System.getenv(key)`를 사용하면 특정 OS 환경 변수의 값을 `String`으로 조회할 수 있다.

실행 결과

```
env PATH=/usr/local/bin:/usr/bin:/bin:/usr/sbin:/sbin
env SHELL=/bin/zsh
env USER=kimyounghan
env TMPDIR=/var/folders/zs/...
env COMMAND_MODE=unix2003
env LOGNAME=kimyounghan
env LC_CTYPE=ko_KR.UTF-8
env HOME=/Users/kimyounghan
```

OS 환경 변수를 설정하고, 필요한 곳에서 `System.getenv()`를 사용하면 외부 설정을 사용할 수 있다.
이제 데이터베이스 접근 URL과 같은 정보를 OS 환경 변수에 설정해두고 읽어들이면 된다.
예를 들어서 개발 서버에서는 `DBURL=dev.db.com`과 같이 설정하고, 운영 서버에서는 `DBURL=prod.db.com`와 같이 설정하는 것이다.
이렇게 하면 `System.getenv("DBURL")`을 조회할 때 각각 환경에 따라서 서로 다른 값을 읽게 된다.

하지만 OS 환경 변수는 이 프로그램 뿐만 아니라 다른 프로그램에서도 사용할 수 있다. 쉽게 이야기해서 전역 변수 같은 효과가 있다. 여러 프로그램에서 사용하는 것이 맞을 때도 있지만, 해당 애플리케이션을 사용하는 자바 프로그램 안에서만 사용되는 외부 설정값을 사용하고 싶을 때도 있다. 다음에는 특정 자바 프로그램안에서 사용하는 외부 설정을 알아보자.

외부 설정 - 자바 시스템 속성

자바 시스템 속성(Java System properties)은 실행한 JVM 안에서 접근 가능한 외부 설정이다. 추가로 자바가 내부에서 미리 설정해두고 사용하는 속성들도 있다.

자바 시스템 속성은 다음과 같이 자바 프로그램을 실행할 때 사용한다.

- 예) `java -Durl=dev -jar app.jar`
- `-D` VM 옵션을 통해서 `key=value` 형식을 주면 된다. 이 예제는 `url=dev` 속성이 추가된다.
- 순서에 주의해야 한다. `-D` 옵션이 `jar` 보다 앞에 있다.

JavaSystemProperties - src/test 하위

```
package hello.external;

import lombok.extern.slf4j.Slf4j;

import java.util.Properties;

@Slf4j
public class JavaSystemProperties {

    public static void main(String[] args) {
        Properties properties = System.getProperties();
        for (Object key : properties.keySet()) {
            log.info("prop {}={}", key,
System.getProperty(String.valueOf(key)));
        }

    }
}
```

- `System.getProperties()`를 사용하면 `Map`과 유사한(`Map`의 자식 타입) `key=value` 형식의 `Properties`를 받을 수 있다. 이것을 통해서 모든 자바 시스템 속성을 조회할 수 있다.
- `System.getProperty(key)`를 사용하면 속성값을 조회할 수 있다.

실행 결과

```
#JAVA 기본 설정 속성
prop java.specification.version=17
prop java.class.version=61.0
prop file.encoding=UTF-8
prop os.name=Mac OS X
prop sun.java.command=hello.external.JavaSystemProperties
prop user.name=kimyounghan
```

- 자바가 기본으로 제공하는 수 많은 속성들이 추가되어 있는 것을 확인할 수 있다. 자바는 내부에서 필요할 때 이런 속성들을 사용하는데, 예를 들어서 `file.encoding=UTF-8`를 통해서 기본적인 파일 인코딩 정보 등으로 사용한다.

이번에는 사용자가 직접 정의하는 자바 시스템 속성을 추가해보자.

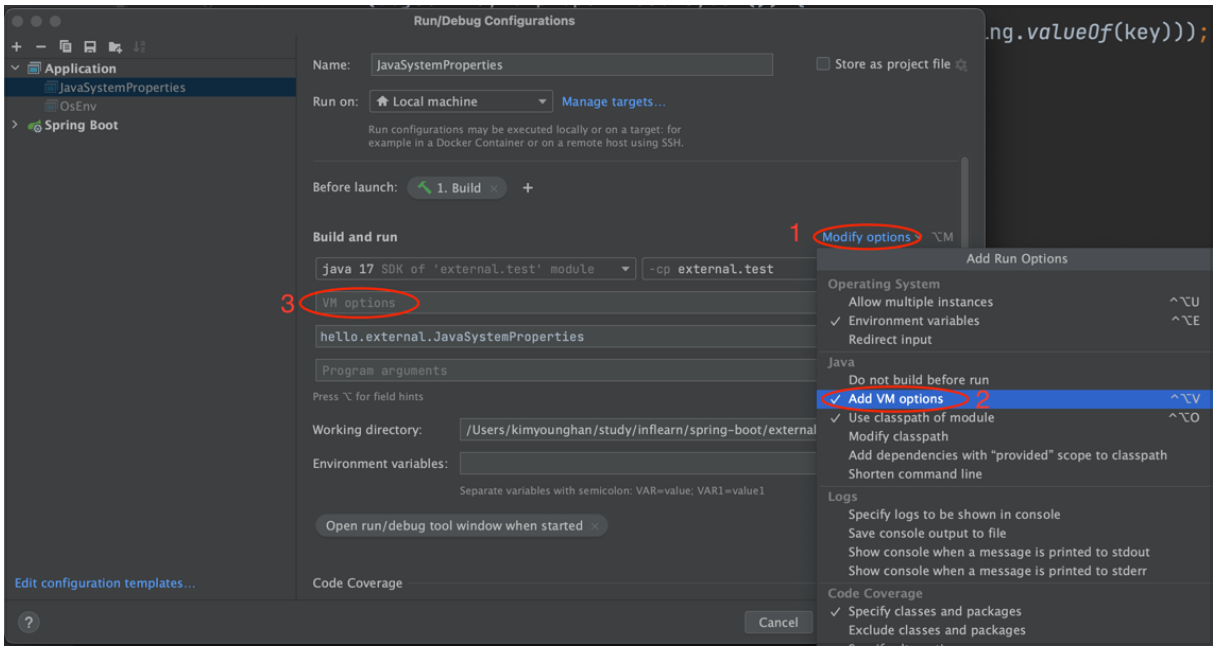
`url`, `username`, `password`를 조회하는 코드를 추가하자

JavaSystemProperties - 추가

```
package hello.external;

import lombok.extern.slf4j.Slf4j;

import java.util.Properties;

@Slf4j
public class JavaSystemProperties {

    public static void main(String[] args) {
        Properties properties = System.getProperties();
        for (Object key : properties.keySet()) {
            log.info("prop {}={}", key,
System.getProperty(String.valueOf(key)));
        }

        String url = System.getProperty("url");
        String username = System.getProperty("username");
        String password = System.getProperty("password");

        log.info("url={}", url);
        log.info("username={}", username);
        log.info("password={}", password);
    }
}
```

- 실행할 때 자바 시스템 속성을 추가해야 한다.

IDE에서 실행시 VM 옵션 추가



- 1. Modify options를 선택한다.
- 2. Add VM options를 선택한다.
- 3. VM options에 다음을 추가한다.
  - `-Durl=devdb -Dusername=dev_user -Dpassword=dev_pw`

실행 결과

```
#추가한 자바 시스템 속성
url=devdb
username=dev_user
password=dev_pw
```

- 실행해보면 `-D` 옵션을 통해 추가한 자바 시스템 속성들을 확인할 수 있다.

Jar 실행

`jar`로 빌드되어 있다면 실행시 다음과 같이 자바 시스템 속성을 추가할 수 있다.

```
java -Durl=devdb -Dusername=dev_user -Dpassword=dev_pw -jar app.jar
```

자바 시스템 속성을 자바 코드로 설정하기

자바 시스템 속성은 앞서 본 것 처럼 `-D` 옵션을 통해 실행 시점에 전달하는 것도 가능하고, 다음과 같이 자바 코드 내부에서 추가하는 것도 가능하다. 코드에서 추가하면 이후에 조회시에 값을 조회할 수 있다.

- 설정: `System.setProperty(propertyName, "propertyValue")`
- 조회: `System.getProperty(propertyName)`

참고로 이 방식은 코드 안에서 사용하는 것이기 때문에 외부로 설정을 분리하는 효과는 없다.

외부 설정 - 커맨드 라인 인수

커맨드 라인 인수(Command line arguments)는 애플리케이션 실행 시점에 외부 설정값을 `main(args)` 메서드의 `args` 파라미터로 전달하는 방법이다.

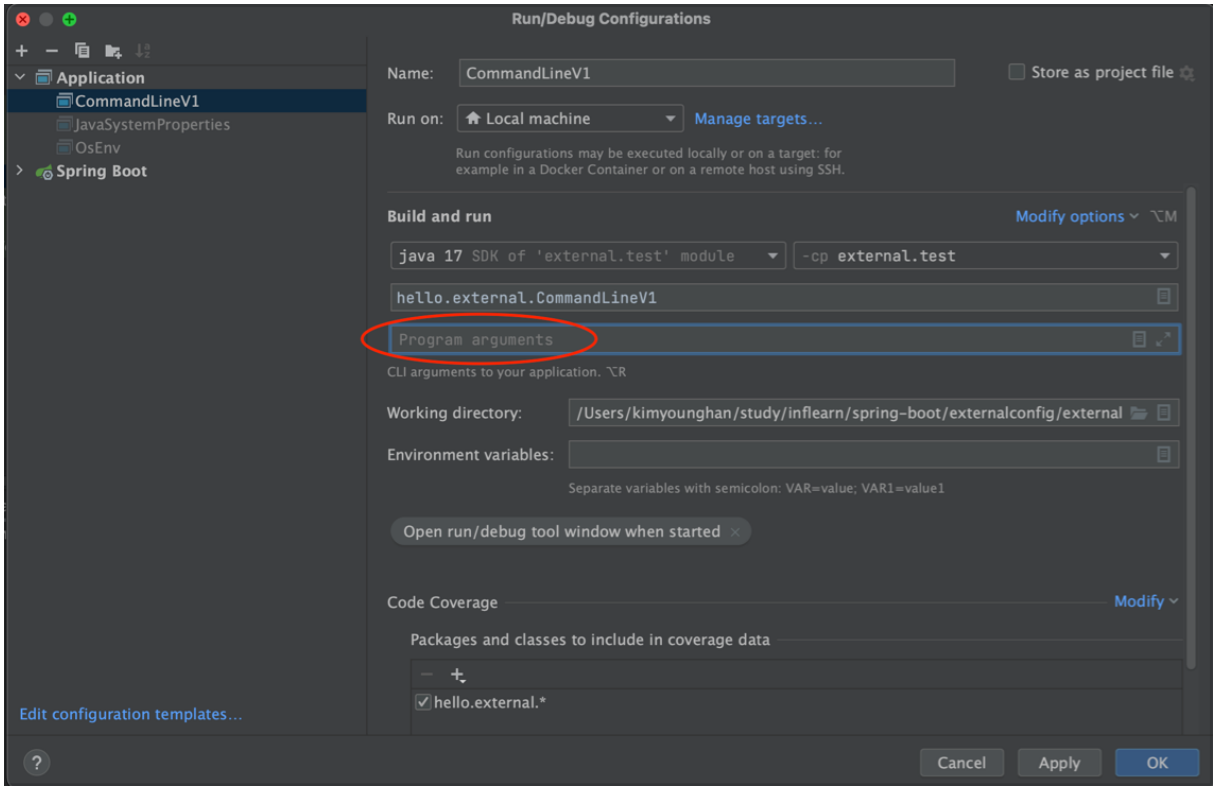
다음과 같이 사용한다.

- 예) `java -jar app.jar dataA dataB`
- 필요한 데이터를 마지막 위치에 스페이스로 구분해서 전달하면 된다. 이 경우 `dataA` , `dataB` 2개의 문자가 `args` 에 전달된다.

CommandLineV1 - src/test 하위

```
package hello.external;

import lombok.extern.slf4j.Slf4j;

/**
 * CommandLine 인수는 스페이스로 구분
 * java -jar app.jar dataA dataB -> [dataA, dataB] 2개
 * java -jar app.jar url=devdb -> [url=devdb] 1개
 * url=devdb 이라는 단어를 개발자가 직접 파싱해야 함
 **/
@Slf4j
public class CommandLineV1 {

    public static void main(String[] args) {
        for (String arg : args) {
            log.info("arg {}", arg);
        }
    }
}
```

IDE에서 실행시 커맨드 라인 인수 추가



- 여기에 `dataA dataB` 를 입력하고 실행하자.
- 커맨드 라인 인수는 공백(space)으로 구분한다.

참고

빨간색으로 칠한 `Program arguments` 가 보이지 않는다면 바로 위에 있는 파란색의 `Modify options` 버튼을 눌러서 추가할 수 있다.

실행 결과

```
CommandLineV1 - arg dataA
CommandLineV1 - arg dataB
```

Jar 실행

`jar` 로 빌드되어 있다면 실행시 다음과 같이 커맨드 라인 인수를 추가할 수 있다.

`java -jar project.jar dataA dataB`

key=value 형식 입력

애플리케이션을 개발할 때는 보통 `key=value` 형식으로 데이터를 받는 것이 편리하다.

이번에는 커맨드 라인 인수를 다음과 같이 입력하고 실행해보자

`url=devdb username=dev_user password=dev_pw`

실행 결과

```
CommandLineV1 - arg url=devdb
CommandLineV1 - arg username=dev_user
CommandLineV1 - arg password=dev_pw
```

실행 결과를 보면 알겠지만 커맨드 라인 인수는 `key=value` 형식이 아니다. 단순히 문자를 여러게 입력 받는 형식인 것이다. 그래서 3가지 문자가 입력되었다.

- `url=devdb`
- `username=dev_user`
- `password=dev_pw`

이것은 파싱되지 않은, 통 문자이다.
이 경우 개발자가 `=`을 기준으로 직접 데이터를 파싱해서 `key=value` 형식에 맞도록 분리해야 한다.
그리고 형식이 배열이기 때문에 루프를 돌면서 원하는 데이터를 찾아야 하는 번거로움도 발생한다.
실제 애플리케이션을 개발할 때는 주로 `key=value` 형식을 자주 사용하기 때문에 결국 파싱해서 `Map` 같은 형식으로 변환하도록 직접 개발해야 하는 번거로움이 있다.

외부 설정 - 커맨드 라인 옵션 인수

일반적인 커맨드 라인 인수
커맨드 라인에 전달하는 값은 형식이 없고, 단순히 띄어쓰기로 구분한다.

- `aaa bbb` ➡ `[aaa, bbb]` 값 2개
- `hello world` ➡ `[hello, world]` 값 2개
- `"hello world"` ➡ `[hello world]` (공백을 연결하려면 `"` 를 사용하면 된다.) 값 1개
- `key=value` ➡ `[key=value]` 값 1개

커맨드 라인 옵션 인수(command line option arguments)
커맨드 라인 인수를 `key=value` 형식으로 구분하는 방법이 필요하다. 그래서 스프링에서는 커맨드 라인 인수를 `key=value` 형식으로 편리하게 사용할 수 있도록 스프링 만의 표준 방식을 정의했는데, 그것이 바로 커맨드 라인 옵션 인수이다.

스프링은 커맨드 라인에 `-`(dash) 2개(`--`)를 연결해서 시작하면 `key=value` 형식으로 정하고 이것을 커맨드 라인 옵션 인수라 한다.

- `--key=value` 형식으로 사용한다.

- `--username=userA --username=userB` 하나의 키에 여러 값도 지정할 수 있다.

CommandLineV2 - src/test 하위

```
package hello.external;

import lombok.extern.slf4j.Slf4j;
import org.springframework.boot.ApplicationArguments;
import org.springframework.boot.DefaultApplicationArguments;

import java.util.List;
import java.util.Set;

@Slf4j
public class CommandLineV2 {

    public static void main(String[] args) {
        for (String arg : args) {
            log.info("arg {}", arg);
        }
        ApplicationArguments appArgs = new DefaultApplicationArguments(args);
        log.info("SourceArgs = {}", List.of(appArgs.getSourceArgs()));
        log.info("NonOptionArgs = {}", appArgs.getNonOptionArgs());
        log.info("OptionNames = {}", appArgs.getOptionNames());

        Set<String> optionNames = appArgs.getOptionNames();
        for (String optionName : optionNames) {
            log.info("option args {}={}", optionName,
appArgs.getOptionValues(optionName));
        }

        List<String> url = appArgs.getOptionValues("url");
        List<String> username = appArgs.getOptionValues("username");
        List<String> password = appArgs.getOptionValues("password");
        List<String> mode = appArgs.getOptionValues("mode");
        log.info("url={}", url);
        log.info("username={}", username);
        log.info("password={}", password);
        log.info("mode={}", mode);
```

```
    }
}
```

스프링이 제공하는 `ApplicationArguments` 인터페이스와 `DefaultApplicationArguments` 구현체를 사용하면 커맨드 라인 옵션 인수를 규격대로 파싱해서 편리하게 사용할 수 있다.

실행

커맨드 라인 인수를 다음과 같이 입력하고 실행해보자

`--url=devdb --username=dev_user --password=dev_pw mode=on`

이해를 돕기 위해 `--`(dash)가 없는 `mode=on`이라는 옵션도 마지막에 추가했다.

여기서 커맨드 라인 옵션 인수와, 옵션 인수가 아닌 것을 구분할 수 있다.

옵션 인수

`--`로 시작한다.

- `--url=devdb`
- `--username=dev_user`
- `--password=dev_pw`

옵션 인수가 아님

`--`로 시작하지 않는다.

- `mode=on`

실행 결과

```
arg --url=devdb
arg --username=dev_user
arg --password=dev_pw
arg mode=on

SourceArgs = [--url=devdb, --username=dev_user, --password=dev_pw, mode=on]
NonOptionArgs = [mode=on]

OptionNames = [password, url, username]
option args password=[dev_pw]
option args url=[devdb]
option args username=[dev_user]

url=[devdb]
username=[dev_user]
```

```
password=[dev_pw]
mode=null
```

실행 결과를 분석해보자

- `arg` : 커맨드 라인의 입력 결과를 그대로 출력한다.
- `SourceArgs` : 커맨드 라인 인수 전부를 출력한다.
- `NonOptionArgs = [mode=on]` : 옵션 인수가 아니다. `key=value` 형식으로 파싱되지 않는다. `--` 를 앞에 사용하지 않았다.
- `OptionNames = [password, url, username]` : `key=value` 형식으로 사용되는 옵션 인수다. `--` 를 앞에 사용했다.
- `url` , `username` , `password` 는 옵션 인수이므로 `appArgs.getOptionValues(key)` 로 조회할 수 있다.
- `mode` 는 옵션 인수가 아니므로 `appArgs.getOptionValues(key)` 로 조회할 수 없다. 따라서 결과는 `null` 이다.

참고

- 참고로 옵션 인수는 `--username=userA --username=userB` 처럼 하나의 키에 여러 값을 포함할 수 있기 때문에 `appArgs.getOptionValues(key)` 의 결과는 리스트(`List`)를 반환한다.
- 커맨드 라인 옵션 인수는 자바 언어의 표준 기능이 아니다. 스프링이 편리함을 위해 제공하는 기능이다.

외부 설정 - 커맨드 라인 옵션 인수와 스프링 부트

스프링 부트는 커맨드 라인을 포함해서 커맨드 라인 옵션 인수를 활용할 수 있는 `ApplicationArguments` 를 스프링 빈으로 등록해둔다. 그리고 그 안에 입력한 커맨드 라인을 저장해둔다. 그래서 해당 빈을 주입 받으면 커맨드 라인으로 입력한 값을 어디서든 사용할 수 있다.

CommandLineBean - src/main 하위

```
package hello;

import jakarta.annotation.PostConstruct;
import lombok.extern.slf4j.Slf4j;
import org.springframework.boot.ApplicationArguments;
import org.springframework.stereotype.Component;

import java.util.List;
```

```
import java.util.Set;

@Slf4j
@Component
public class CommandLineBean {

    private final ApplicationArguments arguments;

    public CommandLineBean(ApplicationArguments arguments) {
        this.arguments = arguments;
    }

    @PostConstruct
    public void init() {
        log.info("source {}", List.of(arguments.getSourceArgs()));
        log.info("optionNames {}", arguments.getOptionNames());
        Set<String> optionNames = arguments.getOptionNames();
        for (String optionName : optionNames) {
            log.info("option args {}={}", optionName,
arguments.getOptionValues(optionName));
        }
    }
}
```

주의! `src/main` 하위에 위치한다.

실행

커맨드 라인 인수를 다음과 같이 입력하고 실행해보자

`--url=devdb --username=dev_user --password=dev_pw mode=on`

다음을 실행한다. `ExternalApplication.main()`

실행 결과

```
CommandLineBean: source [--url=devdb, --username=dev_user, --password=dev_pw,
mode=on]
CommandLineBean: optionNames [password, url, username]
CommandLineBean: option args password=[dev_pw]
CommandLineBean: option args url=[devdb]
CommandLineBean: option args username=[dev_user]
```

실행 결과를 보면, 입력한 커맨드 라인 인수, 커맨드 라인 옵션 인수를 확인할 수 있다.

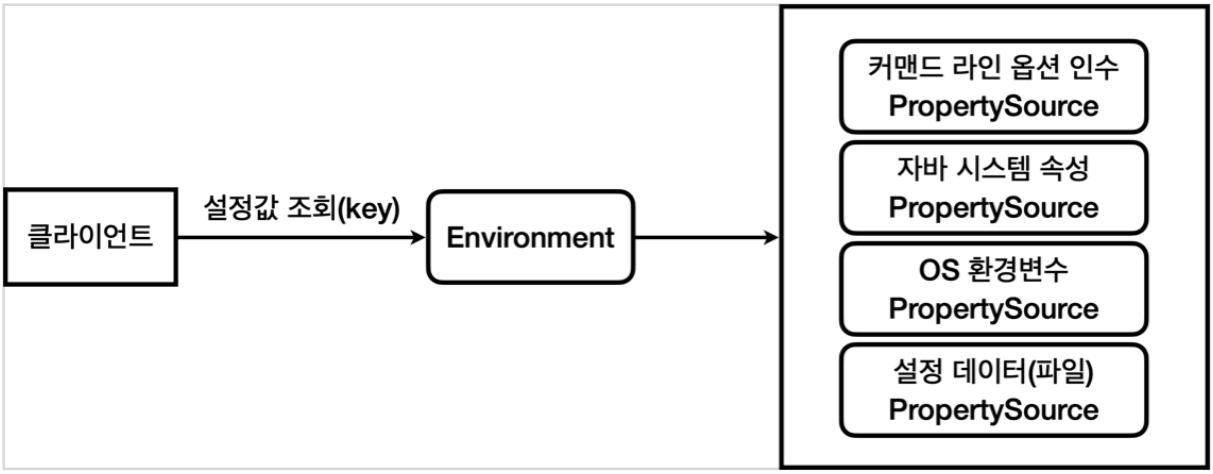
외부 설정 - 스프링 통합

지금까지 살펴본, 커맨드 라인 옵션 인수, 자바 시스템 속성, OS 환경변수는 모두 외부 설정을 `key=value` 형식으로 사용할 수 있는 방법이다. 그런데 이 외부 설정값을 읽어서 사용하는 개발자 입장에서 단순하게 생각해보면, 모두 `key=value` 형식이고, 설정값을 외부로 뽑아둔 것이다. 그런데 어디에 있는 외부 설정값을 읽어야 하는지에 따라서 각각 읽는 방법이 다르다는 단점이 있다.
예를 들어서 OS 환경 변수에 두면 `System.getenv(key)` 를 사용해야 하고, 자바 시스템 속성을 사용하면 `System.getProperty(key)` 를 사용해야 한다. 만약 OS에 환경 변수를 두었는데, 이후에 정책이 변경되어서 자바 시스템 속성에 환경 변수를 두기로 했다고 가정해보자. 그러면 해당 코드들을 모두 변경해야 한다.

외부 설정값이 어디에 위치하든 상관없이 일관성 있고, 편리하게 `key=value` 형식의 외부 설정값을 읽을 수 있으면 사용하는 개발자 입장에서 더 편리하고 또 외부 설정값을 설정하는 방법도 더 유연해질 수 있다. 예를 들어서 외부 설정값을 OS 환경변수를 사용하다가 자바 시스템 속성으로 변경하는 경우에 소스코드를 다시 빌드하지 않고 그대로 사용할 수 있다.

스프링은 이 문제를 `Environment` 와 `PropertySource` 라는 추상화를 통해서 해결한다.

스프링의 외부 설정 통합



PropertySource

- `org.springframework.core.env.PropertySource`
- 스프링은 `PropertySource` 라는 추상 클래스를 제공하고, 각각의 외부 설정를 조회하는 `XxxPropertySource` 구현체를 만들어두었다.
  - 예)

    - `CommandLinePropertySource`
    - `SystemEnvironmentPropertySource`
- 스프링은 로딩 시점에 필요한 `PropertySource` 들을 생성하고, `Environment` 에서 사용할 수 있게 연결해둔다.

Environment

- `org.springframework.core.env.Environment`
- `Environment` 를 통해서 특정 외부 설정에 종속되지 않고, 일관성 있게 `key=value` 형식의 외부 설정에 접근할 수 있다.
    - `environment.getProperty(key)` 를 통해서 값을 조회할 수 있다.
    - `Environment` 는 내부에서 여러 과정을 거쳐서 `PropertySource` 들에 접근한다.
    - 같은 값이 있을 경우를 대비해서 스프링은 미리 우선순위를 정해두었다. (뒤에서 설명한다.)
- 모든 외부 설정은 이제 `Environment` 를 통해서 조회하면 된다.

설정 데이터(파일)

여기에 우리가 잘 아는 `application.properties` , `application.yml` 도 `PropertySource` 에 추가된다. 따라서 `Environment` 를 통해서 접근할 수 있다.

EnvironmentCheck, src/main 하위

```
package hello;

import jakarta.annotation.PostConstruct;
import lombok.extern.slf4j.Slf4j;
import org.springframework.core.env.Environment;
import org.springframework.stereotype.Component;

@Slf4j
@Component
public class EnvironmentCheck {

    private final Environment env;

    public EnvironmentCheck(Environment env) {
        this.env = env;
    }

    @PostConstruct
    public void init() {
```

```
        String url = env.getProperty("url");
        String username = env.getProperty("username");
        String password = env.getProperty("password");
        log.info("env url={}", url);
        log.info("env username={}", username);
        log.info("env password={}", password);
    }
}
```

- 커맨드 라인 옵션 인수 실행
  - `--url=devdb --username=dev_user --password=dev_pw`
- 자바 시스템 속성 실행
  - `-Durl=devdb -Dusername=dev_user -Dpassword=dev_pw`

`ExternalApplication.main`을 실행하자.

실행 결과

```
env url=devdb
env username=dev_user
env password=dev_pw
```

정리

커맨드 라인 옵션 인수, 자바 시스템 속성 모두 `Environment`를 통해서 동일한 방법으로 읽을 수 있는 것을 확인했다.

스프링은 `Environment`를 통해서 외부 설정을 읽는 방법을 추상화했다. 덕분에 자바 시스템 속성을 사용하다가 만약 커맨드 라인 옵션 인수를 사용하도록 읽는 방법이 변경되어도, 개발 소스 코드는 전혀 변경하지 않아도 된다.

우선순위

예를 들어서 커맨드 라인 옵션 인수와 자바 시스템 속성을 다음과 같이 중복해서 설정하면 어떻게 될까?

- 커맨드 라인 옵션 인수 실행
  - `--url=proddb --username=prod_user --password=prod_pw`
- 자바 시스템 속성 실행
  - `-Durl=devdb -Dusername=dev_user -Dpassword=dev_pw`

우선순위는 상식 선에서 딱 2가지만 기억하면 된다.

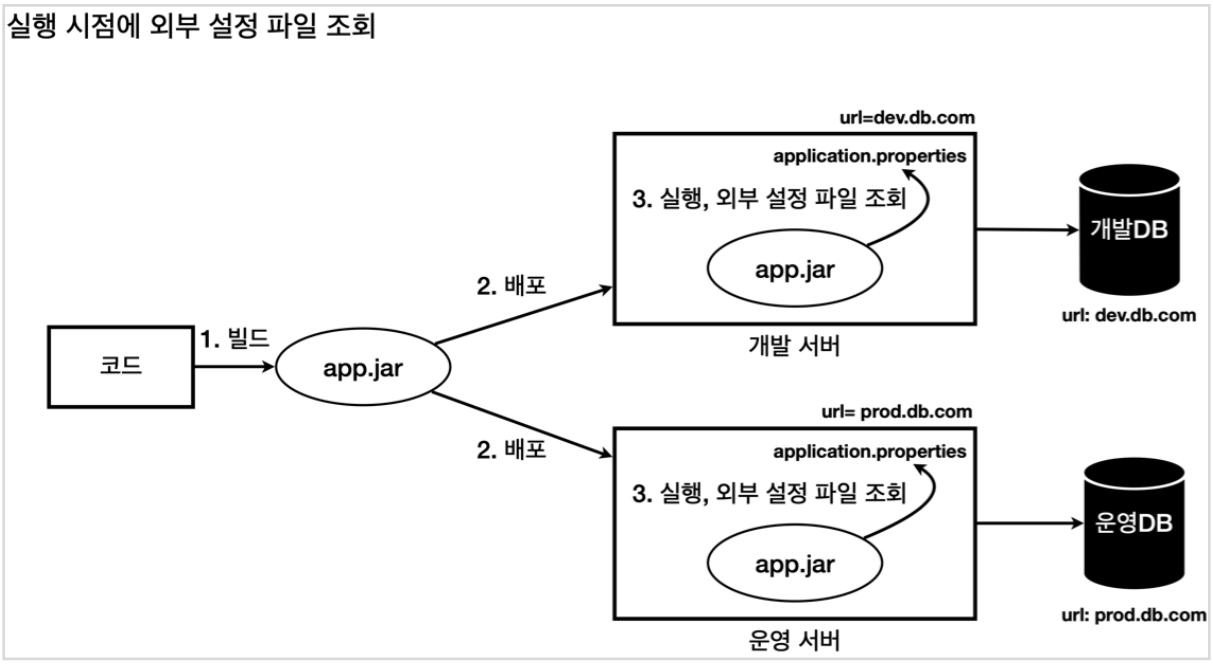
- 더 유연한 것이 우선권을 가진다. (변경하기 어려운 파일 보다 실행시 원하는 값을 줄 수 있는 자바 시스템 속성이 더 우선권을 가진다.)
- 범위가 넓은 것 보다 좁은 것이 우선권을 가진다. (자바 시스템 속성은 해당 JVM 안에서 모두 접근할 수 있다. 반면에 커맨드 라인 옵션 인수는 `main`의 arg를 통해서 들어오기 때문에 접근 범위가 더 좁다.)

자바 시스템 속성과 커맨드 라인 옵션 인수의 경우 커맨드 라인 옵션 인수의 범위가 더 좁기 때문에 커맨드 라인 옵션 인수가 우선권을 가진다.
우선순위는 뒤에서 더 자세히 다루겠다.

설정 데이터1 - 외부 파일

지금까지 학습한 OS 환경 변수, 자바 시스템 속성, 커맨드 라인 옵션 인수는 사용해야 하는 값이 늘어날 수록 사용하기가 불편해진다. 실무에서는 수십개의 설정값을 사용하기도 하므로 이런 값들을 프로그램을 실행할 때 마다 입력하게 되면 번거롭고, 관리도 어렵다.

그래서 등장하는 대안으로는 설정값을 파일에 넣어서 관리하는 방법이다. 그리고 애플리케이션 로딩 시점에 해당 파일을 읽어들이면 된다. 그 중에서도 `.properties`라는 파일은 `key=value` 형식을 사용해서 설정값을 관리하기에 아주 적합하다.



`application.properties` **개발 서버**에 있는 외부 파일

```
url=dev.db.com
username=dev_user
password=dev_pw
```

`application.properties` **운영 서버**에 있는 외부 파일

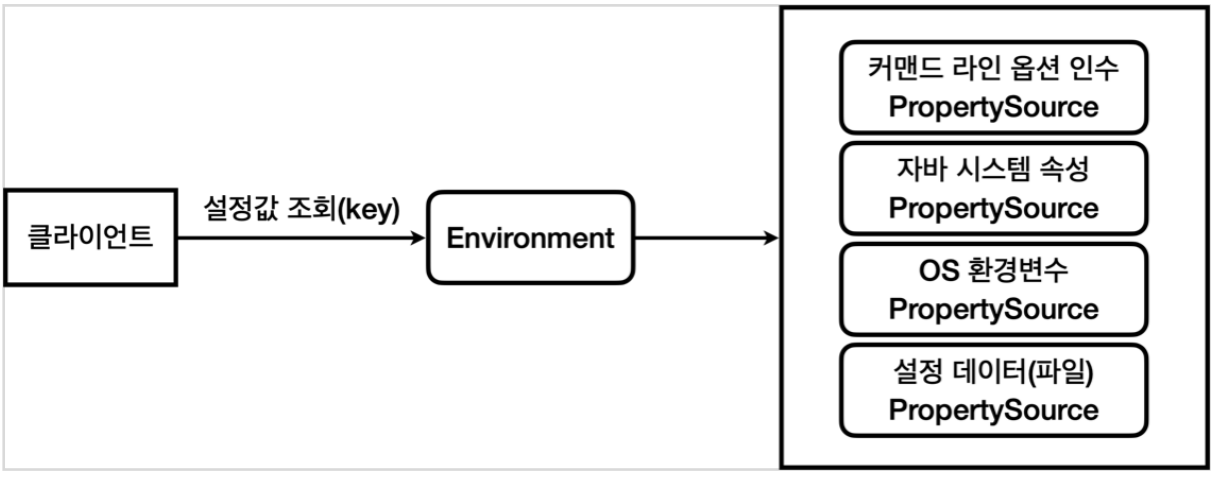
```
url=prod.db.com
username=prod_user
password=prod_pw
```

예를 들면 개발 서버와 운영 서버 각각에 `application.properties`라는 같은 이름의 파일을 준비해둔다. 그리고 애플리케이션 로딩 시점에 해당 파일을 읽어서 그 속에 있는 값들을 외부 설정값으로 사용하면 된다. 참고로 파일 이름이 같으므로 애플리케이션 코드는 그대로 유지할 수 있다.

스프링과 설정 데이터

개발자가 파일을 읽어서 설정값으로 사용할 수 있도록 개발을 해야겠지만, 스프링 부트는 이미 이런 부분을 이미 다 구현해두었다. 개발자는 `application.properties`라는 이름의 파일을 자바를 실행하는 위치에 만들어 두기만 하면 된다. 그러면 스프링이 해당 파일을 읽어서 사용할 수 있는 `PropertySource`의 구현체를 제공한다. 스프링에서는 이러한 `application.properties` 파일을 설정 데이터(Config data)라 한다.

당연히 설정 데이터도 `Environment`를 통해서 조회할 수 있다.



참고

지금부터 설명할 내용은 `application.properties` 대신에 `yml` 형식의 `application.yml`에도 동일하게 적용된다. `yml`과 `application.yml`은 뒤에 자세히 설명한다.

동작 확인

- `./gradlew clean build`
- `build/libs` 로 이동
- 해당 위치에 `application.properties` 파일 생성

```
url=dev.db.com
username=dev_user
password=dev_pw
```

- `java -jar external-0.0.1-SNAPSHOT.jar` 실행

실행 결과

```
env url=devdb
env username=dev_user
env password=dev_pw
```

이렇게 각각의 환경에 따라 설정 파일의 내용을 다르게 준비하면 된다. 덕분에 설정값의 내용이 많고 복잡해도 파일로 편리하게 관리할 수 있다.

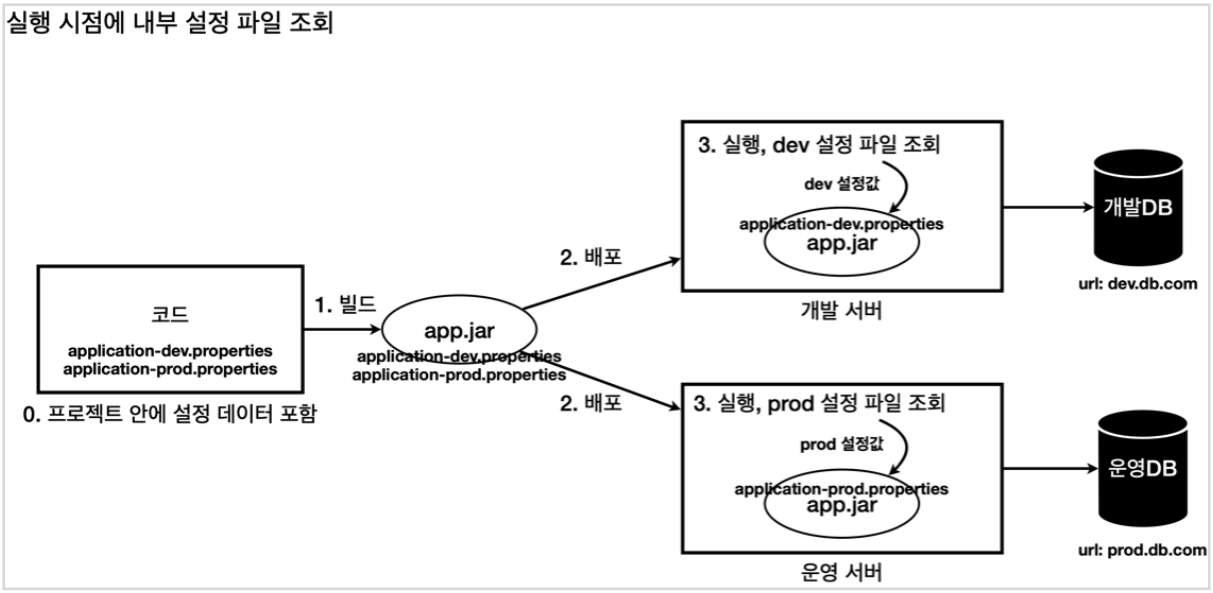
남은 문제

- 외부 설정을 별도의 파일로 관리하게 되면 설정 파일 자체를 관리하기 번거로운 문제가 발생한다.
- 서버가 10대면 변경사항이 있을 때 10대 서버의 설정 파일을 모두 각각 변경해야 하는 불편함이 있다.
- 설정 파일이 별도로 관리되기 때문에 설정값의 변경 이력을 확인하기 어렵다. 특히 설정값의 변경 이력이 프로젝트 코드들과 어떻게 영향을 주고 받는지 그 이력을 같이 확인하기 어렵다.

설정 데이터2 - 내부 파일 분리

설정 파일을 외부에 관리하는 것은 상당히 번거로운 일이다. 설정을 변경할 때 마다 서버에 들어가서 각각의 변경 사항을 수정해두어야 한다.(물론 이것을 자동화 하기 위해 노력할 수는 있다)

이 문제를 해결하는 간단한 방법은 설정 파일을 프로젝트 내부에 포함해서 관리하는 것이다. 그리고 빌드 시점에 함께 빌드되게 하는 것이다.
이렇게 하면 애플리케이션을 배포할 때 설정 파일의 변경 사항도 함께 배포할 수 있다. 쉽게 이야기해서 `jar` 하나로 설정 데이터까지 포함해서 관리하는 것이다.



- 0. 프로젝트 안에 소스 코드 뿐만 아니라 각 환경에 필요한 설정 데이터도 함께 포함해서 관리한다.
  - 개발용 설정 파일: `application-dev.properties`
  - 운영용 설정 파일: `application-prod.properties`
- 1. 빌드 시점에 개발, 운영 설정 파일을 모두 포함해서 빌드한다.
- 2. `app.jar`는 개발, 운영 두 설정 파일을 모두 가지고 배포된다.
- 3. 실행할 때 어떤 설정 데이터를 읽어야 할지 최소한의 구분은 필요하다.
  - 개발 환경이라면 `application-dev.properties`를 읽어야 하고,
  - 운영 환경이라면 `application-prod.properties`를 읽어야 한다.
  - 실행할 때 외부 설정을 사용해서 개발 서버는 `dev`라는 값을 제공하고, 운영 서버는 `prod`라는 값을 제공하자. 편의상 이 값을 프로필이라 하자.
    - `dev` 프로필이 넘어오면 `application-dev.properties`를 읽어서 사용한다.
    - `prod` 프로필이 넘어오면 `application-prod.properties`를 읽어서 사용한다.

외부 설정으로 넘어온 프로필 값이 `dev`라면 `application-dev.properties`를 읽고 `prod`라면 `application-prod.properties`를 읽어서 사용하면 된다. 스프링은 이미 설정 데이터를 내부에 파일로 분리해두고 외부 설정값(프로필)에 따라 각각 다른 파일을 읽는 방법을 다 구현해두었다.

스프링과 내부 설정 파일 읽기

`main/resources`에 다음 파일을 추가하자

`application-dev.properties` 개발 프로필에서 사용

```
url=dev.db.com
username=dev_user
password=dev_pw
```

`application-prod.properties` 운영 프로필에서 사용

```
url=prod.db.com
username=prod_user
password=prod_pw
```

프로필

스프링은 이런 곳에서 사용하기 위해 프로필이라는 개념을 지원한다.

`spring.profiles.active` 외부 설정에 값을 넣으면 해당 프로필을 사용한다고 판단한다.

그리고 프로필에 따라서 다음과 같은 규칙으로 해당 프로필에 맞는 내부 파일(설정 데이터)을 조회한다.

- `application-{profile}.properties`

예)

- `spring.profiles.active=dev`
  - `dev` 프로필이 활성화 되었다.
  - `application-dev.properties` 를 설정 데이터로 사용한다.
- `spring.profiles.active=prod`
  - `prod` 프로필이 활성화 되었다.
  - `application-prod.properties` 를 설정 데이터로 사용한다.

실행

- IDE에서 커맨드 라인 옵션 인수 실행
  - `--spring.profiles.active=dev`
- IDE에서 자바 시스템 속성 실행
  - `-Dspring.profiles.active=dev`
- Jar 실행
  - `./gradlew clean build`
  - `build/libs` 로 이동
  - `java -Dspring.profiles.active=dev -jar external-0.0.1-SNAPSHOT.jar`
  - `java -jar external-0.0.1-SNAPSHOT.jar --spring.profiles.active=dev`

dev 프로필로 실행 결과

```
The following 1 profile is active: "dev"
...
env url=devdb
env username=dev_user
env password=dev_pw
```

prod 프로필로 실행 결과

```
The following 1 profile is active: "prod"
...
env url=prod.db.com
env username=prod_user
env password=prod_pw
```

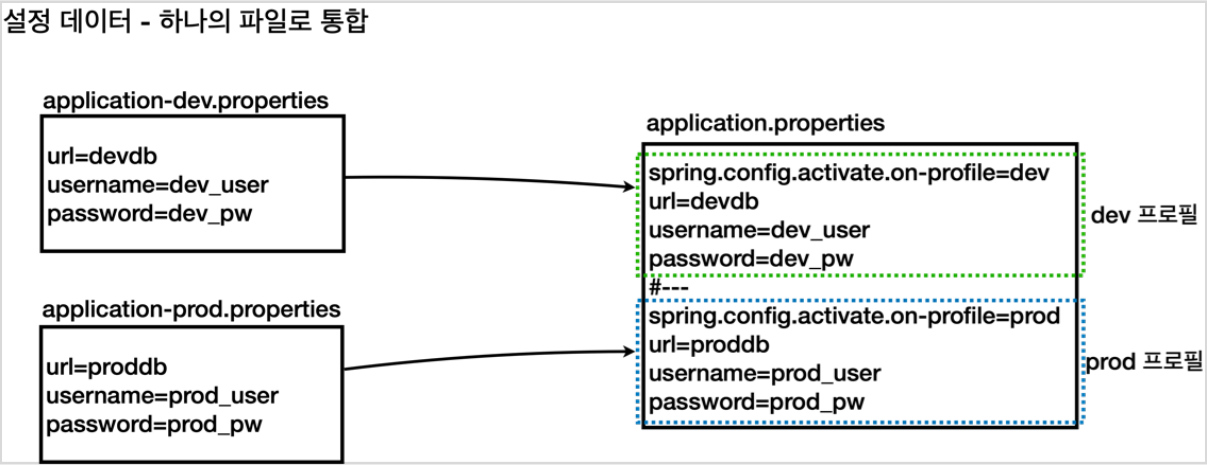
이제 설정 데이터를 프로젝트 안에서 함께 관리할 수 있게 되었고, 배포 시점에 설정 정보도 함께 배포된다.

남은 문제

설정 파일을 각각 분리해서 관리하면 한눈에 전체가 들어오지 않는 단점이 있다.

설정 데이터3 - 내부 파일 합체

설정 파일을 각각 분리해서 관리하면 한눈에 전체가 들어오지 않는 단점이 있다.
스프링은 이런 단점을 보완하기 위해 물리적인 하나의 파일 안에서 논리적으로 영역을 구분하는 방법을 제공한다.



- 기존에는 dev 환경은 `application-dev.properties`, prod 환경은 `application-prod.properties` 파일이 필요했다.
- 스프링은 하나의 `application.properties` 파일 안에서 논리적으로 영역을 구분하는 방법을 제공한다.
- `application.properties` 라는 하나의 파일 안에서 논리적으로 영역을 나눌 수 있다.
  - `application.properties` 구분 방법 `#---` 또는 `!---` (dash 3)
  - `application.yml` 구분 방법 `---` (dash 3)
- 그림의 오른쪽 `application.properties` 는 하나의 파일이지만 내부에 2개의 논리 문서로 구분되어 있다.

    - dev 프로필이 활성화 되면 상위 설정 데이터가 사용된다.
    - prod 프로필이 활성화 되면 하위 설정 데이터가 사용된다.
- 프로필에 따라 논리적으로 구분된 설정 데이터를 활성화 하는 방법
    - `spring.config.activate.on-profile`에 프로필 값 지정

설정 데이터를 하나의 파일로 통합하기

우선 기존 내용을 사용하지 않도록 정리해야 한다.

다음 내용은 사용하지 않도록 `#`을 사용해서 주석 처리하자.

`application-dev.properties` 주석 처리

```
#url=dev.db.com
#username=dev_user
#password=dev_pw
```

`application-prod.properties` 주석 처리

```
#url=prod.db.com
#username=prod_user
#password=prod_pw
```

`main/resources`에 다음 파일을 추가하자

`application.properties`

```
spring.config.activate.on-profile=dev
url=dev.db.com
username=dev_user
password=dev_pw
#---
spring.config.activate.on-profile=prod
url=prod.db.com
username=prod_user
password=prod_pw
```

주의!

- 속성 파일 구분 기호에는 선행 공백이 없어야 하며 정확히 3개의 하이픈 문자가 있어야 한다.

- 구분 기호 바로 앞과 뒤의 줄은 같은 주석 접두사가 아니어야 한다.

파일을 분할하는 `#---` 주석 위 아래는 주석을 적으면 안된다.

```
...
#
#---
...
```

- 분할 기호 위에 주석이 있다. 문서가 정상적으로 읽히지 않을 수 있다.

```
...
#---
#
...
```

- 분할 기호 아래에 주석이 있다. 문서가 정상적으로 읽히지 않을 수 있다.

실행

- 커맨드 라인 옵션 인수 실행
  - `--spring.profiles.active=dev`
- 자바 시스템 속성 실행
  - `-Dspring.profiles.active=dev`
- Jar 실행
  - `./gradlew clean build`
  - `build/libs` 로 이동
  - `java -Dspring.profiles.active=dev -jar external-0.0.1-SNAPSHOT.jar`
  - `java -jar external-0.0.1-SNAPSHOT.jar --spring.profiles.active=dev`

dev 프로필로 실행 결과

```
The following 1 profile is active: "dev"
...
env url=devdb
env username=dev_user
env password=dev_pw
```

prod 프로필로 실행 결과

```
The following 1 profile is active: "prod"
...
env url=prod.db.com
env username=prod_user
env password=prod_pw
```

이제 `application.properties` 라는 파일 하나에 통합해서 다양한 프로필의 설정 데이터를 관리할 수 있다.

우선순위 - 설정 데이터

`application.properties`

```
spring.config.activate.on-profile=dev
url=dev.db.com
username=dev_user
password=dev_pw
#---
spring.config.activate.on-profile=prod
url=prod.db.com
username=prod_user
password=prod_pw
```

이런 상태에서 만약 프로필을 적용하지 않는다면 어떻게 될까?
`--spring.profiles.active=dev` 이런 옵션을 지정하지 않는다는 뜻이다.
프로필을 적용하지 않고 실행하면 해당하는 프로필이 없으므로 키를 각각 조회하면 값은 `null` 이 된다.

실행 결과

```
No active profile set, falling back to 1 default profile: "default"
...
env url=null
env username=null
```

```
env password=null
```

실행 결과를 보면 첫줄에 활성 프로필이 없어서 `default` 라는 이름의 프로필이 활성화 되는 것을 확인할 수 있다. 프로필을 지정하지 않고 실행하면 스프링은 기본으로 `default` 라는 이름의 프로필을 사용한다.

기본값

내 PC에서 개발하는 것을 보통 로컬(`local`) 개발 환경이라 한다. 이때도 항상 프로필을 지정하면서 실행하는 것은 상당히 피곤할 것이다.
설정 데이터에는 기본값을 지정할 수 있는데, 프로필 지정과 무관하게 이 값은 항상 사용된다.

`application.properties` - 수정

```
url=local.db.com
username=local_user
password=local_pw
#---
spring.config.activate.on-profile=dev
url=dev.db.com
username=dev_user
password=dev_pw
#---
spring.config.activate.on-profile=prod
url=prod.db.com
username=prod_user
password=prod_pw
```

스프링은 문서를 위에서 아래로 순서대로 읽으면서 설정한다.

여기서 처음에 나오는 다음 논리 문서는 `spring.config.activate.on-profile` 와 같은 프로필 정보가 없다. 따라서 프로필과 무관하게 설정 데이터를 읽어서 사용한다. 이렇게 프로필 지정과 무관하게 사용되는 것을 기본값이라 한다.

```
url=local.db.com
username=local_user
password=local_pw
```

실행

프로필을 지정하지 않고 실행해보자.

실행 결과

```
No active profile set, falling back to 1 default profile: "default"
...
env url=local.db.com
env username=local_user
env password=local_pw
```

실행 결과 특정 프로필이 없기 때문에 기본값이 사용된다.

이번에는 프로필을 지정하고 실행해보자.

실행

- 커맨드 라인 옵션 인수 실행
  - `--spring.profiles.active=dev`
- 자바 시스템 속성 실행
  - `-Dspring.profiles.active=dev`

실행 결과

```
env url=dev.db.com
env username=dev_user
env password=dev_pw
```

프로필을 준 부분이 기본값 보다는 우선권을 가지는 것을 확인할 수 있다.

설정 데이터 적용 순서

이번에는 설정 데이터의 적용 순서에 대해서 좀 더 자세히 알아보자.

`application.properties`

```
url=local.db.com
username=local_user
password=local_pw
#---
spring.config.activate.on-profile=dev
url=dev.db.com
```

```
username=dev_user
password=dev_pw
#---
spring.config.activate.on-profile=prod
url=prod.db.com
username=prod_user
password=prod_pw
```

사실 스프링은 **단순하게 문서를 위에서 아래로 순서대로 읽으면서 사용할 값을 설정**한다.

- 1. 스프링은 순서상 위에 있는 `local` 관련 논리 문서의 데이터들을 읽어서 설정한다. 여기에는 `spring.config.activate.on-profile` 와 같은 별도의 프로필을 지정하지 않았기 때문에 프로필과 무관하게 항상 값을 사용하도록 설정한다.

```
url=local.db.com
username=local_user
password=local_pw
```

- 2. 스프링은 그 다음 순서로 `dev` 관련 논리 문서를 읽는데 만약 `dev` 프로필이 설정되어있다면 기존 데이터를 `dev` 관련 논리 문서의 값으로 대체한다. 물론 `dev` 프로필을 사용하지 않는다면 `dev` 관련 논리 문서는 무시되고, 그 값도 사용하지 않는다.

```
url=local.db.com    -> dev.db.com
username=local_user -> dev_user
password=local_pw   -> dev_pw
```

- 3. 스프링은 그 다음 순서로 `prod` 관련 논리 문서를 읽는데 만약 `prod` 프로필이 설정되어있다면 기존 데이터를 `prod` 관련 논리 문서의 값으로 대체한다. 물론 `prod` 프로필을 사용하지 않는다면 `prod` 관련 논리 문서는 무시되고, 그 값도 사용하지 않는다.

```
url=dev.db.com    -> prod.db.com
username=dev_user -> prod_user
password=dev_pw   -> prod_pw
```

참고로 프로필을 한번에 둘 이상 설정하는 것도 가능하다.

- `--spring.profiles.active=dev,prod`

순서대로 설정 확인

극단적인 예시를 통해서 순서를 확실히 이해해보자.

`application.properties` - 수정

```
url=local.db.com
username=local_user
password=local_pw
#---
spring.config.activate.on-profile=dev
url=dev.db.com
username=dev_user
password=dev_pw
#---
spring.config.activate.on-profile=prod
url=prod.db.com
username=prod_user
password=prod_pw
#---
url=hello.db.com
```

스프링이 설정 파일을 위에서 아래로 순서대로 읽어서 사용할 값을 설정한다는 것은 이 예제를 실행해보면 확실히 이해할 수 있다.

결과가 어떻게 나올지 먼저 상상해보자.

- 1. 스프링은 처음에 `local` 관련 논리 문서의 데이터들을 읽어서 설정한다. 여기에는 별도의 프로필을 지정하지 않았기 때문에 프로필과 무관하게 항상 값이 설정된다.
- 2. 스프링은 그 다음 순서로 `dev` 관련 논리 문서를 읽는데 만약 `dev` 프로필이 설정되어있다면 기존 데이터를 `dev` 관련 논리 문서의 값으로 대체한다.
- 3. 스프링은 그 다음 순서로 `prod` 관련 논리 문서를 읽는데 만약 `prod` 프로필이 설정되어있다면 기존 데이터를 `prod` 관련 논리 문서의 값으로 대체한다.
- 4. 스프링은 마지막으로 `hello` 관련 논리 문서의 데이터들을 읽어서 설정한다. 여기에는 별도의 프로필을 지정하지 않았기 때문에 프로필과 무관하게 항상 값이 설정된다.

위에서 아래로 순서대로 실행하는데, 마지막에는 프로필이 없기 때문에 항상 마지막의 값들을 적용하게 된다.

만약 `prod` 프로필을 사용한다면 다음과 같이 설정된다.

```
The following 1 profile is active: "prod"
...
url=hello.db.com
username=prod_user
password=prod_pw
```

물론 이렇게 사용하는 것은 의미가 없다. 이해를 돕기 위해 이렇게 극단적인 예제를 사용했다. 보통은 기본값을 처음에 두고 그 다음에 프로필이 필요한 논리 문서들을 둔다.

정리하면 다음과 같다.

- 단순하게 문서를 위에서 아래로 순서대로 읽으면서 값을 설정한다. 이때 기존 데이터가 있으면 덮어쓴다.
- 논리 문서에 `spring.config.activate.on-profile` 옵션이 있으면 해당 프로필을 사용할 때만 논리 문서를 적용한다.

속성 부분 적용

만약 프로필에서 일부 내용만 교체하면 어떻게 되는지 알아보자.

```
url=local.db.com
username=local_user
password=local_pw
#---
spring.config.activate.on-profile=dev
url=dev.db.com
```

만약 다음과 같이 적용하고 `dev` 프로필을 사용하면 어떤 결과가 나올까?

먼저 순서대로 `local` 관련 정보가 입력된다.

```
url=local.db.com
username=local_user
password=local_pw
```

이후에 `dev` 관련 문서를 읽게 되는데, `dev` 프로필이 활성화 되어 있다고 가정하자. `dev` 관련 문서에서는 `url=dev.db.com` 만 설정한다. 이 경우 기존에 설정값에서 `url` 만 변경된다.

```
The following 1 profile is active: "dev"
...
url=dev.db.com
username=local_user
password=local_pw
```

최종적으로 `url` 부분은 `dev.db.com` 으로 `dev` 프로필에서 적용한 것이 반영되고, 나머지는 처음에 입력한 기본값이 유지된다.

스프링의 우선순위에 따른 설정값은 대부분 지금과 같이 기존 데이터를 변경하는 방식으로 적용된다.

우선순위 - 전체

스프링 부트는 같은 애플리케이션 코드를 유지하면서 다양한 외부 설정을 사용할 수 있도록 지원한다.

> 외부 설정에 대한 우선순위 - 스프링 공식 문서
> https://docs.spring.io/spring-boot/docs/current/reference/html/features.html#features.external-config

우선순위는 위에서 아래로 적용된다. 아래가 더 우선순위가 높다.

자주 사용하는 우선순위

- 설정 데이터(`application.properties`)
- OS 환경변수
- 자바 시스템 속성
- 커맨드 라인 옵션 인수
- `@TestPropertySource` (테스트에서 사용)

설정 데이터 우선순위

- jar 내부 `application.properties`
- jar 내부 프로필 적용 파일 `application-{profile}.properties`
- jar 외부 `application.properties`
- jar 외부 프로필 적용 파일 `application-{profile}.properties`

> 설정 데이터 우선순위 - 스프링 공식 문서
> https://docs.spring.io/spring-boot/docs/current/reference/html/

> features.html#features.external-config.files

우선순위 이해 방법

우선순위는 상식 선에서 딱 2가지만 생각하면 된다.

- 더 유연한 것이 우선권을 가진다. (변경하기 어려운 파일 보다 실행시 원하는 값을 줄 수 있는 자바 시스템 속성이 더 우선권을 가진다.)
- 범위가 넓은 것 보다 좁은 것이 우선권을 가진다.
  - OS 환경변수 보다 자바 시스템 속성이 우선권이 있다.
  - 자바 시스템 속성 보다 커맨드 라인 옵션 인수가 우선권이 있다.

추가 또는 변경되는 방식

`Environment`를 통해서 조회하는 관점에서 보면 외부 설정값들은 계속 추가되거나 기존 값을 덮어서 변경하는 것 처럼 보인다. 물론 실제 값을 덮어서 변경하는 것은 아니고, 우선순위가 높은 값이 조회되는 것이다. 그런데 이렇게 이해하면 개념적으로 더 쉽게 이해할 수 있다.

예를 들어서 설정 데이터(`application.properties`)에 다음과 같이 설정했다.

application.properties

```
url=local.db.com
```

자바 시스템 속성을 다음과 같이 적용했다.

자바 시스템 속성 추가

`-Dusername=local_user`

조회 결과

```
url=local.db.com
username=local_user
```

자바 시스템 속성에서 기존에 없던 키 값을 추가했기 때문에 속성이 추가되었다.

커맨드 라인 옵션 인수를 다음과 같이 적용했다.

커맨드 라인 옵션 인수 추가

`--url=dev.db.com`

조회 결과

```
url=dev.db.com
```

```
username=local_user
```

커맨드 라인 옵션 인수는 기존에 있던 `url` 이라는 키 값을 사용했기 때문에 기존에 있던 값이 새로운 값으로 변경되었다.

정리

이렇게 우선순위에 따라서 설정을 추가하거나 변경하는 방식은 상당히 편리하면서도 유연한 구조를 만들어준다. 실무에서 대부분의 개발자들은 `applicaiton.properties` 에 외부 설정값들을 보관한다. 이렇게 설정 데이터를 기본으로 사용하다가 일부 속성을 변경할 필요가 있다면 더 높은 우선순위를 가지는 자바 시스템 속성이나 커맨드 라인 옵션 인수를 사용하면 되는 것이다.
또는 기본적으로 `application.properties` 를 jar 내부에 내장하고 있다가, 특별한 환경에서는 `application.properties` 를 외부 파일로 새로 만들고 변경하고 싶은 일부 속성만 입력해서 변경하는 것도 가능하다.

정리